DISS. ETH NO. 19163

DATA STREAM PROCESSING ON EMBEDDED DEVICES

A dissertation submitted to

ETH ZURICH

for the degree of

Doctor of Sciences

presented by

RENÉ MÜLLER

Master of Science ETH in Computer Science, ETH Zurich

September 24, 1978

citizen of
Walterswil, Solothurn

accepted on the recommendation of

Prof. Dr. Gustavo Alonso, examiner
Prof. Dr. Donald Kossmann, co-examiner
Prof. Dr. Samuel R. Madden, co-examiner
Dr. Satnam Singh, co-examiner

2010

Typeset with LaTeX
Printer: ETH Zurich
Printed in Switzerland

Abstract

Over the years, online processing of data has become more important. In inventory management, monitoring and financial applications data is generated in streams that are processed on the fly instead of being stored in a repository and processed offline later. As data volumes increase, processing has to be offloaded from the central stream processing engine. Processing either has to be moved to the data sources or to specialized accelerators placed between data source and engine. Stream processing platforms thus become heterogeneous. The problem is how to optimize query execution when it spans different streaming systems.

This thesis discusses stream processing on two different platforms: wireless sensor networks and field-programmable gate arrays (FPGAs). Both are typically connected to traditional server-class streaming processors. Driven by the different optimization goals (e.g., throughput, latency, resource consumption) operators have to be carefully placed. In some cases it is more efficient to place operators, for example, into the sensor network. Other operators that require a substantial amount of state are better placed on the server. The problems addressed in this work is the design of the underlying execution platforms that facilitate the operator placement, strategies, and cost-models used for optimization.

In the first part of the thesis *SwissQM* is presented. *SwissQM* is a stream processing platform for wireless sensor networks and is based on a small virtual machine that is deployed on resource-constrained sensor nodes. Declarative queries submitted by the user are translated into short bytecode sequences and are disseminated in the network. The bytecode programs implement streaming operators that are executed at the data source in the network. The remaining operators of the queries are placed onto the base station that connects the wireless sensor network to, e.g., the Internet. The base station also performs multi-query optimization of multiple user queries that are executed concurrently. The thesis proposes an energy-based cost model and presents optimization strategies that rely on the cost model. Multi-query optimization maximizes utilization of the network infrastructure such that expensive deployments can be accessed by several users and applications concurrently.

The second part of the thesis applies the same techniques to FPGAs, i.e., the automated compilation of queries into digital circuits that can be placed onto

FPGAs. First, a stream processing algebra is defined, which is later used to express stream execution plans. *Glacier*, a library of hardware components and a set of translation rules is presented as compositional approach to translate queries into FPGAs circuits. A key property of the generated hardware circuits is the well-defined performance behavior. The dissertation also presents optimization techniques to trade-off various parameters on the FPGAs such as throughput, latency, and chip space.

The thesis proposes a common solution for both the sensor network and the FPGA domain, which allows users to specify queries in the same declarative language. As such, it increases the level of abstraction on both platforms from embedded systems programming and hardware description languages to a high-level language. As a domain-specific language it makes this technology available to broader range of users.

Kurzfassung

Die Online-Datenverarbeitung hat in den letzten Jahren an Bedeutung gewonnen. Lagerverwaltungs- und Überwachungssysteme sowie Anwendungen aus dem Finanzsektor verarbeiten Daten, sogenannte Datenströme, zunehmend ohne vorgängiges Speichern. Mit steigendem Datenvolumen muss ein Teil der Verarbeitung vom Streamprozessor ausgelagert werden. Die Verarbeitung wird entweder in die Datenquelle selbst oder auf einen Hardware-Beschleuniger im Datenpfad zwischen Quelle und Streamprozessor ausgelagert. Die Streamverarbeitung erfolgt somit auf einer heterogenen Plattform. Das Problem besteht nun darin, die Verarbeitung zu Optimieren, auch wenn diese unterschiedliche Systeme umschliesst.

Im Rahmen dieser Dissertation werden zwei Streamverarbeitungplattformen vorgestellt: drahtlose Sensornetzwerke und FPGAs (Field-Programmable Gate Arrays). Beide sind üblicherweise mit einem bestehenden, Server-basierten Streamverarbeitungssystem verbunden. Unterschiedliche Optimierungsziele (z.B. Durchsatz, Latenz, Resourcenverbrauch) bestimmten das Platzieren der Operatoren. In bestimmten Fällen ist es effizienter, einen Operator in das Sensornetzwerk zu verschieben. In anderen Fällen werden Operatoren, die üblicherweise sehr speicherintensiv sind, sinnvollerweise auf dem leistungsfähigeren Server ausgeführt. Das Design der Ausführungsplattform, welche das Verschieben von Operatoren ermöglicht, sowie die entsprechenden Platzierungsstrategien und Kostenmodelle für die Optimierung werden in dieser Dissertation behandelt.

Im ersten Teil der Arbeit wird *SwissQM* vorgestellt. *SwissQM* ist eine Streamverarbeitungsplattform für drahtlose Sensornetzwerke, die auf einer virtuellen Maschine (VM) basiert. Diese VM wird auf den Sensorknoten installiert. Deklarative Anfragen, die Anwender an das System stellen, werden in kurze Bytecodesequenzen übersetzt und im Netzwerk verteilt. Die Bytecodeprogramme implementieren diejenigen Streamoperatoren, welche ins Sensornetzwerk verschoben wurden. Die übrigen Verarbeitungsschritte werden auf der Basisstation ausgeführt, welche das Sensornetzwerk mit beispielsweise dem Internet verbindet. Die Basisstation führt ebenfalls Multiquery-Optimierung durch, d.h., parallele Anfragen von mehreren Anwendern werden optimiert und gleichzeitig ausgeführt. Dadurch kann die üblicherweise äussert teure Installation von mehreren Benutzern oder Endanwendungen gleichzeitig verwendet und die Kosten entsprechend aufgeteilt werden.

Im zweiten Teil werden diese Techniken auf FPGAs angewendet, d.h., die automatische Übersetzung von Anfragen in digitale Schaltung für FPGAs. Zuerst wird eine Streamalgebra eingeführt, anhand deren Anfragepläne dargestellt werden können. Anschliessend wird *Glacier* vorgestellt. Es handelt sich dabei um eine Bibliothek aus Hardwarekomponenten und einen Satz von Regeln, anhand derer die Anfragepläne durch Einsatz von Komposition in FPGA-Schaltungen übersetzt werden können. Eine wichtige Eigenschaft der erzeugten Hardwareschaltungen ist das wohldefinierte Verhalten bezüglich Latenz und Durchsatz. Die Arbeit stellt weiter Optimierungstechniken vor, die es erlauben, verschiedene Parameter wie beispielsweise, Durchsatz, Latenz und die erforderliche Chip-Fläche gegeneinander abzuwägen.

In dieser Dissertation wird eine allgemeine Lösung vorstellt, die sowohl für Sensornetzwerke als auch für FPGAs eingesetzt werden kann. Benutzer können Anfragen in deklarativer Form stellen. Als solche, entspricht diese einer domänenspezifischen Sprache, welche den Abstraktionsgrad vom Programmieren eingebetteter Systeme und dem Design von Schaltungen auf die Ebene einer Hochsprache erhöht. Das macht die Plattformen für einen weiteren Benutzerkreis verwendbar.

Contents

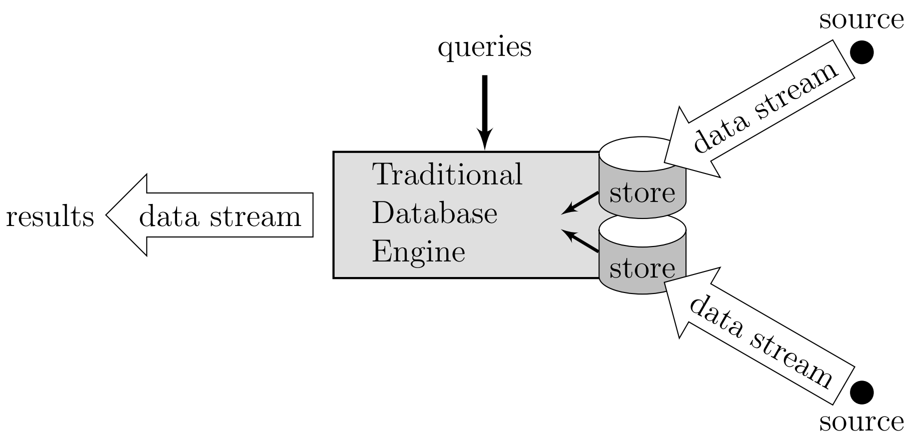
1 Introduction

1.1 Motivation

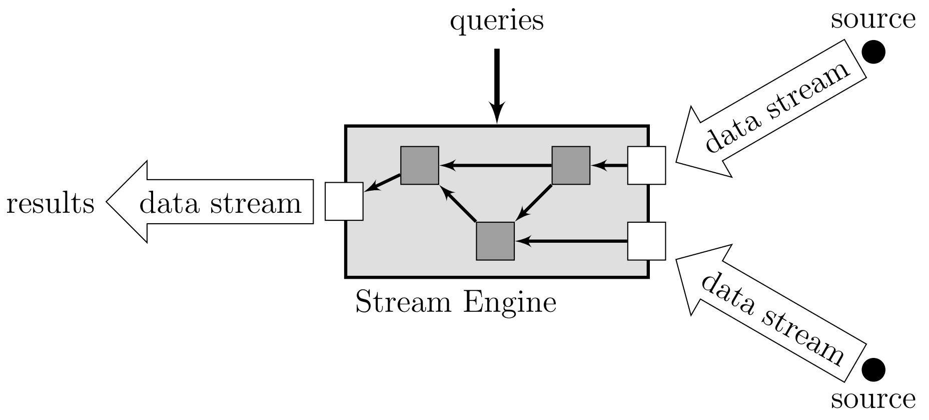
Data Stream Systems. The proliferation of information systems as led to high data volumes that have to be transferred, processed, and stored. Databases increase in size and number, ranging from Google-scale systems to a large number of small federated databases in small and medium-sized enterprises. Next to traditional databases, data is increasingly available as (data) streams. Data streams are generated from many different sources, such as deeply embedded sensors, network monitoring systems, and financial systems.

One approach to implement data stream processing is to use existing database technology as illustrated in Figure 1.1(a). The streams are stored in tables of a traditional database system such that conventional query processing can be performed through queries. There is, however, a steadily increasing amount of data produced. Figure 1.2 shows the development of the generated data volume and storage capacity as predicted by Gantz et al. [GCM+08]. There is a widening gap between the information generated and the available storage. This trend has led to *data stream processing systems.* Data streams are directly fed into the engine as shown in Figure 1.1(b) and processed on the fly through in a network of data operators. There are already many commercial stream processing systems available today [Str, Mic, Ora, Syb, Tru].

The approach used in these systems, as sketched in Figure 1.1(b), is based on two assumptions: *(a)* the streams can be brought into the engine, that is, there



(a) Streams into databases



(b) In-flight processing of data streams

Figure 1.1: Architectures for stream processing systems

is enough I/O bandwidth available, and *(b)* the engine has enough resources to process the data.

If this is not the case, part of the processing has to be offloaded. Stream processing typically contains filter operations, e.g., predicate queries, that select only a subset of the data and, hence, reduce data. This type of data reduction functionality can be ideally offloaded, as it not only removes part of the processing load from the central engine but also because it reduces the amount of data that needs to transferred. In practice, offloading means pushing part of the processing to the data sources as shown in Figure 1.3(a) or to an intermediary between the sources and the engine. Such an intermediary can be considered an accelerator that that sits on the data path between source and engine (Figure 1.3(b)). In the

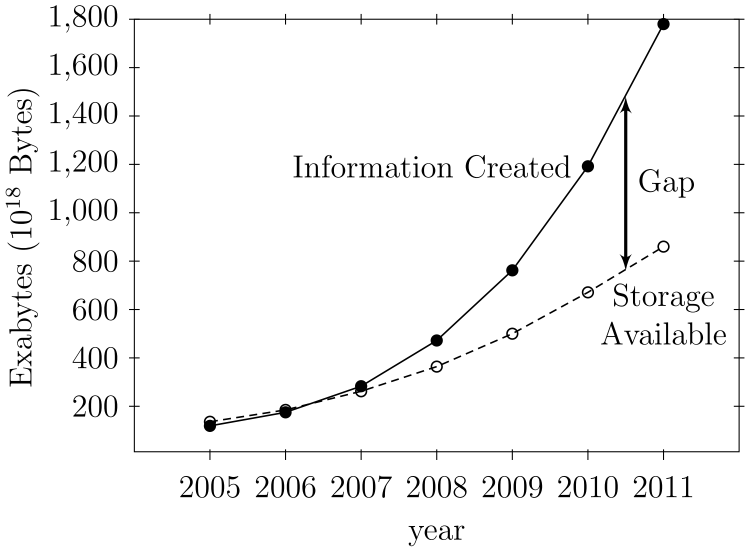


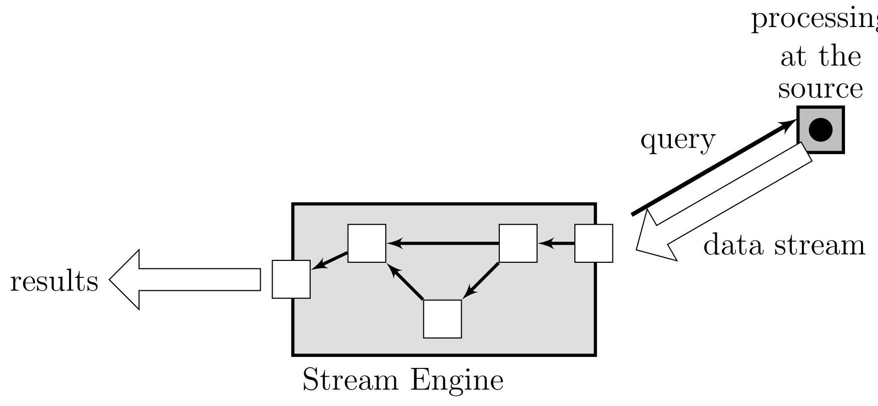
Figure 1.2: Data Volume vs. Storage Capacity by Gantz et al. [GCM+08]

context of database processing these accelerators have been used on the I/O path to disks [Net09] or to the network [Net].

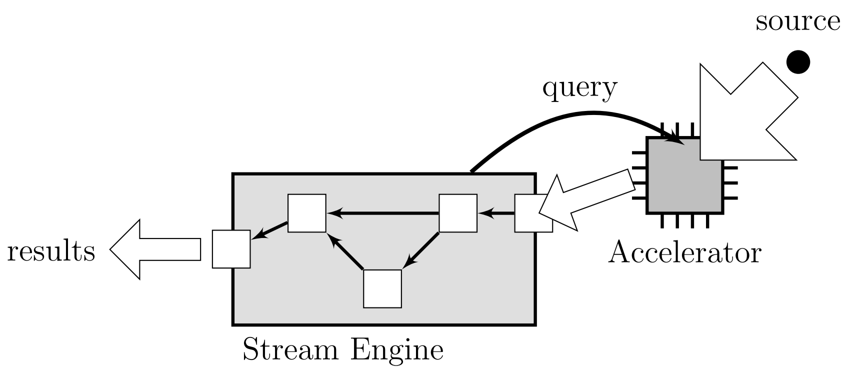
Streaming systems thus become heterogeneous as they not only include the core engine but also the accelerator and the actual data source. A query planner and optimizer needs to span multiple entities: the engine, an accelerator, and the data source. The ultimate goal is to have a common solution that autonomously distributes operations onto the elements of a such a heterogeneous system (Figure 1.4).

This thesis focuses on data stream processing on those heterogeneous platforms. It studies the problem of how the different characteristics of the individual subsystem can be be hidden from the user such that a general interface can be provided. It contributes to a unified stream processing solution that considers the costs and properties of the underlying platforms when generating query execution plans. Heterogeneous systems typically have very different cost models. An optimizer thus has to be able to decide *where* to place an operator in the query execution graph. This thesis contributes towards such a solution by extending stream processing onto two different execution platforms: *wireless sensor networks* (WSNs) and *field-programmable gate arrays* (FPGAs).

Embedded Systems. Moore's Law has not only resulted in faster CPUs and multi-core technology on commodity computers but had an impact on the embedded computing space. Small microcomputer systems equipped with sensor and ra-



(a) Pushing queries to the data source



(b) Processing streams on the data path

Figure 1.3: Heterogeneous architectures for stream processing systems

dio communication modules and powered by batteries have been deployed as wireless sensor networks [JOW+02,KDD04,LBV06,MCP+02,SJ04,THGT07,TPS+05]. They are not only capturing sensor data. Their microprocessors provide sufficient computation power such that they can also process data (e.g., filtering, smoothing, etc.) and selectively report data if the sensors detect an interesting pattern. As such they can contribute to stream processing. In the first part of this dissertation the wireless sensor networks are used for offloading query processing to the data source as illustrated in Figure 1.3(a).

The second part of the thesis discusses the accelerator approach shown in Figure 1.3(b). The accelerator is implemented on a field-programmable gate array (FPGA). FPGAs are "programmable" hardware that provide a number of logic gates that can be configured to implement any arbitrary digital circuit. They are widely used in the embedded system world. Programming, in this context means

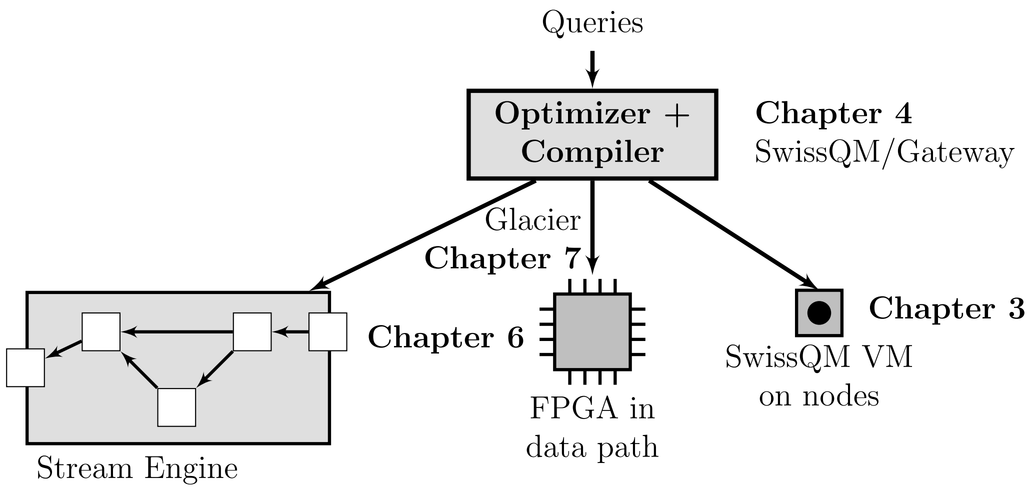


Figure 1.4: Heterogeneous stream engine with offloading to data source and data path accelerator

essentially how to map data processing steps into digital circuits and onto FPGAs. FPGAs offer a high flexibility at a very low level and, unlike sensor networks, they cannot be programmed in the traditional way. Furthermore, they exhibit very different properties that have to be accounted for when designing FPGA circuits.

The thesis describes the different execution platforms and cost models used for query compilation for sensor networks and FPGAs. The goal for sensor network is maximizing the utilization of a deployment, i.e., providing concurrent access to multiple users, while minimizing the energy consumption or equivalently maximizing battery life. The optimization goal for FPGA-based accelerators is the minimization of latency or maximizing throughput. At the same time chip consumption has to be minimized.

1.2 Contributions

In this dissertation the heterogeneous system shown in Figure 1.4 is envisioned. Users can submit queries, which are then partitioned on the underlying execution platforms, the main-memory stream engine, an FPGA, and the wireless sensor network. The type of execution plans depends on the platform. For FPGAs they are hardware circuits, for the sensor network they are bytecode programs. In summary, the thesis makes following contributions:

Virtual Machine. As embedded systems, sensor networks are difficult to program. To large extend low-level programming is needed. In our approach we increase the abstraction level by providing a virtual machine that is deployed in the sensor network. We propose *SwissQM*, a stack-based virtual machine for resource-constrained sensor nodes. SwissQM was specifically designed and tailored to facilitate stream processing tasks. SwissQM can be used as application-specific virtual machine. Its instruction set can be extended with application-specific instructions. In general, SwissQM raises the abstraction level of network programming. Instead of using low-level programming languages, data processing tasks can be implemented using high-level bytecode macros.

Query-to-Bytecode Compiler. The SwissQM virtual machine is used as an execution platform for streaming queries. Users can submit data collection tasks in the form of continuous queries to a gateway device that is connected to the sensor network. Queries are automatically compiled into bytecode sequences and are disseminated in the network. This first of all provides an even higher level of abstraction to the end-user. Since a declarative interface is used, no program has to be written at all. Second, the dissemination of execution plans as bytecode programs is very efficient and very flexible. For example, complex expressions and user-defined function can easily be compiled into bytecode.

Cost Models. Accurate cost models are built from a detailed analysis of the characteristics of the underlying hardware technology. For sensor networks we provide an energy-based cost model. Its parameters are identified by measurements of an real hardware.

Multi-query Optimization Strategies. Using the cost model we provide different strategies to perform multi-query optimization to maximize utilization in sensor networks. We evaluate the strategies using different workloads and provide the resulting heuristics for an optimizer.

Characterization of FPGA Computing. We use *sorting networks* as a use case to characterize the properties of FPGAs as a processing platform. We present and validate a cost model of the chip usage in this context. We discuss the trade-offs of the different attachment methods of FPGAs to conventional computing systems. We also provide a set of design guidelines for the design of FPGA-based computing solutions.

Query-to-Hardware Compiler. We introduce a compositional algebra for data stream processing that can be used to express streaming queries. The algebra consists of well-defined operators. The thesis describes *Glacier*, a hardware component library, that provides the necessary operators. The library provides operators for selection, projection, windowing, grouping, and window joins. Finally, a set of translation rules is presented that can be used to automatically translate plans expressed in this algebra into hardware circuits for FPGAs. This translation step for queries to FPGA-circuits is similar to the query-to-bytecode translation for sensor networks. In both cases, the abstraction level is raised from low-level, in case for FPGAs gate-level, programming to a high-level declarative interface.

1.3 Structure

The outline of the dissertation follows the architecture Figure 1.4. In detail, the dissertation is structured as follows:

Chapter 2 introduces wireless sensor networks. It describes the hardware platform, the constraints, and limitations. Through experiments we describe the behavior of the wireless communication channel and the implication on the design of the software stack.

Chapter 3 describes the design and implementation of the SwissQM virtual machine and the gateway system that performs the query-to-bytecode translation. The power and flexibility of SwissQM is illustrated through several examples of increasing complexity. The chapter also provides an evaluation of SwissQM in terms of execution performance, bytecode flexibility, and communication cost for program distribution.

Chapter 4 first introduces an energy-based cost model for sensor networks. Then it describes how several user queries can be combined into a single query. The problem of merging queries with different sampling rates is also addressed. In our approach, multi-query optimization is implemented in the query processor at the gateway. The query processor considers both execution costs as well costs for propagating workload updates into the network. Finally, different multi-query optimization strategies are presented and evaluated using sets of random queries. This completes the declarative query execution platform SwissQM for wireless sensor networks.

Chapter 5 opens up the venue for the FPGA part. It provides an introduction into FPGAs and describes the building blocks that are used later in chapters 6 and 7.

Chapter 6 evaluates FPGAs as a computing platform using sorting networks as an example. It describes how sorting networks can be implemented on FPGA logic. The impact of different implementation strategies on the overall performance and space requirement is analyzed. Two different use cases are provided that show how the FPGA can be used in a complete system. For both systems we measure the end-to-end performance. The chapter provides insight in the design space of FPGA solutions. Techniques developed and evaluated in this chapter serve as a basis for the design of query-to-hardware compiler.

Chapter 7 introduces the operator algebra for streaming queries. A set of translation rules is provided that can be used in a compositional way to translate queries, i.e., query plans expressed in this algebra, into hardware circuits. A number of operators is presented that are part of our *Glacier* component library. We provide optimization guidelines for the circuits that allow balancing latency against throughput. This chapter also presents *Handshake Join* our novel approach to window-based stream joins. An evaluation shows that this pipelining-based technique has excellent scalability characteristics. The chapter completes the FPGA aspect of the heterogeneous solution outlined in Figure 1.4.

Chapter 8 takes up the original vision and contrasts it with the results obtained from SwissQM and Glacier. The chapter summarizes the thesis and discusses possible future work directions.

Part I

Data Stream Processing in Wireless Sensor Networks

2 Data Processing in Wireless Sensor Networks

Thanks to Moore's law, the IT industry has seen computers getting smaller, cheaper, and more powerful during the last three decades. Now and in the coming years, computers are increasingly being enhanced with powerful sensing devices. Following Mark Weiser's *Pervasive Computing* paradigm [Wei99] computing devices are embedded into the physical world. They are extended with sensors and combined into *sensor networks* to tackle larger data acquisition tasks, e.g., monitoring the behavior of a large population of animals [MCP^{+}02] or the climate of a large territory [THGT07]. Commercial applications of sensor networks include supply chain management [KDD04], support of elderly people [SJ04], and security facilities for manufacturing sites and homes.

Sensor networks typically employ wireless communication although there are several examples of wired sensor networks too. For example, wired sensors can be found in structural health monitoring applications and cars. In the context of the dissertation the nature of the wireless communication is explicitly considered. Properties related to sampling, and the quality of sensor readings, however, are equally important for wired sensor networks. In this dissertation only *wireless sensor networks* (WSN) are considered. For clarity the term "sensor networks" is often used interchangeably for WSNs.

Despite the numerous existing use cases, building and deploying sensor networks remain elusive and difficult tasks. Most existing deployments use application-specific, *hard-coded* software. Typically, there is little support for data independence, high level abstractions, multiple users, and, above all, integration with the higher layers of the data processing chain. Systems are also very rigid on how sensors can be programmed and what the sensor network can do.

In this chapter, we describe the characteristics of wireless sensor networks and introduce data processing for wireless sensor networks. The discussion begins with the different hardware platforms. Then we illustrate the properties of wireless communication and the resulting network architecture before we turn to data processing and declarative interfaces to WSNs, in particular, query processing. Section 2.3 covers the relevant related work.

2.1 Wireless Sensor Nodes

Wireless sensor networks consist of many—ranging from 10s to 100s, possibly 1000s—nodes. Each node has limited computing and communication capabilities. The nodes build a network using wireless communication that collectively perform a data acquisition or event detection task.

2.1.1 Hardware

The use of tiny wirelessly connected computing devices equipped with sensors was envisioned as *Smart Dust* by Hahn et al. in 1999 [KKP99]. They explored whether an autonomous sensing, computing, and communication system can packed into a device of a cubic-millimeter volume. Considering the small size they referred to the nodes as *motes* (small particles or specks). Their prototype was a microelectromechanical system (MEMS) with an optical communication component. Communication was implemented through small movable mirrors that deflect and modulate a laser beam. While "Smart Dust" was an interesting case study the design was highly experimental. Sensor nodes used later on were significantly larger in size, had more powerful microcontrollers, and used radio communication instead. Although roughly the size of a matchbox those later nodes were still called *motes*.

Many different note platforms are deployed today. In this dissertation the three different hardware types shown in Table 2.1 were used. The platforms have different properties. They contain different microcontrollers, communication chips and sensors. The microcontrollers used in Mica2 and Tmote Sky nodes provide comparable processing power. The amount of volatile memory used for program data is in the order of a few kilobytes. Tens of kilobytes of flash memory are available for program storage. The amount of RAM and flash memory directly limits the complexity of the software stack that can run on a node. Specific embedded operating systems and applications have the be carefully designed in order to fit onto these resource constrained devices. The newer Intel Imote2 node

Table 2.1: Characteristics of sensor node types used in this dissertation

	Mica2	**Tmote Sky**	**Imote2**
microcontroller	Atmel ATmega128L	TI MSP430	Intel PXA271
RAM	4 kB	10 kB	32 MB
Flash storage	128 kB	48 kB	32 MB
radio chip	CC1000	CC2420	CC2420
communication	916 MHz, FSK	IEEE 802.15.4	IEEE 802.15.4
radio bandwidth	19.2 kbit/sec	250 kbit/sec	250 kbit/sec
max. power	5 dBm (3 mW)	0 dBm (1 mW)	0 dBm (1 mW)
sensors	light temperature microphone (separate board)	light photo light temperature humidity (onboard)	light photo light 3D acceleration 4 channel ADC (separate board)
manufacturer	Crossbow	moteiv	Intel/Crossbow
price	CHF 290	CHF 140	CHF 640
introduction	2002	2005	2006

shown in Table 2.1 contains a full-featured ARM CPU (PXA271) and significantly more memory. It is even able to run Linux and applications that use the Microsoft .NET Micro Framework.

Wireless sensor network typically operate in unlicensed ISM frequency bands. The motes shown in Table 2.1 use two different communication technologies. The older Mica2 platform uses a proprietary radio that operates in the 900 MHz ISM band. It uses *Frequency-shift Keying* (FSK) and Manchester coding. Operating at a symbol rate of 38.4 kbaud the resulting gross bit rate is 19.2 kbit/sec. The newer Tmote Sky and Imote2 nodes use a more energy-efficient radio technology and a physical layer that follows the IEEE 802.15.4 standard for personal area networks. Relying on a common standard permits interoperability between device families. The CC2420 transceivers operands on the 2.4 GHz ISM band that is shared with traditional 802.11bgn wireless LAN and Bluetooth. The transceiver complexity of the CC2420 is significantly larger than of the CC1000. It uses *Direct-Sequence Spread Spectrum* (DSSS) communication. Four bits are encoded into a pseudo-random sequence consisting of 32 chips. The chip sequences are modulated using *Offset Quadrature Phase-shift Keying* (OQPSK) at a rate of 2 MChips/sec resulting in a gross bandwidth of 250 kbit/sec. The use of spread-spectrum communication in the CC2420 instead narrow-band radio in the CC1000 provides additional robustness against noise and interference. This reduces the bit error rate and increases the overall reliability of the communication channels in the wireless network. Additional robustness is necessary as the same frequency band is occupied by different radio technologies (802.11 and Bluetooth, microwave ovens, etc.). However, the wireless communication channel is nevertheless much less reliable than communication over a wire. In the wireless case not only noise has to be considered but also fading effects, i.e., temporary fluctuations in the channel attenuation [TV05]. The packet loss probability can be reduced at the cost of increased overhead by reducing the packet sizes. For example, the low-power personal area network standard IEEE 802.15.4 specifies a maximum payload length of 115 bytes of a single packet [IEE03]. The actual usage of the channel is 133 bytes ($\approx 16\,\%$ overhead) including additional header and footer information. In contrast, traditional wireless LAN (unencrypted 802.11) supports payloads of up to 2304 bytes [IEE97]. The overhead per frame in 802.11 is 58 bytes including header and preamble bits ($\approx 2.5\,\%$ overhead).

Sensor nodes are typically battery powered. In some cases they also use energy scavenging techniques. Most common is photovoltaic energy conversion through solar cells. Mechanical energy has also been used (vibrations and piezoelectric converters [RWR03]). In order to mitigate the difficulties of intermittent power generation the electrical energy is stored in secondary batteries or high capacity capacitors on the nodes. Wireless nodes have to be designed and—equally important—used very energy efficiently. This is particularly relevant for systems whose energy supply is based on primary batteries since replacing drained batteries is often difficult, e.g., in Alpine deployments [THGT07], or at all impossible.[1] In general, the lifetime of a deployment is given by the battery lifetime of its nodes. Therefore, all components of the notes have to be duty-cycled, i.e., turned off when they are not used in order to save energy. We provide a detailed analysis of the power breakdown for one hardware platform in Chapter 4.

The motes shown in Table 2.1 contain different sensors capturing various physical phenomena. Together with the built-in sensors the nodes can be considered as prototype systems that can be used in research. The quality of the sensors is in general not sufficient for scientific measurements. For example, the temperature sensors do not provide the necessary resolution and since they are mounted on the circuit boards the thermal coupling to the measurement environment is difficult. Nevertheless, the nodes can be used to acquire real measurements for prototype applications is system research focusing on routing and data processing. Specific

[1]The Swiss Avalanche Research Institute operates a deployment of acoustic sensors (geophones) at the Wannengrat near Davos Switzerland. The sensors are covered by several meters of snow during the winter season. Digging out the sensors for a battery replacement would irreversibly damage the snow pack and destroy the measurement setup.

deployments such as the permafrost measurements in the Swiss Alps [THGT07] require custom designs and can only make limited use of off-the-shelf components. Experiments performed during this dissertation are done in a controlled lab setup using commodity nodes.

There is a debate about the size and capabilities of a sensor node. The notes presented here differ significantly in size. While the Mica2 and Tmote Sky nodes only provide a few kilobytes of memory the Imote2 is equivalent to a PDA and can run a general purpose operating system. Following Moore's Law technology can follow two different paths. Either sensors nodes with severely limited capabilities such as Mica2 shrink in footprint size, or alternatively, more powerful processors and radios become available with the same footprint. Currently limiting technology parameters such as RAM size may disappear in the future allowing for additional abstractions that simplify the engineering effort and ultimately make general purpose computing possible on sensor nodes. Another possibility is to combine different node types in the same network provided that their radio is compatible, for example, using TMote Sky and Imote2 nodes. An approach explored in Tenet [GJP+06] is to build a hierarchical multi-tiered networks where each of the more powerful nodes, e.g., the Imote2 nodes, controls a small number of smaller and less expensive nodes, e.g., Tmote Sky nodes. Future sensor networks may also consist of cell phones equipped with sensors. Devices such as the iPhone currently are already viable candidates for a sensing platform. It was already used in the SoundSense system [LPL+09]. Thiagarajan et al. use mobile phones in cars to track trajectories in the *VTrack* system [TRL+09]. Although the computing platform is more powerful than the mote-scale devices some important problems remain such as noisy sensor data and energy consumption. We focused on mote-scale hardware that was available in 2006. Abstracting from absolute performance numbers, we believe that the conclusions drawn in this dissertation are independent of the parameters of the actual technology used.

2.1.2 Programming

Programming wireless sensor networks is equivalent to programming networked embedded systems. Developing embedded systems is a nontrivial task, in particular, in case of wireless sensor networks that provide several constraints (memory, energy consumption, unreliable radio communication). Due to the resource constraints specialized operating systems and programming models have been used. Initially, event-based systems were used (TinyOS [HSW+00], Contiki [DGV04]). Multi-threading was very limited (Nut/OS + BTnut [Beu06]) or believed impossible at all given the amount of RAM available. Han et al. [BCD+05] showed that is still possible to implement and multi-threading using only a limited amount of thread state. Applications were originally written in C language and required a significant amount low-level programming. TinyOS uses a specialized version of C, called *nesC* (Network Embedded Systems C) [GLvB$^+$03]. It extends the C language by component model and provides constructs for the event-driven programming model of TinyOS. Throughout this dissertation work TinyOS/nesC was used for network implementations.

Compiled application programs are linked against operating system libraries to system images that are downloaded onto the sensor nodes, i.e., the binary images are written to the program flash. Downloading the application onto the nodes typically happens before deployment as it requires physical access to the device. Reprogramming the nodes after the deployment hence is difficult. However, in order to still be able to fix software bugs after the initial deployment, the ability to perform in-network reprogramming is important in any practical setting. Hui and Culler propose *Deluge* [HC04] an in-network programming solution for mote-scale devices such as the Mica2 and the Tmote Sky platform. New program images can be injected and propagated through the network. Despite its usefulness in-network reprogramming introduces two problems. First, microcontrollers for embedded systems are radically different than general purpose CPUs. For example, the Atmel ATmega128L controller is based on a Harvard memory architecture, i.e., it uses two different memory spaces for instructions and data. Hence, programs that are received over the air and thus are stored in the data memory cannot be directly executed by the CPU. The program image that is received in fragments eventually needs to be written into the node's flash memory. This has to be done in a coordinated fashion. The second problem are the high energy costs for the program dissemination in the network.

The following simple calculation illustrates the energy cost. Hui and Culler report [HC04] that Deluge is able to propagate 90 bytes/sec in the network (using the CC1000 radio). A first observation is that this only corresponds to $\approx 4\,\%$ of the gross bandwidth of the CC1000 radio. Replacing the entire 128 kB flash image of a single Mica2 node requires 24 minutes (in ideal situations when no retransmissions due to packet loss are necessary). According to the datasheet, the CC1000 transceiver consumes 76 mW during transmission and 29 mW while receiving. A simplifying assumption is that each node receive the entire program and will also resend it to its neighbors. This results in an ideal energy cost of 150 J for the entire program image. The cost for the CPU and writing to flash memory is not included. The Mica2 node is powered by two AA batteries. Consider a standard industrial alkaline Leclanché battery that provides 2.65 Ah at an average discharge voltage of 1.2 V. This results in a total energy capacity of a Mica2 node of 23 KJ. [2] Therefore, exchanging a full program image uses at least 0.7 % of the

[2]For comparison, McDonalds reports a food energy of 2071 KJ for a Big Mac® hamburger.

total energy. In order to put this number into perspective assume that the sole purpose of a network is to be reprogrammed once every day. Using the energy consumption of the back-of-the-envelope calculation the life time of the network is at most 5 month. Typically we observe a higher energy consumption caused by retransmissions due to transmission failures.

The programming costs can be reduced by either sending deltas or using virtual machine-based approaches that typically result in more concise program descriptions. Panta and Bagchi propose *Hermes* [PB09] that uses an version of the Rsync algorithm on two different code images and computes the delta between the version. One important technique they use is indirection tables for function calls. Virtual machines for WSNs execute high level byte code can be replaced with less transmissions than binary images. Levis and Culler in *Maté* [LC02] and in *SwissQM*, which is a key contribution of this dissertation.

2.2 Architectures of Wireless Sensor Networks

The specific wireless technologies used in embedded nodes give rise to distinct network architectures. In contrast to traditional cellular and wireless local area networks, such as the IEEE 802.11 protocols, WSNs typically are not infrastructure based, that is, the do not rely on a central access point that arbitrates access and communication over the shared medium. Sensor networks built in such a way are called infrastructure-less or *Ad-hoc Networks*. Although IEEE 802.11 also defines an ad-hoc operation mode its not used for mote-scale devices because traditional 802.11 requires too much power and requires a complex protocol stack. Radio technologies for WSNs are specifically designed to fit into the resource constrained environment. There is a noteworthy exception; the ETH BTnote platform [Beu06] by Beutel et al. that uses Bluetooth technology. However, the designers have noted that it is difficult to build larger networks based on Bluetooth alone [BDMT05]. In fact, the BTnode platform provides two different radio solutions. Next to the Bluetooth radio the nodes contain a CC1000 chip that is also used in the Mica2 nodes.

2.2.1 Wireless Communication Links

Radio links used in sensor and ad-hoc networks have very different characteristics than wire-based communication. The most important property is the higher bit error rate due to the significantly lower *signal-to-noise ratio* (SNR) in radio links. Even worse, interference caused by other transmitters in the vicinity of the receiver aggravate the reliability of the wireless link. Interference contributes to the noise at

This corresponds to energy equivalent of 90 AA batteries.

the receiver. Therefore, for multiple-access systems the *signal-to-interference-plus-noise-ratio* (SINR) that explicitly considers interference. SNR is typically used for point-to-point links. The lower SNR (or SINR) translates into a higher bit error rate. Packet-based communication uses checksums to detect data corruption. Data packets that are received with an incorrect checksum are discarded. Hence, a low SNR leads also to a high packet loss at the receiver. Higher-level measures such as retransmission, forward error correction, and erasure codes mitigate loss at the price of latency and complexity. The absolute bit error values, however, depend on the design of the physical layer, e.g., the modulation scheme used, the sensitivity of the detector at the receiver, and the transmitter power. Environment effects such as the length of the signal path, hence, the signal power at the receiver, as well as channel noise, and interference further affect the bit error rate. Unfortunately, the bit error rate is not directly correlated with the communication distance in practice. This renders the deployment of a sensor network a nontrivial task.

In theoretical work on routing in sensor networks simplified radio models such as the *Unit Disk Propagation Model* are chosen. The model assumes that a transmission can be successfully decoded within a circle round the transmitter. The transmission cannot be heard outside the circle. Experiments show that such models are incorrect. The first reason is that the communication range is not a circle. Scattering and interference yield more complex coverage areas. Second, the probability of a successful is not uniform inside the circle and there is no sharp boundary at the edge of the disk. Third, transmitters can cause interference even if they are out of the communication range. Halkes and Langendoen provide practical guidelines [HL10] for theoretical work. Under these considerations this dissertation focuses on experimental verification using real deployments instead of relying on simulation.

2.2.2 Link Quality Measurements

In order to quantify the link quality that can be found a real deployment an experiment was conducted under laboratory condition at ETH Zurich. 33 Tmote Sky nodes were placed in an large room. The network was deployed in an old chemistry lab room (CAB F31) at ETH Zurich that is under protection order and currently serves as a museum. The nodes were placed on the lab tables. The 33 nodes are deployed on an area of $8 \times 16\,\mathrm{m}^2$ over three years. The floor plan is shown Figure 2.1. Initially 36 nodes were deployed. However, during the experiments three nodes (10, 17, and 32) had to be removed due to hardware failure. The Tmote Sky nodes provide a USB port that can be used for downloading the application image and for communication purposes, e.g., logging. All nodes of the deployment are connected to a desktop computer through a USB tree. This permits programming and logging of each node individually in a reliable manner. The deployment is used for multiple experiments for this dissertation.

The goal of the first experiment is to measure the *Packet Loss Rate* in the deployment using the standard TinyOS networking stack with a direct node-to-node communication. The experiment was designed with the following requirements that should reflect a traffic pattern of a real application.

- The communication pattern is random, i.e., each node randomly selects a destination node it sends a message to.
- The nodes send messages simultaneously and independently. This intentionally leads to interference.
- In order to obtain a maximum coverage area the nodes' transmitters are set to the highest output power of 0 dBm (1 mW).

The experiment is performed in rounds. At the beginning of each round every node i generates a random permutation of the other nodes' IDs. It will then process this list sequentially and sends PING message to next destination node j in the list. When a node j successfully received a message from a node i it records this fact in its sender vector $\mathbf{s}_j$ by setting entry $(\mathbf{s}_j)_i = 1$. Each round begins with a zero sender vector. The beginning of a round is started by dedicated beacon node (node 0 in Figure 2.1). At the end of each round the nodes send their $\mathbf{s}_j$ the data logger computer that is connected via USB.

The experiment uses all 33 nodes of the deployment. $N = 32$ nodes participates exchange PING messages. The size of a PING message is 15 bytes. Node 0 is only used to signal the start of a round. It is placed in the center of the room and is operating at full radio power. Further more, at the beginning no node is transmitting before the round starts, hence, the start signal can be received without interference caused by other nodes. During each round, every node sends a PING message to the 31 other nodes. It does not send a message itself. At the end of the round the N sender vectors are $\mathbf{s}_1, \ldots, \mathbf{s}_N$ are combined into an $N \times N$ matrix

$$\mathbf{S} = \left[\begin{array}{cccc} \mathbf{s}_1 & \mathbf{s}_2 & \cdots & \mathbf{s}_N \end{array} \right] .$$

This matrix can be regarded as the *Adjacency Matrix* of the communication graph for that particular round. By averaging the sender matrix over multiple rounds we can estimate the link quality as the probability of a successful transmission between two nodes. We define the *Link Quality Matrix* $\mathbf{Q}$ as the average of $\mathbf{S}$ over R rounds

$$\mathbf{Q} := \frac{1}{R} \sum_{k=1}^{R} \mathbf{S}_k \ .$$

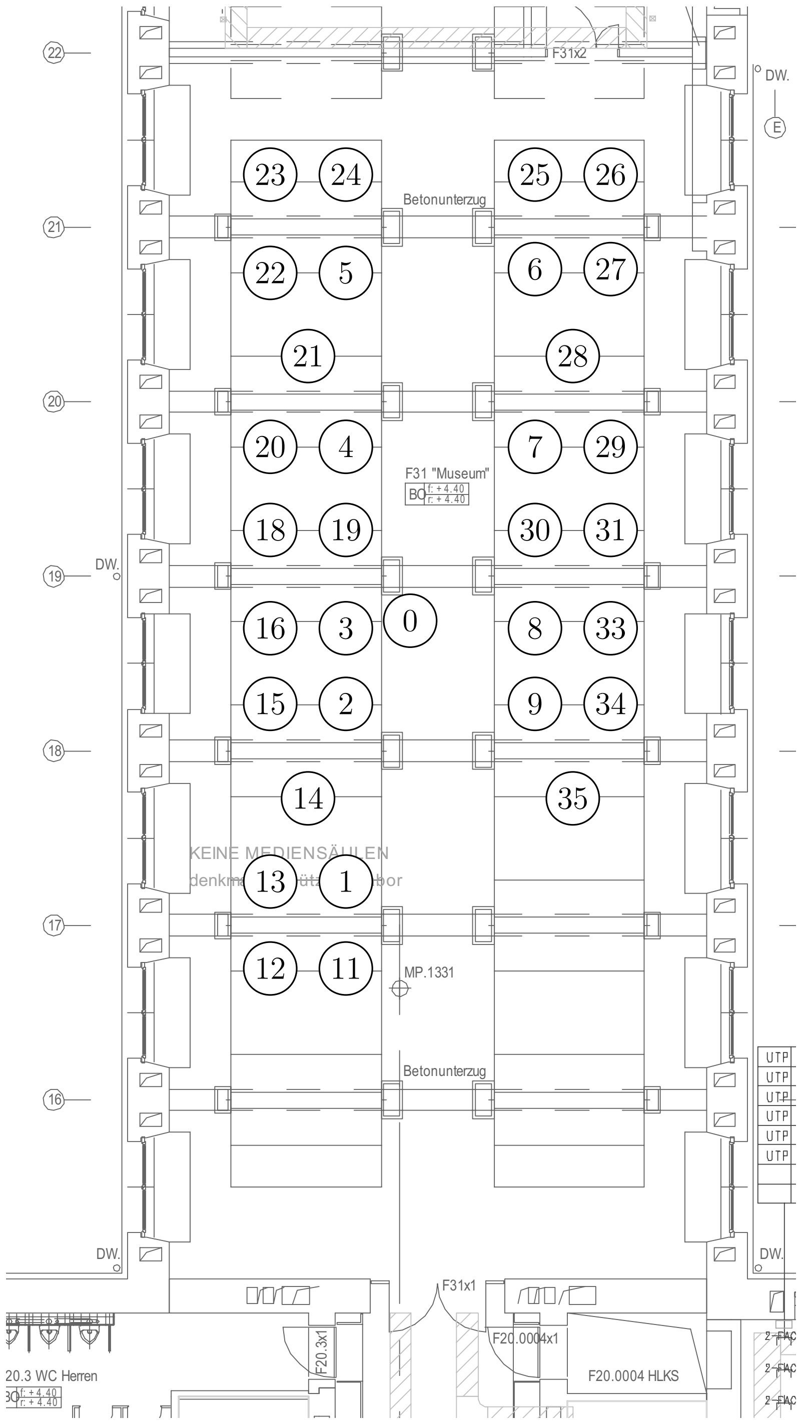


Figure 2.1: Sensor network testbed consisting of 33 Tmote Sky nodes. The nodes are deployed in an old chemistry lab (CAB F31) at ETH Zurich.

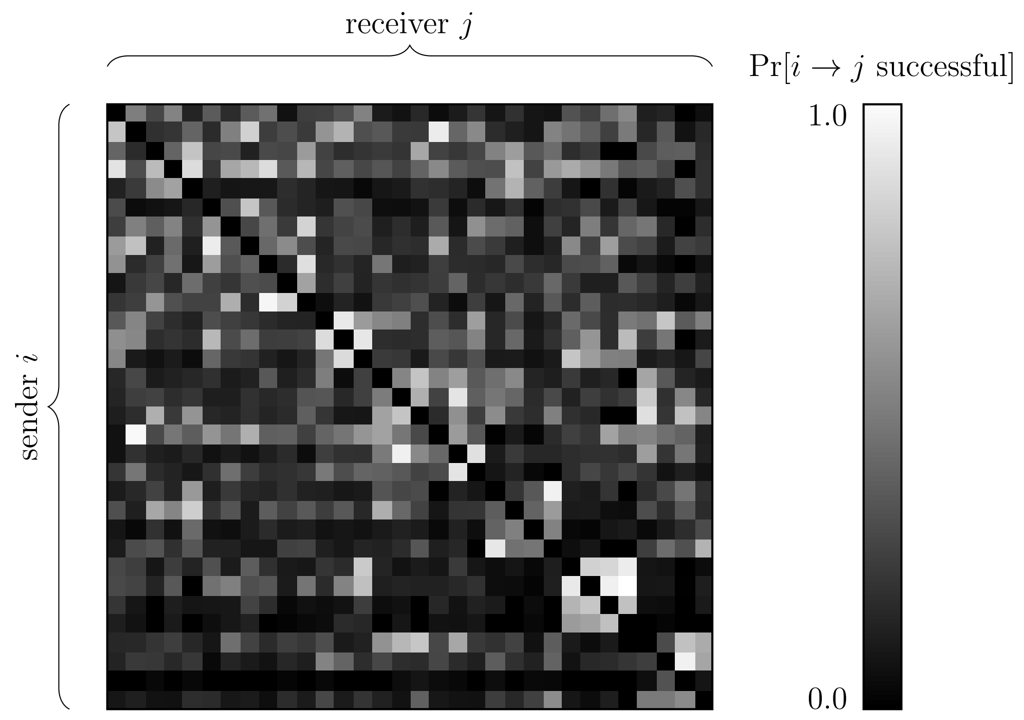


Figure 2.2: Adjacency matrix of link reliability indicates the estimated probability a successful transmission between any two nodes in the testbed

Q is used for illustration of the communication properties in a wireless sensor network. A graphical representation of the link quality matrix is shown in Figure 2.2 it was computed from $R = 170$ rounds. Of the $32 \times 31 = 992$ messages that were sent in each round, 197.6 were received on average. This corresponds to an success probability of 0.2 in the network or equivalently to a packet loss rate of 80 %.

An additional observation is that the matrix is not symmetric, as can be seen from the following metric

$$\frac{\|\mathbf{Q} - \mathbf{Q}^T\|_2}{\|\mathbf{Q}\|_2} \approx 3 \ ,$$

that measures the deviation from a symmetric matrix. Asymmetry in the link quality matrix corresponds to asymmetric communication links. In other words, a good communication link from node i to node j does not necessarily imply an equally good reverse channel from j to back i. One reason for asymmetrical links in practice, are different SINR at the receivers, e.g., different levels of interference. Figure 2.2 also shows that there are “bad senders” (node 34), i.e., nodes whose message are not received very well by the others. A more detailed analysis of these

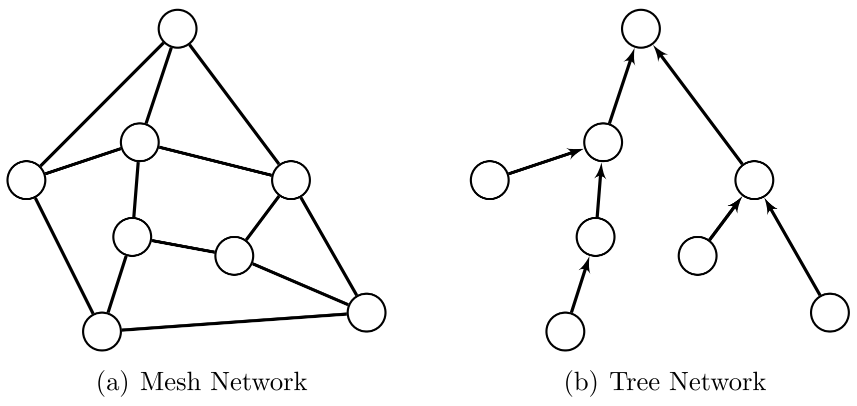


Figure 2.3: Different network topologies

nodes identifies a hardware issue as the most likely cause. Some Tmote Sky nodes seem to have a mismatched antenna and hence have a limited transmission range. In summary, we can conclude that although this experiment uses an artificial communication pattern wireless sensor networks are very unreliable. The experiment illustrates the properties and difficulties of wireless communication.

2.2.3 Network Architectures

Two important factors contributed to the significant packet loss. First, all nodes where operating at the maximum transmission power setting. Second, all nodes where transmitting simultaneously after receiving the beacon message that starts each round. This causes, significant contention on the shared radio channel. The solution is a “desynchronization” of the transmissions and a reduction of the transmitter power. Multi-hop communication is then needed to make sure that nodes in the network remain connected.

Mobile ad-hoc networks (MANETs) use a mesh network structure as shown in Figure 2.3(a). This multi-hop routing structure allows each node to send messages to any other node in the network. The network is thus fully connected. Establishing routing links in such networks is a complex task. Many different routing protocols have been suggested in by the research community such as AODV [PBRD03], a reactive routing protocol, or OLSR [CJ03], a proactive protocol. Mesh networks can support any point-to-point communication pattern. The flexibility of the communication pattern, however, comes at a cost. The routing tables have to be maintained and exchanged among nodes, which adds complexity to the system.

Sensor networks operating on more constrained devices than the 802.11-based MANETs usually exhibit simpler communication pattern. Sensor data is generated

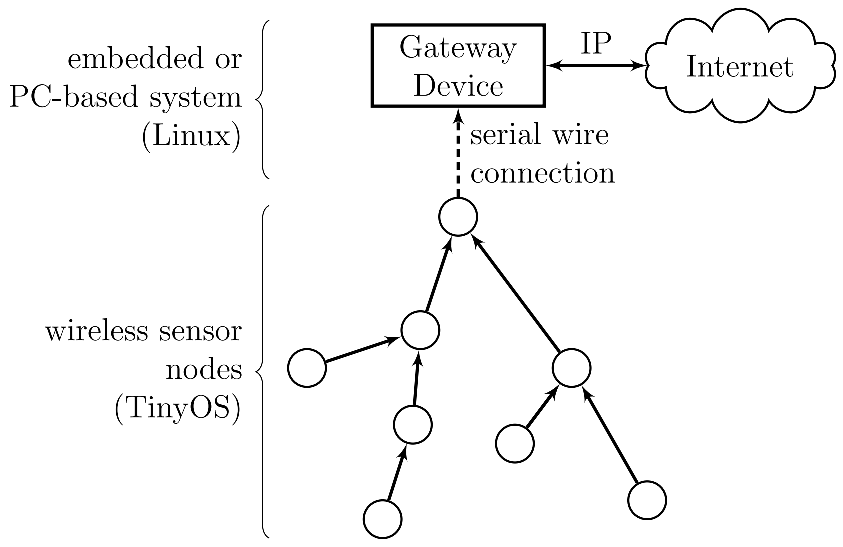

embedded or PC-based system (Linux)
Gateway Device
IP
Internet
serial wire connection
wireless sensor nodes (TinyOS)

Figure 2.4: System Architecture of tree-based sensor network with gateway node at the root of the collection tree. The gateway node connects the sensor network to the Internet.

spatially distributed over the network. The sensor data is then collected at a central location in the network. Hence, rather than a dedicated one-to-one routing in MANETs the traffic pattern here is many-to-one. This can be be implemented using fewer links and less state per node using a *spanning tree* over the network as shown in Figure 2.3(b). In a tree routing network the routing decision is fairly simple. A node can only send a message to one single node, its parent. The root of the tree is located where the data is recorded. Due to the specific role of the root node it also called *Base Station* or *Gateway.* As such, it provides access to the sensor network, e.g., from the Internet. In practice, the gateway node is either a more powerful sensor node or an embedded or standard computer system that is connected to a sensor node over a fixed wire connection. Figure 2.4 depicts the system architecture used in this dissertation. In the laboratory testbed shown in Figure 2.1 the role of the gateway is carried out by a Pentium-III server running Linux. The gateway is also connected to the USB-based backbone network that provides a programming and logging infrastructure to each node.

2.2.4 Collection Tree Routing

Data collection tasks in sensor network lead to many-to-one communication that is implemented through tree routing. An initial protocol *MintRounte* was introduced by Woo et al. [WTC03] for TinyOS 1. Gnawali et al. developed the *Collection*

Tree Protocol (CTP) [GFJ+09], an improved protocol for TinyOS 2 that can better handle higher link dynamics in the 2.4 GHz spectrum. Both protocols dynamically maintain a routing tree that is rooted at the gateway node. A sophisticated link quality estimator is used such that each node can select the currently best parent node it sends messages through towards the root. The application implementations that were completed in the context of this dissertation used CTP available in the TinyOS 2 distribution or MintRoute available in TinyOS 1.

Like the link quality measurements in Section 2.2.2, the behavior of the tree collection routing protocol is analyzed on the ETH testbed (Figure 2.1). The goal of the experiment is to determine the reliability of the CTP-based routing and compare it with the direct communication. A TinyOS 2 application is implemented that builds on top of CTP. In this application each node periodically reports its current parent tree. Such a report tuple has the following schema

$$(nodeid, seqnr, parentid)$$

nodeid and *parentid* contain 16-bit node addresses. A per node sequence number *seqnr* enumerates the report message. Report messages are forwarded along the tree. Each report tuple is sent as a separate message. Different report tuples are not merged into one single message. Including the MAC header and the CTP routing information the 16-byte report tuple is sent as a 27-byte network message.

Experiment Setup. The experiment is repeated on the 33-node testbed for different reporting intervals. The reporting interval determines the traffic volume in the network. The range of the reporting interval is chosen between 100 ms and 10 s. The high packet loss in link quality measurements in Section 2.2.2 was explained by the fact that all nodes were transmitting at the same time. Thus, in a more realistic setup the reporting instants of the individual nodes where randomly distributed inside the reporting interval. In this setup the radio receivers where always active (idle listening). The random distribution of the transmission instants represents and ideal setting because contention in the network is minimized. As a second parameter the influence of the transmitter power is analyzed. The power is varied in the full scale supported by the CC2420 transmitter between -25 dBm (3 μW) and 0 dBm (1 mW). The experiment is repeated for each power and report interval setting.

The experiment determines the *Coverage* in the network, which is defined as the ratio of the number of report tuples received at the base station and the total number sent. In the sensor network literature the term *Yield* is also used. Coverage, thus, can be considered as the average probability of a successful transmission from a node along the tree to the base station. By modeling this process as a Bernoulli trial we can determine the number of tuples to send, hence, the

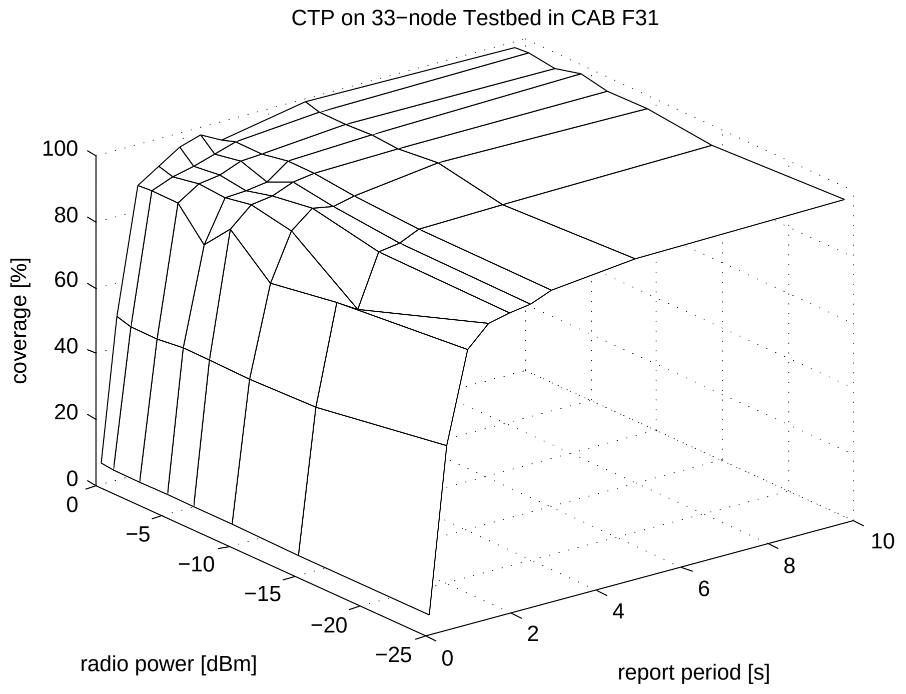
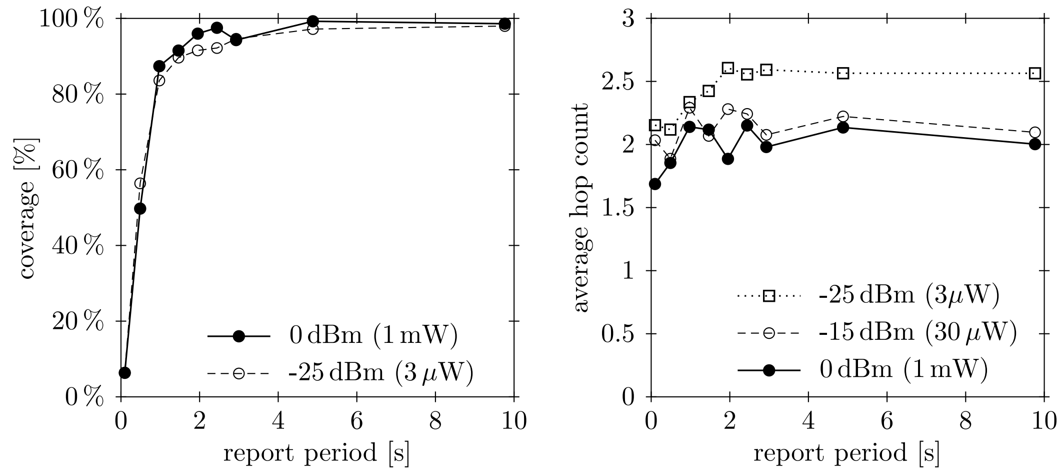


Figure 2.5: Average coverage using Collection Tree Protocol for different data rates and transmission power setting measured in CAB F31 testbed

duration of each experiment, in order to obtain a given confidence level. Using the Clopper-Pearson [CP34] method we can determine that if at least 400 reports per node and experiment are collected the error in the estimated coverage value is ± 0.1 at confidence level 95 %. The experiment runs are completely independent. Before each experiment all network nodes are reset. After the reset a delay of two minutes allows CTP to build up the routing tree before the message exchange starts.

Figure 2.5 depicts the coverage for different reporting periods and power settings. The figure shows that reporting intervals greater 2 s the coverage is $> 90\,\%$. This is a significant improvement over the average 20 % success rate of a one-hop link from previous experiment in Section 2.2.2. The figure also indicates that coverage does not strongly depend on the power setting. The 2D plot in Figure 2.6(a) shows the coverage for the minimum and maximum power setting. Note that even though the power difference is 25 dB there is no significant difference in the coverage curves. The coverage, however, drops for intervals < 1 s. In this range the network is saturated and packet loss increases because of congestion. Figure 2.6(b) shows the average hop count of the messages for different intervals and power set-



(a) Coverage (b) Average Hop Count

Figure 2.6: Average coverage and hop count using Collection Tree Protocol for different data rates and power settings

tings. The graphs confirm the expected behavior, that the number hops increase as the transmitter power is reduced. The “noise” in the hop count for short intervals $< 1\,\mathrm{s}$ is again an effect of contention. The packet loss increases, hence, the quality of links estimator decreases. CTP uses the *ETX link quality estimate* introduced by De Couto et al. [CABM03]. The value captures the expected number of transmissions, i.e., $^1/_{1-p}$ where p is packet loss probability. The ETX values along multi-hop paths to the root increase due to contention. This in turn causes the nodes to most likely pick a direct link to the root node and the average hop count decreases.

Comparing the single hop experiment from Section 2.2.2 with the CTP routing one can see one reason why multi-hop routing in wireless sensor networks is used even when a direct link to the data sink is possible. The packet loss is reduced. Additional techniques such as acknowledgment messages and retransmissions, and adding redundancy either by sending a message over multiple links or using coding techniques can further reduce the data loss. CTP can use acknowledgment frames from IEEE 802.15.4 MAC. However, in our work we experienced various issues with the implementation of acknowledgment mechanism in TinyOS. In fact, they are disabled by default in the TinyOS messaging. We did not enable acknowledgments because the coverage for reasonable large tuple intervals is good enough without.

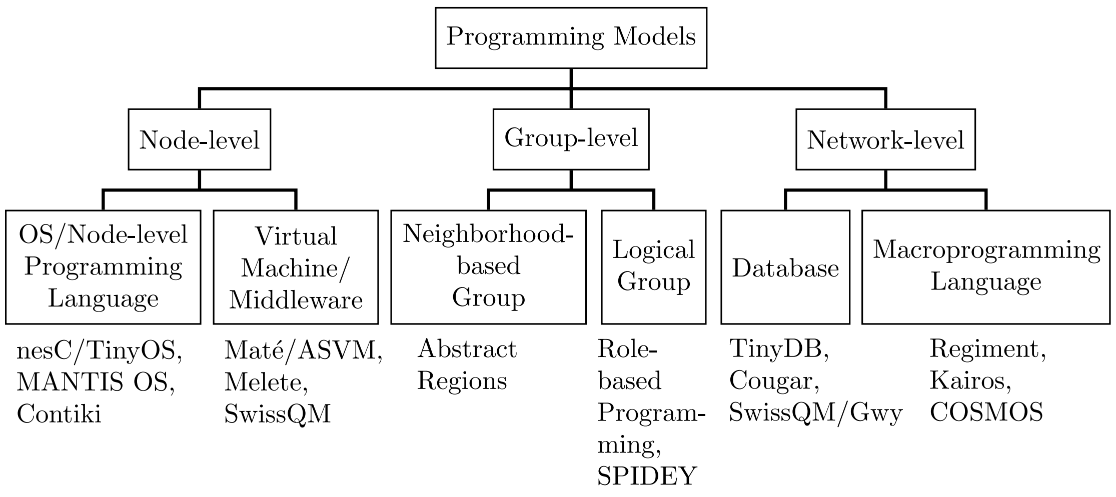

Programming Models
Node-level
Group-level
Network-level
OS/Node-level Programming Language
Virtual Machine/ Middleware
Neighborhood-based Group
Logical Group
Database
Macroprogramming Language
nesC/TinyOS, MANTIS OS, Contiki
Maté/ASVM, Melete, SwissQM
Abstract Regions
Role-based Programming, SPIDEY
TinyDB, Cougar, SwissQM/Gwy
Regiment, Kairos, COSMOS

Figure 2.7: Taxonomy of programming models for sensor networks. Adapted from Sugihara and Gupta [SG08].

2.3 Data Processing in Sensor Networks

The processing capabilities of the embedded CPUs can be also be used for data processing, e.g., data fusion, cleaning of noisy sensor data, or even perform signal processing. The low-level programming described earlier in Section 2.1.2 is non-trivial and typically too difficult for the users interested in sensor networks for field science studies or high-level inventory management applications. Several research projects are addressing some of these limitations. In this section, we provide an overview of the relevant related work in the field. We first consider the programming and system space and then focus on work that is related to the two key ideas in the data processing platform developed during in this dissertation.

2.3.1 Programming Models

Sugihara and Gupta [SG08] introduce a classification of various programming models and systems used in wireless sensor networks. This taxonomy is shown in Figure 2.7 adapted with the SwissQM system. At a high level the programming models are classified by the abstraction level. *Node-level* programming refers to the lowest possible programming model. Here all nodes are programmed separately, i.e., the programs are executed directly by the nodes and, hence, have to written at node granularity. Communication among nodes is explicit. *Group-level* programming no longer considers individual nodes. Programs are written for a *set of nodes*. Communication is explicit only between node sets. At the highest level of the abstraction the network is programmed as a single entity (*Network-level*).

There is no notion of individual nodes or explicitly addressed groups.

Node-level Programming. At the lowest level sensor networks can be programmed to run on bare hardware or limited sensor network operating systems such as TinyOS [HSW+00], Contiki [DGV04], or MANTIS OS [BCD+05]. The programs are written in low-level C or in case of TinyOS in nesC [GLvB+03], a specialized version of C. Virtual Machines (VMs) on top of sensor network operating systems provide an additional abstraction layer. Application programs are written in bytecode and distributed in the network. The advantage of this approach is that due to the high-level interface and a powerful instruction set complex programs can be written with few instructions.

Group-level Programming. Approaches that allow implementing programs using group concepts can be categorized depending on how the groups are formed. *Neighborhood-based groups* are defined on the physical closeness while logical groups are formed using predicates on node properties. *Abstract Regions* by Welsh and Mainland [WM04] provides a TinyOS API with group primitives. The API allows applications to discover when neighbors join or leave the group, to enumerate the group members, to share data in the group in key-value store, and provides a reduction operation that combines the values of a shared variable to a common result (e.g., *sum*, *max*, etc.)

Römer et al. [RFMB04] propose a *role-based programming model* where each node will take specific role (e.g., leaf or cluster-head) and performs a computation accordingly. A program consists of a set of a common rules that are evaluated by every node. The rules allow clustering depending on dynamic node properties such as sensor reading. Clustering rules can refer to properties within a neighborhood and lead to a localized communication pattern similar to Abstract Regions. Mottola and Picco [MP06] propose *SPIDEY*, which also introduces node attributes that can be statically assigned at compile time as well as dynamic attributes on sensor readings. The logical group is defined by a predicate on the node attributes. Primitives for in-group communication are also available in SPIDEY.

Network-level Programming. Programs written at the network-level treat the entire network as single abstract entity. A translation system is available that converts the global specification into node-specific instructions. A common approach used in data-centric applications is to consider the network as a *Database*. Network programs are queries written in a declarative query language, e.g., a SQL-like dialect in TinyDB [MFHH05]. For more general applications, where the expressiveness of the query language—for example SQL itself is not Turing-complete—is insufficient a different approach called *Macroprogramming* can be used. The *Regi-*

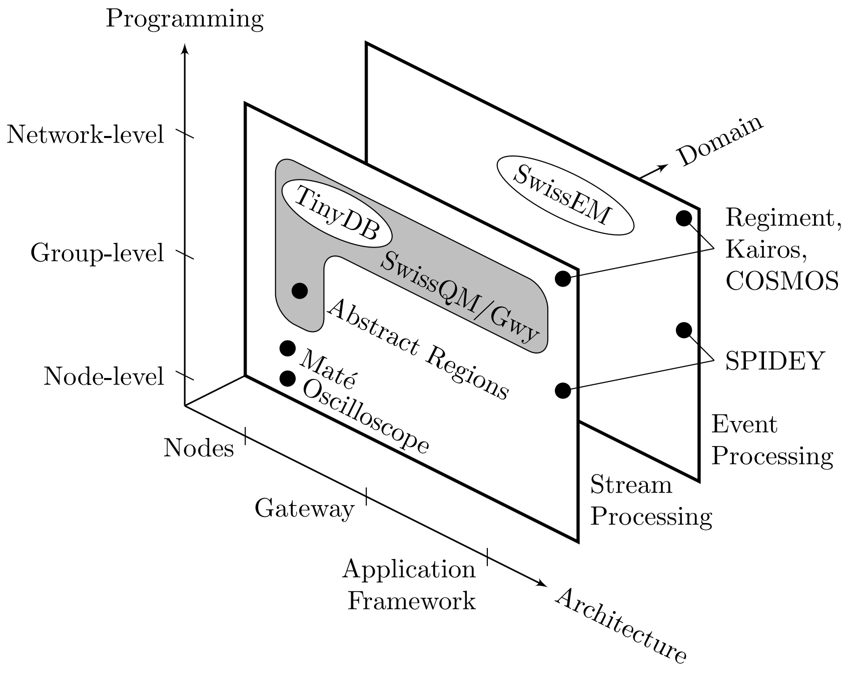


Figure 2.8: System space for wireless sensor networks

ment system by Newton and Welsh [NMW07] uses a functional language similar to Haskell. The use of a functional language also allows hiding state in the programs from the programmers and let the system decide how to parallelize computation and how to assign state to nodes. *Kairos* (Gummadi et al. [GGG05]) is a programming language extension that makes remote data access possible. Kairos allows to access a variable on a remote node using the syntax `variable@node`. A different approach is used in *COSMOS* (Awan et al. [AAJG07]). A macro program is expressed as a dataflow abstraction. The functional components in the data flow graph are then dynamically mapped onto sensor nodes. COSMOS can also be used in heterogeneous networks. The same idea is also used in *WaveScope* (Lewis et al. [GMN$^+$07]).

2.3.2 System Space

An overview of the sensor network system space is shown in Figure 2.8. The system are classified by the programming level, the system architecture level and the application domain. At a very high abstraction level WSN applications are categorized into the stream processing or the event processing domain (shown as planes in Figure 2.8). We consider any application that continuously produces data to belong into the stream processing domain. In the event processing scenario nodes only exchange data if some event happens (e.g., a fire is detected and an alert message is sent). The SwissQM system covers the stream processing domain. The *SwissEM* is an Event Machine that was derived from the SwissQM system was implemented by Li [Li08]. It executes finite automata for event detection problems as bytecode programs in the network.

The position along the architecture axis indicates at which level in system stack the solution is located. Consider for example the *Oscilloscope* application which is an example application that ships with the TinyOS source distribution. Oscilloscope periodically samples a set of sensors and forwards the samples along a collection tree to the basestation using CTP. Oscilloscope itself is a low-level program written in nesC and is executed by all nodes. It is therefore located in the lower left corner of the stream processing plane. Regiment, Kairos and the COSMOS macroprogramming system can be used for both stream and event processing, hence, they are present on both planes. The SwissQM virtual machine itself works at the node-level (and including dedicated instructions such as `merge`) also at the group-level. Together with the SwissQM/Gateway it builds a declarative interface for the sensor network, similar to TinyDB. Since the SwissQM gateway system is built out of modular components and uses a service architecture, it is extensible and also considered as an application framework. Therefore, the SwissQM VM including the gateway system covers the shaded space in the stream processing domain in Figure 2.8. In the next sections we describe the related work of the VM component and the gateway system that provides the declarative query interface.

2.3.3 Virtual Machine-based Approaches

Virtual Machines (VMs) on top of sensor network operating systems provide an additional abstraction layer. The *Maté* VM by Levis and Culler [LC02] is a tiny virtual machine for nodes running TinyOS. The advantage of this approach is that due to the high-level interface and a powerful instruction set complex programs can be written with few instructions. Hence, the programs that are disseminated in the network are relatively short compared to the binary native images. Reprogramming a sensor network by exchanging small bytecode programs is more efficient than replacing entire binary system images using Deluge [HC04]. Maté is a stack-based virtual machine similar to the *Java Virtual Machine* [LY98]. Maté bytecode programs split into a number of *code capsules*. Code capsules are bound to events. They are invoked whenever the corresponding event occurs. For example, if a timer fires or a data packet was received or sent. In other words, the Maté VM provides an event-based execution model that nicely matches the event-based TinyOS operating operating system. A single code capsule fits into one single TinyOS message. This allows atomic code updates and thus avoids buffering partial code updates and the use of a consistency protocol. However, the code size, i.e., the length of a code capsule is also limited (up to 24 instructions). Code capsules are explicitly forwarded by programs by invoking a send instruction. The footprint size of the bytecode interpreter is 7 kB in code size and 600 B in data. Levis released Maté in the TinyOS 1 source distribution as *Bombilla VM*.

Levis et al. later extended the VM approach to *Application Specific Virtual Machines* (ASVMs) [LGC05]. In an ASVM the bytecode instruction set can be extended by application-specific instructions. This allows a flexible boundary between virtual application code and the interpreter engine. The developer decides at deployment time which functionality to add to the instruction set. Similar to traditional programming where commonly used functionality is implemented as subroutines that are invoked when needed. These additional instructions can be invoked like macros from the bytecode. The SwissQM virtual machine draws from this idea too as it can be regarded as application-specific machine if the in-network aggregation operations are considered as application-specific extensions. However, SwissQM uses a specific VM architecture that facilitates stream processing and an atomic code distribution mechanism for the entire program unlike individual code capsules in Maté.

The *Melete* virtual machine for sensor networks by Yu et al. [YRBL06] follows a similar approach as used Maté but allows multiple concurrently executing bytecode programs. It also uses a passive program forwarding approach similar to Levis' ASVM. Yu et al. present an implementation that provides space for up to five concurrent programs that can run on a Tmote Sky node [YRBL06]. SwissQM also permits concurrent execution multiple bytecodes. It also allows to trade-off the amount of per-program state with the number of concurrently executing programs.

2.3.4 Declarative Interfaces

Most relevant to the work of this dissertation are systems that provide a declarative interface to the user. The idea of querying the physical world using a *device database* was introduced by Bonnet et al. [BGS00, BGS01]. A prototype of query engine running on a sensor nodes was shown by Fung et al. [FSG02]. The *COUGAR* project [YG02] started at around 2000 at Cornell focuses on tasking sensor networks through a declarative query interface. It concentrates query optimization aspects. A *Query Proxy* [YG03] running on the sensor nodes executes streaming queries according to the template:

```
   SELECT  {attributes, aggregates}
     FROM  {sensordata S}
    WHERE  {predicate}
 GROUP BY  {attributes}
   HAVING  {predicate}
    EVERY  time span e
```

A network query is executed once every e time span. The queries can also contain aggregate expressions such as *avg*, *count*, *max*. These operands spatially aggregate data, i.e., fuse data from different sensor nodes. Spatial aggregation can be efficiently computed in-network. Common aggregates (distributive aggregates with a constant amount of state) can typically be evaluated more efficiently in-network than at the base station. An efficient method called *Tiny AGgregation* (TAG) was presented by Madden et al. [MFHH02]. In common cases, it reduces the number of messages that have to be sent across the routing tree to one message per tree edge. The COUGAR project has not publicly released any system implementation. The first generally available implementation of a query processor for sensor network was *TinyDB* [MFHH05] developed by Madden et al. TinyDB is built on top of TinyOS and provides a declarative query interface (SQL dialect) to the sensor network as well as some capabilities for in-network data processing and aggregation. Buonadonna et al. [BGH$^+$05] combined TinyDB with a gateway appliance (embedded ARM-based Linux system running Java) into *TASK* (Tiny Application Sensor Kit), which has been used as a platform in may different field research projects. The SwissQM/Gateway system was designed to cover the same space. However, due to the modular design that is based on OSGi components and services SwissQM/Gateway can also be used as a research platform itself. For example, Doman et al. [DPD10] used the SwissQM platform and extended it to implement a fuzzy query processing model.

Levis et al. outline in their ASVM paper [LGC05] a method to implement an application-specific VM for query processing. The *QueryVM* is developed for TinyDB-like in-network data collection tasks. SQL queries are translated into *Motlle* (MOTe Language for Little Extensions) programs. Mottle is a Scheme-inspired script language. These programs are then executed by the QueryVM. Incomplete parts of the system are available in the TinyOS 1 tree. According to one of its authors, David Gay, "it fell by the way side" and was not completed. Even though the VM bytecode interface allows more flexibility for the queries it does make use of it. Like TinyDB it does not allow arbitrary expressions in the queries. SwissQM/Gateway is the first complete system that is publicly available that contains virtual machine for sensor networks and compiler that translates general queries into bytecode programs.

On overview of the network query processing of the COUGAR approach and TinyDB is given by their authors Gehrke and Madden in [GM04]. In general, a declarative interface makes it easier for inexperienced users to use sensor networks for data acquisition tasks. Writing a high-level database query is expected to be easier than developing a low-level software for networked embedded systems. This approach, however, also has a number of disadvantages. First, not every sensor network application can be expressed as a query. While this approach works well for data stream tasks, applications that require collaborative event-processing cannot be implemented as queries. Second, the expressiveness of SQL is limited, as it is by itself not Turing-complete. Third, due to resource constraints on the sensor nodes the functionality of the query processor is severely limited, for example, TinyDB does not support queries with user-defined functions (UDFs). The SwissQM system addresses these issues. Instead of a query processor a virtual machine is deployed on the sensor node and queries are compiled in to bytecode sequences at the SwissQM/Gateway. This allows the integration of additional functionality such as UDFs while maintaining a declarative interface to the user through the gateway.

2.4 Summary

This chapter provided an introduction to wireless sensor networks. The most important observation made was that mote-scale sensor nodes are very resource constrained. This must be considered when developing query execution plans running on those nodes. First of all, the very limited amount of memory available implies that state must be carefully managed. For example, large data windows as used in aggregation queries cannot be kept on the sensor nodes.

Second, sensor networks are highly distributed systems. However, in contrast to Internet-based systems, the wireless ad-hoc networks are much less reliable. A data processing solution needs to account for comparatively low data rates and lost data due to the message corruption. Fortunately, the simpler communication pattern in such networks lessens the implications. Instead of a mesh network where every node exchanges data with every other node, in practice, data is forwarded along a multi-hop tree to a base station. Maintaining a multi-hop tree is less complex than a fully connected mesh network. Thus, any algorithm used for data processing in wireless sensor networks should adhere to this communication pattern and not rely on a mesh structure. More precisely, message exchange between any arbitrary two nodes should be prevented.

The chapter also discussed different programming modes for sensor networks. In general, developing software for sensor networks is non-trivial as it involves low-level programming and, due to the constraints, many cross-layer optimizations. For an end-user perspective the sensor network should be programmable at a higher level of abstraction. Instead of considering individual nodes, programs should be written at the global network level. Declarative interfaces through query-based approaches make the technology available to a broader range of users. This aspect also needs to be considered when building a data processing platform.

3 The SwissQM Approach

As part of this thesis a comprehensive platform was built that supports the entire data cycle in sensor networks: from data acquisition to data processing and storage, including deployment, optimization, routing, and embedding within end-user devices. This chapter describes how data processing is implemented on top of SwissQM (**S**calable **WI**rele**S** **S**ensor network **Q**uery **M**achine) [MAK07a,MAK07b]. In short, SwissQM is intended as the next generation architecture for data acquisition and data processing in sensor networks. Its main objectives are to provide richer and more flexible functionality at the sensor network level, a more powerful and adaptable interface to the outside world, data independence, query language independence, optimized performance in a wider range of settings than current systems, and smooth integration with the rest of the data processing architecture.

3.1 Query Platform for Sensor Networks

SwissQM is based on a specialized virtual machine that runs optimized bytecode rather than queries. As a result, SwissQM does not make any assumptions about the query language used (e.g., SQL or XQuery), about the deployment strategy of the underlying sensor network (e.g., one single network or multiple networks), and can easily provide highly efficient multi-user support. Compared to existing systems, SwissQM provides a *generic* high-level, declarative programming model (it can support both SQL and XQuery) and imposes no data model (e.g., relational or XML). We believe that the use of a declarative interface simplifies the specification of data acquisition tasks for end-users. Furthermore, optimization in a declarative system is much easier than in an imperative approach. SwissQM is also Turing-complete and supports user-defined functions, windowing queries, complex event generation at the sensor level, an extensible instruction set, sophisticated optimizations, sensor over-provisioning, and overlapping but distinct sensor networks. All these features make it possible to implement many and important optimizations in SwissQM that are either not possible or rather cumbersome to implement in existing systems.

3.1.1 Design Considerations

SwissQM has been designed with several requirements in mind:

1. *Separation of sensors and external interface:* SwissQM should not implement any particular query language. The programming model should be independent of the query language used. It must also be dynamically adaptable. As a result, the sensor nodes should not contain application-specific functionality (e.g., the ability to parse SQL or join operators). Such functionality is treated as a dynamically deployable extension.
2. *Dynamic, multi-user, multi-programming environment:* SwissQM should not impose restrictions on the query submission and change rate, nor in the number of queries that can be run concurrently (beyond the inherent limitations of the underlying hardware).
3. *Optimized use of the sensors:* The only processing at the sensor nodes should be that related to capturing, aggregating, and forwarding data. Anything else should be there only because it has been pushed down from above. This increases the memory available for data and leaves room for more queries and/or more sophisticated processing such as event generation or user defined functions.
4. *Extensibility:* SwissQM should be programmable to include the ability to implement user-defined functions and the ability to push down functionality from higher data processing layers. Extensibility also refers to SwissQM itself: It should be possible to extend and modify the behavior of SwissQM as needed.

3.1.2 System Architecture

A SwissQM sensor network is built out of a *gateway node* and one or more *sensor nodes*. The gateway node is assumed to have sufficient computing power and no

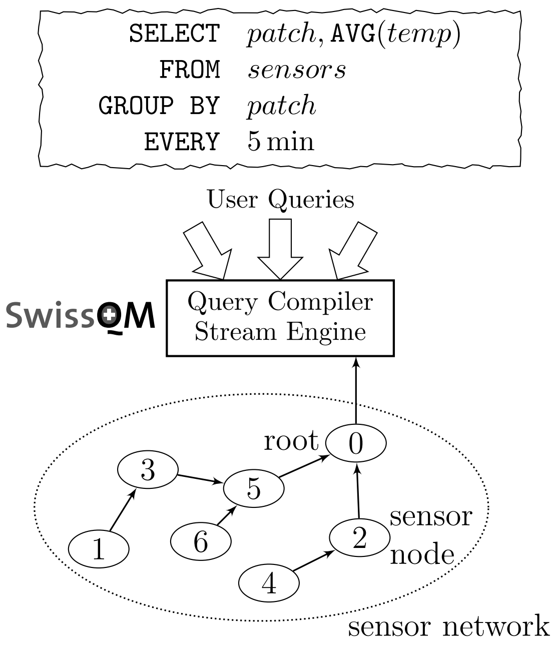


Figure 3.1: Sensors nodes are organized in a tree topology with the root node connected to the gateway

energy or memory restrictions. The gateway acts as the interface to the system. The sensor nodes are assumed to be resource constrained devices, running on batteries. The sensors perform the actual data acquisition.

The sensor nodes are organized in a collection network, i.e., a tree that routes data towards the root, where the gateway node is located as shown in Figure 3.1. This is the same strategy as used in, e.g., TinyDB, since a tree facilitates in-network aggregation and reduces the amount of routing state a node has to keep. The routing tree is built ad-hoc using the *Collection Tree Protocol* (CTP) [GFJ$^+$09]. The protocol assigns each node a link to a parent closer to the root node. The root node of the tree (node 0 in Figure 3.1) is connected to a gateway node. It is equipped with more memory and a more powerful CPU and runs the SwissQM/Gateway system. It is connected to the Internet through an Ethernet or Wifi link and provides access to the sensor networks to multiple users and applications.

3.1.3 SwissQM Gateway

Requirements 1 and 2 have led to a more radical separation between the functionality of the sensor nodes and the gateway that is typically encountered in sensor networks. Since all sensor networks require such a gateway, SwissQM has been designed to exploit the gateway as much as possible. Unless dictated by optimization strategies (e.g., minimization of data transfer), everything that can be done at the gateway is done there rather than at the sensor nodes. Thus, the gateway provides all external interfaces, as well as query optimization and compilation facilities. At the sensor nodes one finds only the code that is strictly needed to capture, aggregate, and propagate the data. Any additional code, e.g., user-defined functions, is located at the sensor nodes only if explicitly pushed down by the gateway. The resulting architecture has considerable advantages. SwissQM can support a wide variety of interfaces (as dictated by requirement 1, e.g., SQL, XQuery, Web services). These interfaces can change over time without requiring changes to the code in the sensor nodes. Sophisticated optimization strategies can be implemented at the gateway without affecting the performance of the sensor nodes. The gateway is also the natural place to implement data cleaning pipelines and virtualization such as those described by Jeffery et al. [JAF$^+$06].

Figure 3.2 shows query processing system located at the gateway. It consists of a *Query Processor* and a *Stream Processor*. The gateway processes *user queries* submitted to the system. The user queries can be expressed in various languages. The gateway processes and combines the user queries into a smaller, more optimized subset of *virtual queries*. The virtual queries are expressed in an internal format suitable for multi-query optimization, query merging, subexpression matching, window-processing optimization, etc. The optimized virtual queries are then transformed into *execution plans*. Two different types of plans are generated, *Network Execution Plans* and *Stream Execution Plans*. The former are sent into the sensor network and are executed by the sensor nodes. They are bytecode programs run by the SwissQM virtual machine. Each program generates a data stream that is fed through the collection tree to the gateway. Operators that are not pushed down into the network are applied on the stream processor as part of a stream execution plan running on a traditional main-memory stream processing engine. Its result streams are returned to the users. This three-tier mechanism virtualizes the sensor network, and permits multi-query optimization (requirement 2) across user queries for a more efficient use of the sensor network. Thanks to the virtual query step, it is also possible to use multi-query optimization on queries submitted in different query languages, e.g., SQL or XQuery.

3.1.4 The Query Virtual Machine

Requirements 3 and 4 motivated us to implement the sensor nodes as a virtual machine, the *Query Machine* (QM) (the name emphasizes the dual role of query engine and virtual machine). The query machine is a stack-based integer virtual machine. We chose a stack-based machine based on the results presented by [SCAG08]. A stack-based VM has typically smaller bytecodes for the same

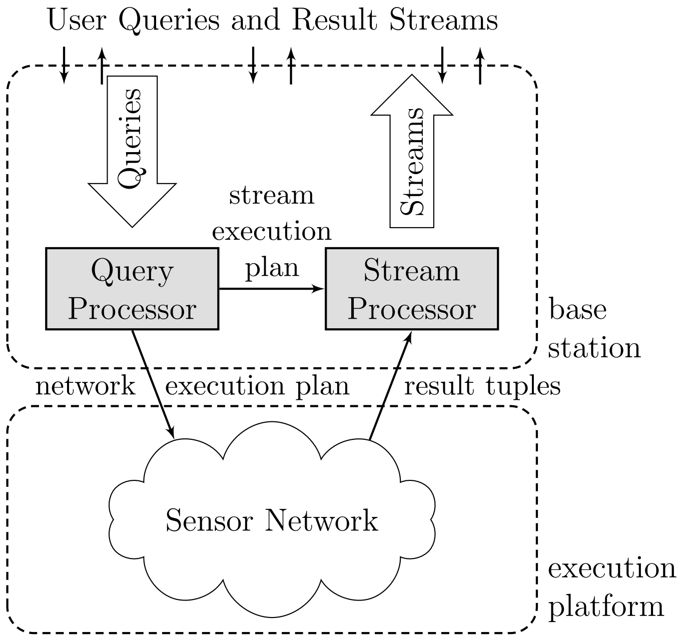


Figure 3.2: Architecture of SwissQM query processing system

program than a register-based VM. Programs of a register-based VM need less instructions than programs of a stack-based VM because no instructions are needed to put the operands onto the top of the stack. However, instructions can be represented in a much more compact way in a stack-based VM since the location of the operands is implicit, resulting in an overall more compact bytecode of programs for stack-based VMs. A compact representation of instructions is important in order to reduce the size of the bytecode for application programs. A small bytecode program consumes less memory and less bandwidth when it is disseminated, and it also increases the reliability of program dissemination. The particularly compact bytecode format is also relevant for future sensor nodes that will contain more powerful processor and are less memory-constrained. A compact program representation during transmission is still important since low-power ad-hoc radio technologies are unlikely to improve significantly in terms of bandwidth and reliability in the next years. For example, the newer ARM-based Intel Mote2 platform (see Table 2.1 on page 13) still contains the same IEEE 802.15.4 CC2420 radio as the memory-constrained Tmote Sky platform. Only if the proliferation of 3G cellular technologies also reaches the sensor networks domain the use of a traditional platform such as Java may be more important than custom-designed bytecode format. However, when comparing power requirements of current 3G devices with current CC2420-based nodes this is unlikely to happen soon.

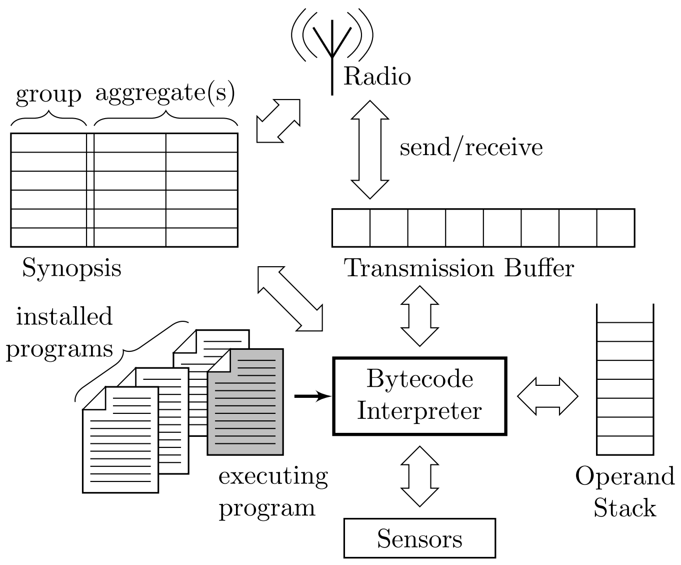


Figure 3.3: Query Machine Components

SwissQM was designed as an *interpreting virtual machine.* A *just-in-time compiler (JIT)* that translates the SwissQM bytecode into native code that can be directly executed by the microcontroller requires additional resources (code and data memory) on the device. An interpreter has a lower footprint size than a corresponding JIT-based solution. Even given abundant amount of resources, e.g., when performing the JIT operation at the less constrained base station, the memory architecture of microcontrollers such as the ATmega128L makes it difficult to use jitting. Controllers typically have different memories for code and data (Harvard architecture). It is thus difficult to execute code that was computed as data by a JIT compiler or was received over the network.

Another design consideration is the number of instructions available in the bytecode. There is an obvious trade-off between expressiveness and size: the more instructions are supported, the larger the encoding. The instruction set of the QM is a small subset of the Java Virtual Machine [LY98] extended with specialized instructions to reduce the size of the programs. The QM uses a uniform type to simplify the implementation and reduce the footprint of the QM. All data types are represented as 16-bit signed integer types (Booleans are represented using C-style semantics). Floating point types are not supported to keep the size of the instruction set as small as possible (the necessary conversion can be easily done at the gateway and that way all operations at the QM are on integers—see requirement 3).

The QM includes the following components shown in Figure 3.3: An *Operand*

Stack that stores 16-bit elements used as operands and results of instructions. A *Transmission Buffer* that temporarily holds data that are to be sent or were received but not yet processed. It is accessed through an index over the 16-bit data elements. A *Synopsis* data structure is used to maintain state for aggregation queries. The sensor nodes can exchange both the synopsis and the transmission buffer via radio. The sensors data is acquired to through a number of sensor instructions. Similar to load instructions the samples are stored onto the operand stack. A number of *QM Programs* expressed in QM bytecode (produced at the gateway from the user submitted queries) can be installed simultaneously on a node.

A design decision that made very early was to implement a dedicated SwissQM virtual machine instead of building a system on top of the Maté VM [LC02] or the publicly available Bombilla implementation. The Maté approach has several properties that render it unusable as SwissQM's execution platform. (1) In Maté programs are broken up into capsules of 24 instructions and disseminated in an epidemic-like approach. A capsule is attached to an event (reception of message or a timeout of a timer) or installed as a subroutine. A compiled query would thus consist of multiple capsules, which have to forward themselves in a viral manner through the network. Execution of a capsule starts as soon as it is received. This may lead to difficulties to achieve a coordinated execution. SwissQM uses atomic dissemination of the bytecode. The scheduling of a program is chosen such that the program starts no sooner than most of the network nodes are ready to execute the program, i.e., they have received all messages of that particular program. This minimizes potential inconsistencies in the results caused by the dissemination. (2) Maté provides one single variable (one word) that can be shared among contexts, i.e., code from different capsules. The SwissQM virtual machine provides specific memories such as the transmission and the synopsis buffers, which can be used similar to a traditional heap. (3) Maté can only run one single program at any time. In order to run multiple concurrent queries the SwissQM execution platform must be able to run multiple independent bytecode programs.

3.1.5 Query Execution

The following example illustrates the stages in SwissQM query execution platform. Assume that a user submits the following sliding window query to the gateway.

$$\begin{array}{rl}
\texttt{SELECT} & \textit{nodeid},\ \texttt{SWINDOW}(\textit{temp}, 6\,\mathrm{h}, \texttt{AVG}) \\
\texttt{FROM} & \textit{sensors} \\
\texttt{WHERE} & \textit{nodeid} < 10 \\
\texttt{EVERY} & 30\,\mathrm{s}
\end{array}$$

The query returns the average temperature 6 h sliding average of the temperature sensors on the first 10 nodes according to a static node enumeration at deploy time. The sensors are sampled every 30 seconds. A sliding window is kept for each of the sensors. It is evaluated after each new sample, i.e., every 30 seconds. Therefore, each window contains $360\,\text{min} \times 2\,\text{samples/min} = 720$ samples. This query could potentially be directly executed in the network as it is uses local data and requires only simple arithmetic operations. However, the 16-bit data words the 720 samples of each window occupy $\approx 1.4\,\text{kB}$ of memory. This exceeds the size of the available heap (synopsis). Hence, the sliding-window aggregate needs to evaluated at the gateway where memory is available for all 10 windows. The gateway splits the query into a network execution plan corresponding to the non-aggregation query

$$\begin{array}{rl} \texttt{SELECT} & nodeid,\ temp \\ \texttt{FROM} & sensors \\ \texttt{WHERE} & nodeid < 10 \\ \texttt{EVERY} & 30\,\text{s}\ . \end{array}$$

It also generates a stream execution plan it feeds to the stream engine running at the gateway (Figure 3.2), which will compute the aggregate on the $(nodeid, temp)$-stream. The aggregate stream is then returned to the user as a result to the submitted query.

3.2 SwissQM Implementation

In this section we provide a detailed description of SwissQM virtual machine implementation. The initial version was developed for Mica2 nodes running TinyOS 1 [MA06]. A second development branch was started later supporting both Mica2 and the newer Tmote Sky platforms on TinyOS 2.

3.2.1 Data Types

In order to reduce the number of instructions and the footprint size of the implementation, the query machine provides only a single data type: a signed 16-bit integer type. Boolean types used, e.g., in conditional branch instructions are interpreted following the C language rule, 0 for `false` and any other value for `true`. Floating-point types are currently not supported. When memory capacity increases in the future, floating-point support, i.e., a new data type and the corresponding instructions, can be added easily. The lack of floating-point processing is not too restrictive. First, most processors used in sensor networks do not feature a floating-point unit so that the computations need to be emulated in software anyway. Second, the raw data returned by sensors is typically an integer value

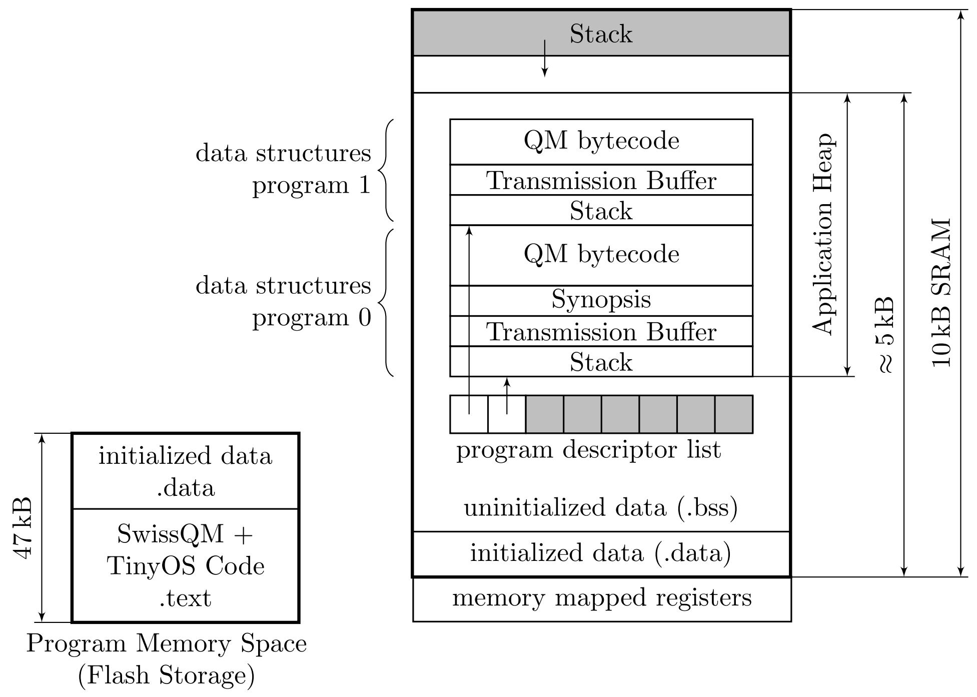


Figure 3.4: Memory layout in SwissQM with two programs (program 0 requires a synopsis, program 1 does not)

from an A/D converter; sensors rarely generate floating point values. Third, the evaluation of floating-point expressions can easily be carried out at the gateway. For example, an average value of some sensor readings can be computed as the quotient of the sum and the count, which are both integer values.

3.2.2 Memory Layout

It is important to differentiate between the SwissQM application itself and the user programs that are executed by SwissQM. We use the term *program* to refer to the latter. Both the SwissQM application and the bytecode programs must be stored in memory, but the requirements are different.

The memory of sensor nodes typically consists of persistent memory (Flash) and volatile data memory (SRAM) and is organized as shown in Figure 3.4. The SwissQM application code resides in the Flash memory. The volatile data memory in SRAM stores the global state of the SwissQM application. This memory region is further divided into two sections, the *initialized data section* (`.data`), which holds initialized global variables and constants, and the *uninitialized data section*

(`.bss`), which holds uninitialized global variables of the SwissQM application. The content of the initialized data section is copied from Flash into SRAM during the boot process. The implementation of SwissQM for the Tmote Sky platform and the full instruction set requires 4.5 kB of SRAM memory. The remaining 5.5 kB SRAM is used as stack for the SwissQM application.

SwissQM allocates a 384 byte sized heap in the volatile `.bss` section. When a new program is loaded, memory is allocated on this heap in order to store the dynamic data structures for this program as well as its bytecode. Every program has its own stack (16 bytes deep) and transmission buffer (16 bytes wide). If requested by a program, another 16 bytes can be allocated for the synopsis structure. Figure 3.4 depicts the application heap for two concurrently executed programs; a synopsis is only allocated for Program 0.

SwissQM keeps a fixed list of eight program descriptors. Each descriptor contains information about its associated program, e.g., the sampling period, the current epoch counter value, and the length of the program. Most importantly, the descriptor keeps a pointer to the program's data structures (stack, synopsis, and transmission buffer) on the heap. When a program is stopped, its program descriptor is invalidated and the heap memory deallocated. For managing the heap, a memory allocator is integrated into the SwissQM system. The complexity of this memory allocator is low because all memory allocation is static at the time when a program is loaded. That is, programs cannot dynamically allocate memory (there is no *malloc* instruction in SwissQM). Since all references to heap data use an indirection, a simple compacting allocator was implemented for SwissQM because compacting the heap only requires updating a single reference for each program.

Footprint aside, the main reason to use only static memory allocation is to allow the gateway to perform program admission control. With static allocation, the gateway knows exactly how much memory a program needs and can thus forward to the network only as many programs as can physically run on the nodes. The gateway can perform similar resource calculations for other parameters (timing, sampling intervals, etc.). This greatly simplifies the code at the nodes; programs do not need to check for any overflow condition and do not need to make allowances for thrashing situations.

The sizes of the application heap, the stack, synopsis and transmission buffer, and the maximum number of programs are deploy-time parameters. The sizes can be adjusted depending on application needs, allowing to trade off the number of programs that can be installed simultaneously onto the VM and the amount of state available for each program.

Table 3.1: Instruction set of the Query Machine (Tmote Sky configuration)

	# instructions
stack operations	16
load/store instructions	9
arithmetic and logic operations	11
control instructions	13
transmission instructions	2
sensing instructions	11
aggregation instructions	2
Total	**64**

3.2.3 Instruction Set

SwissQM uses a small subset of the integer and control instructions of the Java Virtual Machine (JVM) specification [LY98]. Table 3.1 shows the basic set of instructions of the QM for the Tmote Sky platform. The latest implementation of the system for the Tmote Sky platform contains 64 bytecode instructions. 37 instructions are identical to the JVM specification. The instructions are grouped by functionality in Table 3.1. The first group includes instructions for stack-manipulation operations, load/store, arithmetic and logic operations, and control instructions such as conditional and unconditional jumps. Together with two transmission instructions this first group represents the core instruction set of the VM. The remaining 27 instructions provide access to the sensors or are used for accessing the synopsis and the transmission buffer. The complete set of SwissQM instructions for the Mica2 platform is listed in Table 3.2. The difference is between the two instruction sets are the platform-dependent sensor instructions. Eleven sensor instructions for Tmote Sky and seven for Mica2, as well as a number of application-specific instructions extend the core set. The sensor instructions access the physical sensors or system parameters and push the obtained value onto the stack. Two transmission instructions are provided: `send_tb` sends the content of the transmission buffer whereas `send_sy` sends the current aggregation state stored in the synopsis. Finally, the SwissQM platform provides two specialized extensions for in-network aggregation based on the TAG approach [MFHH02]. The aggregation instruction `merge` is described in Section 3.2.6.

Each bytecode instruction is encoded using one single byte, which imposes a limit of 256 instructions. Total instruction length including operands can be one, two, or three bytes, depending on whether the instruction has implicit operands (e.g., `iadd`), an 8-bit immediate operand (e.g., `ipushb`=push byte immediate), or a

16-bit immediate operand (e.g., `ipushw`=push word immediate). The implementation of instructions is modular to enable extensibility. Each group is implemented as a separate nesC component. The groups of the current SwissQM design and the space each group requires both in ROM (i.e., program Flash) and RAM memory is shown in Table 3.2.

Table 3.2: The complete SwissQM instruction set (Mica2 configuration)

				Bytes in	
	Instruction	**Description** [a]	**Time**	**ROM**	**RAM**
Stack Instructions	`dup`	$\alpha, x \Rightarrow \alpha, x, x$	15.2 μs	1,562	0
	`dup_x1`	$\alpha, y, x \Rightarrow \alpha, x, y, x$	43.6 μs		
	`dup_x2`	$\alpha, z, y, x \Rightarrow \alpha, x, z, y, x$	59.0 μs		
	`dup2`	$\alpha, y, x \Rightarrow \alpha, y, x, y, x$	51.0 μs		
	`dup2_x1`	$\alpha, z, y, x \Rightarrow \alpha, y, x, z, y, x$	67.0 μs		
	`dup2_x2`	$\alpha, w, z, y, x \Rightarrow \alpha, y, x, w, z, y, x$	83.0 μs		
	`pop`	$\alpha, x \Rightarrow \alpha$	14.4 μs		
	`pop2`	$\alpha, y, x \Rightarrow \alpha$	23.6 μs		
	`swap`	$\alpha, y, x \Rightarrow \alpha, x, y$	37.6 μs		
	`iconst_0`	$\alpha \Rightarrow \alpha, 0$	12.0 μs		
	`iconst_1`	$\alpha \Rightarrow \alpha, 1$	12.0 μs		
	`iconst_2`	$\alpha \Rightarrow \alpha, 2$	12.0 μs		
	`iconst_4`	$\alpha \Rightarrow \alpha, 4$	12.0 μs		
	`iconst_m1`	$\alpha \Rightarrow \alpha, -1$	12.0 μs		
	`ipushb <int8>`	$\alpha \Rightarrow \alpha, \text{sign\_ext}(i)$	13.4 μs		
	`ipushw <int16>`	$\alpha \Rightarrow \alpha, i$	13.4 μs		
Buffer Instructions	`iload <int8>`	$\alpha \Rightarrow \alpha,$ `buf[i]`	20.8 μs	1,528	44
	`iload`	$\alpha, i \Rightarrow \alpha,$ `buf[i]`	29.4 μs		
	`istore <int8>`	$\alpha, x \Rightarrow \alpha$ and `buf[i]` $= x$	23.4 μs		
	`istore`	$\alpha, x, i \Rightarrow \alpha$ and `buf[i]` $= x$	31.0 μs		
	`iload_sy <int8>`	$\alpha \Rightarrow \alpha,$ `syn[i]`	21.8 μs		
	`iload_sy`	$\alpha, i \Rightarrow \alpha,$ `syn[i]`	30.2 μs		
	`istore_sy <int8>`	$\alpha, x \Rightarrow \alpha$ and `syn[i]` $= x$	23.8 μs		
	`istore_sy`	$\alpha, x, i \Rightarrow \alpha$ and `syn[i]` $= x$	31.4 μs		
	`send_tb`	send transmission buffer	22–42 ms		
	`send_sy`	send synopsis	22–42 ms		

[a]For each instruction the state of the operand stack before and after the invocation is specified on the left and the right of $\Rightarrow$ respectively. α is used to designate the remainder of the stack that is left unchanged by the instruction.

continued from previous page

Arithmetic and Logic Instructions	`iadd`	$\alpha, y, x \Rightarrow \alpha, y + x$	41.0 μs	1,056	0
	`isub`	$\alpha, y, x \Rightarrow \alpha, y - x$	41.0 μs		
	`imul`	$\alpha, y, x \Rightarrow \alpha, y * x$	42.0 μs		
	`idiv`	$\alpha, y, x \Rightarrow \alpha, \lfloor y/x \rfloor$	71.0 μs		
	`irem`	$\alpha, y, x \Rightarrow \alpha, y \bmod x$	73.0 μs		
	`ineg`	$\alpha, x \Rightarrow \alpha, -x$	30.8 μs		
	`iinc`	$\alpha, x \Rightarrow \alpha, x + 1$	30.0 μs		
	`idec`	$\alpha, x \Rightarrow \alpha, x - 1$	30.8 μs		
	`iand`	$\alpha, y, x \Rightarrow \alpha, y \& x$	40.4 μs		
	`ior`	$\alpha, y, x \Rightarrow \alpha, y \| x$	41.6 μs		
	`inot`	$\alpha, x \Rightarrow \alpha, \tilde{}x$	31.2 μs		
Control Instructions	`if_icmpeq <int8>`	$\alpha, y, x \Rightarrow \alpha$, jump if $y = x$	34.2 μs	1,278	0
	`if_icmpneq <int8>`	$\alpha, y, x \Rightarrow \alpha$, jump if $y \neq x$	36.2 μs		
	`if_icmplt <int8>`	$\alpha, y, x \Rightarrow \alpha$, jump if $y < x$	35.0 μs		
	`if_icmple <int8>`	$\alpha, y, x \Rightarrow \alpha$, jump if $y \leq x$	36.4 μs		
	`if_icmpgt <int8>`	$\alpha, y, x \Rightarrow \alpha$, jump if $y > x$	34.4 μs		
	`if_icmpge <int8>`	$\alpha, y, x \Rightarrow \alpha$, jump if $y \geq x$	35.2 μs		
	`ifeq <int8>`	$\alpha, x \Rightarrow \alpha$, jump if $x = 0$	25.2 μs		
	`ifneq <int8>`	$\alpha, x \Rightarrow \alpha$, jump if $x \neq 0$	23.4 μs		
	`iflt <int8>`	$\alpha, x \Rightarrow \alpha$, jump if $x < 0$	22.6 μs		
	`ifle <int8>`	$\alpha, x \Rightarrow \alpha$, jump if $x \leq 0$	25.8 μs		
	`ifgt <int8>`	$\alpha, x \Rightarrow \alpha$, jump if $x > 0$	23.8 μs		
	`ifge <int8>`	$\alpha, x \Rightarrow \alpha$, jump if $x \geq 0$	25.8 μs		
	`goto <int8>`	$\alpha \Rightarrow \alpha$, jump always	6.5 μs		
Sensor Instructions	`get_nodeid`	$\alpha \Rightarrow \alpha, \textit{nodeid}$	12.1 μs	1,854	5
	`get_parent`	$\alpha \Rightarrow \alpha, \textit{parent}$	13.4 μs		
	`get_light`	$\alpha \Rightarrow \alpha, \textit{light}$	342 μs		
	`get_temp`	$\alpha \Rightarrow \alpha, \textit{temp}$	348 μs		
	`get_noise`	$\alpha \Rightarrow \alpha, \textit{noise}$	3.38 ms		
	`get_tone`	$\alpha \Rightarrow \alpha, \textit{tonecount}$	3.38 ms		
	`get_voltage`	$\alpha \Rightarrow \alpha, \textit{batteryvoltage}$	804 μs		
Aggregation Instructions	`merge`	$\alpha, \mathrm{aggop}_m, \ldots, \mathrm{aggop}_1, m, n \Rightarrow \alpha$	# transmission	1,064	0
		n: number of grouping exprs			
		m: number of aggregates			
		aggop_i: ith agg. operation	30 μs/row		
	`clear_sy`	clear synopsis	6.9 μs		

Instead of providing a dedicated instruction for each sensor, sampling could also be implemented using a virtual function call. For example, an instruction similar to `invokevirtual` available in the Java virtual machine could be used instead. The actual sensor to be accessed is specified as an index into a vtable-like structure, similar to the index into the constant pool in Java. Although, this introduces the ability of "overwriting", i.e., redefining sensors, it has two disadvantages that render this approach inapplicable for SwissQM. First, the size of the instruction is larger since space for the function index must be provided in the immediate field.

For example, `invokevirtual` in Java uses a two-byte index. The chosen solution uses one single byte to express all sensing instructions. Second, the sampling code can lo longer be easily dispatched by the virtual machine. Since every SwissQM instruction is one full byte it can be used to directly dispatch an instruction. This is implemented in nesC using a parameterized interface with the instruction byte as a parameter. This allows a loose coupling of instruction's implementation components to the VM core by using one single wiring statement in the nesC configuration module of the VM.

Following requirement 4, the instruction set is modular. As need dictates (or device capabilities evolve), additional instruction classes can be added or removed (e.g., more sophisticated aggregation or floating point support). Additionally, when SwissQM is ported to a different platform the sensing instructions have to be adapted in order to reflect the physical sensors (humidity, barometric pressure, acceleration, etc.) that are available on that particular platform. It is also possible to extend the initial instruction set with more powerful instructions. For instance, if an application ends up generating programs in which a long sequence of instructions appears often, it is possible to encode that sequence into a single bytecode instruction, as already suggested by Muller et al. [CSCM00]. In general, such optimizations are application-specific and involve language optimizations and memory trade-offs that must be taken into account by the compiler. Nevertheless, the possibility of extending the bytecode instruction set and implementing such optimizations is one of the key advantages of SwissQM. As an example, additional application-specific instructions were added to SwissQM for example by Doman et al. [DPD10] that provide functionality to evaluate the fuzzy membership functions.

3.2.4 QM Programs

Data acquisition tasks in sensor networks are performed according to one of the following two sequences of well defined phases. The first sequence is used when no in-network aggregation takes place, i.e., a node operates only on its own data and forwards it. The phases in this sequence are *sampling*, *processing*, and *sending*. The second sequence is applied when nodes aggregate their data with the data received from other nodes before forwarding the result. This second sequence has four phases: *sampling*, *processing*, *merging received data*, and *sending*.

The program structure in SwissQM is divided into three sections that are combined to implement these different processing phases. The sections are similar to code capsule handlers in Maté [LC02]. The difference in SwissQM is it initializes the context, i.e., the content of the transmission and the synopsis buffer, depending on the section that is invoked when the corresponding event occurs. SwissQM supports the following sections:

Delivery Section. The instructions of the delivery section of a program are periodically executed in response to a timer event. Associated with the delivery section is an execution interval, called *sampling period* that determines the time between two invocations of this section. The delivery section is intended to be used for instructions that sample the sensors and generate a stream of data tuples that is sent towards the gateway node. Result messages generated in the delivery sections contain an *epoch number* that associates the data of the message with a particular invocation of the delivery section. The epoch number is incremented after the execution of the delivery section is completed. By default, the epoch number is always added to the data sent by an application. An optional argument specifies whether the synopsis is to be implicitly cleared at the end of each execution (section argument `"epochclear"`) or whether the `clear_sy` instruction must be used to explicitly clear the synopsis (section argument `"manualclear"`). Any valid QM program must at least contain a *delivery* section.

Reception Section. This section is executed when a node receives a message from any of its children. The section is optional and is used to intercept the message, e.g., in order to aggregate data obtained from the children. If no reception section is present in a QM program, a node simply forwards the data to its parent.

Init Section. This section is executed once at the beginning before the delivery and reception sections are executed for the first time. It is optional and can be used to initialize the synopsis.

Figure 3.5 shows the life cycle of a QM program. After a new program is loaded its *init* section is executed. Depending on a timer interrupt and the reception of a message the *delivery* and *reception* sections are scheduled until the program is stopped. A QM program is identified by its program number. Messages generated by the program contain this identification number to allow nodes to correlate messages and programs. The program number is used to identify the program whose reception section has to be scheduled after a message is received.

3.2.5 Simple Program Examples

Basic Sampling and Sending. A first example of a SwissQM program is a simple program that samples the temperature once every minute. Every node reports the temperature and its node ID. Intermediate nodes simply forward the messages from their children. The corresponding QM program is as follows:

```
.section delivery, "@60s"
    get_nodeid        # read the node's ID
```

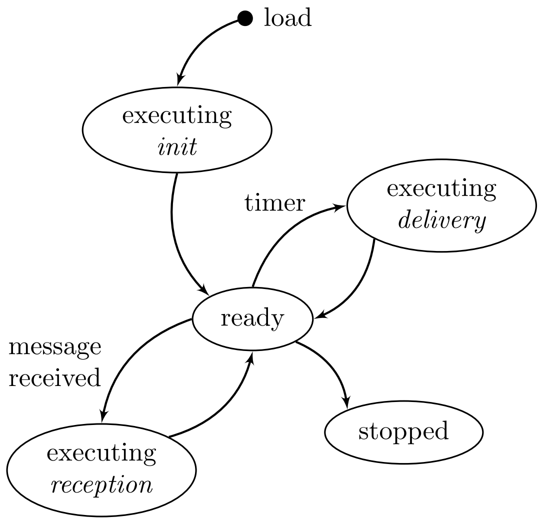


Figure 3.5: Life cycle of a QM program

```
3      istore     0    # store it at pos. 0
4      get_temp        # read temperature sensor
5      istore     1    # store it at pos. 1
6      send_tb         # send transmission buffer
7
8 .section reception
9      send_tb         # forward tuple from child
```

Lines 1 and 8 declare the two sections (delivery and reception) defined in this QM program. In the *delivery* section the node ID is read and pushed on the operand stack (Line 2). This node ID is copied to the first position of the transmission buffer (Line 3). Next, the temperature sensor is read and its value is copied into the second position of the transmission buffer (Lines 4 and 5). The content of the transmission buffer is sent to the parent node (Line 6). The QM knows the size of the transmission buffer from the indices used in previous instructions to copy data into the transmission buffer.

In this example, the *reception* section can be omitted because the default behavior of simply forwarding incoming messages from the children to the parent is sufficient. For presentation purposes, Lines 8 and 9 of the example program show how this default behavior can be implemented in a SwissQM program. A use case for the reception section is to filter out messages received from the children and not forwarding them by adding a conditional branch around the `send_tb` instruction on Line 9. This example program requires 8 bytes and can be disseminated to the sensor nodes using one single message.

This simple example already illustrates two important properties of SwissQM. First, any code section can write to the data transmission buffer. As a result, SwissQM must take care of race conditions. SwissQM avoids race conditions by executing each code section in an atomic way. This mutual exclusion is enforced through global variables. Second, several obvious optimizations are possible. For instance, a node could merge its data with the data received from its children into a single message. This approach reduces the number of messages even if no aggregation is carried out.

Control Instructions. A second example illustrates how to generate and propagate events. In this example, the nodes sample their light and temperature sensors every 10 seconds. If at least one of these readings exceeds a given threshold, the node reports its ID. This behavior corresponds to the query:

```
SELECT  nodeid
  FROM  sensors
 WHERE  light > 100   OR   temp > 60°C
 EVERY  10 s  .
```

A possible use of this program is the detection of fire and its localization (assuming the position of the nodes is known at the gateway). The program is as follows:

```
 1 .section delivery, "@10s"
 2       get_light          # read light sensor
 3       ipushw     900     # push light threshold value
 4       if_icmpgt  send    # jump to send if greater
 5       get_temp           # read temp sensor
 6       ipushw     500     # push temp threshold value
 7       if_icmpgt  send    # jump to send if greater
 8       goto       skip    # skip sending
 9 send: get_nodeid         # get the node's ID
10       istore     0       # store it at pos. 0
11       send_tb            # send transmission buffer
12 skip:
```

The code is largely self-explanatory. The *reception* section has been omitted because the default behavior is sufficient. The bytecode of this program consists of 18 bytes and can be disseminated using two messages. An important aspect of this example is the execution time. When a message is sent, the program can take up to 65 ms; the bulk of the time is spent by the `send_tb` instruction in Line 11 as the MAC layer needs to be accessed and the data sent over the low bandwidth radio. If no message is generated, the program takes 22 ms. In this case, most of the time (10 to 20 ms) is spent by the TinyOS sensor code, which enforces a delay between two samples of different sensors. The TinyOS code specifies a 10 ms delay; in practice, however, the delay can be up to 20 ms due to timing issues in the TinyOS event scheduler. This type of complex overhead calculations for the programs is one of the reasons to perform the planning of the execution of concurrent programs at the gateway. The bottom line is that these subtle but crucial problems are one of the reasons to use SwissQM in the first place: Having a well characterized bytecode language makes the cost analysis feasible and the predictions over the resources consumed by a program sufficiently accurate.

Keeping State. A third example illustrates how a program can keep state across different invocations. Such state cannot be kept in the transmission buffer because the transmission buffer is flushed after the execution of a code section. To keep state across invocations of a code section, a program must declare the use of a synopsis. In this example, every node samples its light sensor every five seconds. In order to smooth noise from the sensors, a user-defined function with an *exponential weighted moving average* (EWMA) filter is applied to the readings (smoothing factor $\alpha = 0.8$), i.e., the filter produces

$$y_k = \alpha y_{k-1} + (1-\alpha)u_k \qquad k > 0 \ ,$$

where y_k is the new output value, y_{k-1} is the value produced in the last iteration, and u_k is the actual sensor reading. The 1st order filter is initialized as $y_0 = u_k$. The nodes generate a stream of tuples containing the value of y_k and their ID.

```
.section init
   get_nodeid        # get the node's ID
   istore_sy   0     # store it in synopsis pos. 0
   get_light         # sample u_0
   istore_sy   1     # store it as y_0 in synopsis

.section delivery, "@5s","manualclear"
   get_light         # sample u_k
   iload_sy    1     # read y_{k-1}
   isub              # u_k - y_{k-1}
   ipushb      5     # push 5 on stack
   idiv              # (u_k - y_{k-1})/5
   iload_sy    1     # read y_{k-1}
   iadd              # y_{k-1} + (u_k - y_{k-1})/5
```

```
15  istore_sy   1     # update synopsis $y_{k-1} \Leftarrow y_k$
16  send_sy           # send synopsis
```

In addition to the *delivery* section, this program also contains an *init* section that is called once before the *delivery* section is run for the first time. In this example, the synopsis is used to store the filter output y_k. It also shows that the synopsis can be used as buffer for the result message to be sent (Line 16). An interesting optimization that a compiler could take advantage of is to store data that must be sent in every invocation but never changes (e.g., the node ID) in the synopsis. In the program, the node ID is setup once in the *init* section (Lines 2–3) when the synopsis is first initialized. Lines 4 and 5 sample the sensor and store it in the synopsis to provide an initial value y_0. Since the synopsis state has to be kept between invocations of the *delivery* section, the argument `"manualclear"` is specified in the section declaration (Line 7). This declaration implies that a synopsis will be allocated when the program is loaded. In the *delivery* section the light sensor is sampled (Line 8), and y_{k-1} is read from the first position of the synopsis (Lines 9 and 13). Lines 10–14 compute y_k such that the errors introduced by integer arithmetic are minimal. The synopsis is updated (Line 15) and the complete synopsis (node ID at position 0 and y_k at position 1) is sent to the parent (Line 16). The size of the complete program is 19 bytes and fits into two TinyOS messages.

The execution of the *delivery* section requires 23–43 ms. The execution time without `send_sy` is only 940 μs. Thus, computation costs are negligible compared to the cost of sending a message. This observation is a clear indication that the processing capacity of the nodes can be used without significantly interfering with energy consumption or the time-budget of a program. In general, the computation is only limited by the amount of memory available. Again, one of the important advantages of SwissQM is that it makes memory consumption (only static allocation) as well as time and energy consumption of programs predictable so that all planning can be carried out at the gateway.

3.2.6 In-network Aggregation

In-network data processing has the potential to reduce the number of messages sent to the gateway [MFHH02]. The most relevant form of in-network processing is data aggregation. SwissQM implements it using the synopsis and a specialized `merge` instruction.

Synopsis. In SwissQM, the synopsis is a fixed size buffer of 16 bytes allocated on demand when a program is loaded. As shown in the previous examples, SwissQM has several instructions to manipulate and send the synopsis. The synopsis can

Table 3.3: Aggregates supported by the `merge` instruction

Aggregate	**State**	**Initializer**	**Merger**	**Finalizer**
`COUNT`	c	$c = 1$	$c = c_1 + c_2$	$= c$
`MAX`	m	$m = \mathit{expr}$	$m = \max(m_1, m_2)$	$= m$
`MIN`	m	$m = \mathit{expr}$	$m = \min(m_1, m_2)$	$= m$
`SUM`	s	$s = \mathit{expr}$	$s = s_1 + s_2$	$= s$
`AVG`	(s, c)	$(\mathit{expr}, 1)$	$(s_1 + s_2, c_1 + c_2)$	$= {}^{s}/_{c}$
`VARIANCE`	(s, t, c)	$(\mathit{expr}, \mathit{expr}^2, 1)$	$(s_1 + s_2, t_1 + t_2, c_1 + c_2)$	$= {}^{t}/_{c} - {}^{s^2}/_{c^2}$

be used in two ways. In *raw mode*, the synopsis is regarded as an array of 16-bit elements, like the transmission buffer, that can be accessed over an element index. The instructions `iload_sy` and `istore_sy` are used to load data from the synopsis onto the stack and store data from the stack into the synopsis. The EWMA filter example shown in Section 3.2.4 uses this mode of operation. In *managed mode*, the synopsis is accessed through the `merge` instruction that combines data from the transfer buffer with the synopsis.

Merge Instruction. The `merge` instruction is used to express a complex operation with a single bytecode instruction. As such, it is an example of the type of powerful instructions that can be included in SwissQM to capture what otherwise would be a repetitive sequence of instructions (since the type of aggregation performed tends to be similar across many applications). `merge` is a parameterized instruction that implements the aggregate operations shown in Table 3.3. Which one it performs depends on the parameters that it finds on the stack. The `merge` instruction has the following parameter format:

$$\texttt{merge}(n, m, \mathrm{aggop}_1, \ldots, \mathrm{aggop}_m) \ .$$

The first parameter is the number of grouping expressions n. The second parameter m is the number of aggregation expressions that follow. The parameter aggop_i are constants that specify the aggregation operations (first column in Table 3.3). m, n, and the aggregation type constants are pushed on the stack immediately before calling the `merge` operation.

```
ipushb  aggop_m    # last aggregate operation
   ⋮
ipushb  aggop_2    # 2nd aggregate operation
ipushb  aggop_1    # 1st aggregate operation
ipushb  m          # number of aggregate expressions
ipushb  n          # number of grouping expressions
merge              # invoke merge
```

In Table 3.3, the state column shows the 16-bit variables used for each type of aggregation. Following the TAG [MFHH02] approach, aggregation involves three functions. The *initializer* function of an aggregate is used to create the initial aggregation state, e.g., the state that represents the data sampled by a node from its own sensors. This state is then merged with the aggregation state received from its children using the *merger* function. The initializer and merger functions run on the nodes. In SwissQM, the finalizer function is applied on the gateway. The finalizer computes the final value of the aggregation from the received data (see example below and Figure 3.6). Although not strictly necessary, executing the finalizer at the gateway allows more complex aggregation functions to be implemented without having to worry about the impact of such operations on the sensor nodes (e.g., lack of floating point support or complex math processing).

Merge example. The example illustrates the use of the `merge` instruction and how query processing à la TinyDB can be implemented in SwissQM. By generalizing this implementation, it is easy to see how a system like TinyDB could be implemented on top of SwissQM. Consider the following TinyDB query:

$$\begin{array}{rl} \texttt{SELECT} & \textit{parent}, \texttt{MAX}(\textit{light}) \\ \texttt{FROM} & \textit{sensors} \\ \texttt{GROUP BY} & \textit{parent} \\ \texttt{EVERY} & 10\,\mathrm{s}\ . \end{array}$$

This query returns the maximum light value among all nodes that have the same parent node in the tree. Leaf nodes simply send their synopsis containing the light value sampled every ten seconds and the ID of their parent. Intermediate nodes send a synopsis that includes: (1) its own parent and its own light value and (2) a pair (node ID, `MAX(light)`) for every non-leaf node in its subtree. In the query, `parent` is the grouping expression and `MAX(light)` the aggregate expression. In general, there can be more than one grouping expression that forms the aggregation groups, as well as multiple aggregate expressions.

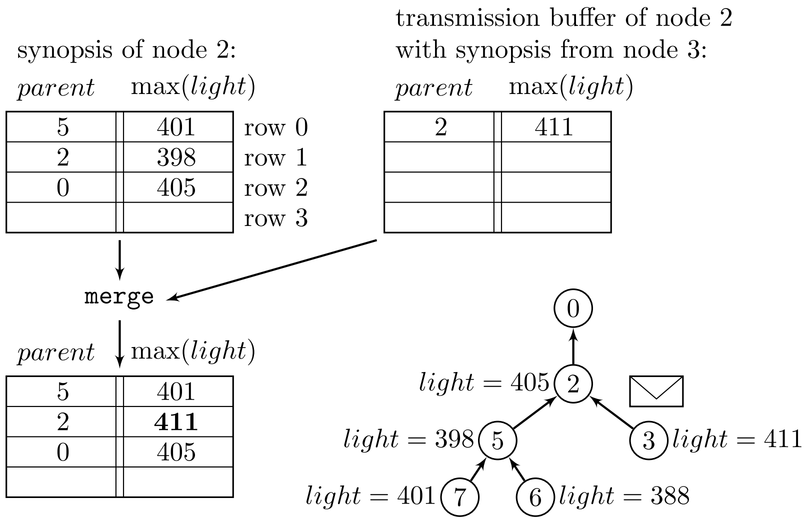


Figure 3.6: Merging aggregation state from the transmission buffer to the local synopsis

In “managed mode”, the synopsis is organized as a table. Each line of the synopsis contains information about one group, i.e., a parent ID and the maximum light reading for that ID. Figure 3.6 shows an example routing tree and the synopsis of node 2. Node 3 sends its synopsis to node 2, which will copy the message into the transmission buffer. Using its own synopsis and the transmission buffer, node 2 updates the synopsis and replaces the entry for parent 2. The update of the local synopsis is carried out by the `merge` instruction. Next, node 2 forwards the updated synopsis to node 0, which in turn carries out the same calculations in order to report the final result to the gateway.

The TinyDB query and the in-network aggregation is translated into the following SwissQM bytecode program.

```
.section delivery, "@10s","epochclear"
   get_parent         # get ID of this node's parent
   istore      0      # store it as group expression
   get_light          # read light sensor
   istore      1      # store it as partial agg. state
   iconst_2           # aggregate type MAX=2
   iconst_1           # number of aggregate expressions
   iconst_1           # number of group expressions
   merge              # merge transmission buffer
   send_sy            # send synopsis
```

```
11
12 .section reception
13     iconst_2            # aggregate type MAX=2
14     iconst_1            # number of aggregate expressions
15     iconst_1            # number of group expressions
16     merge               # merge transmission buffer
```

The aggregation state added by a node is created in the delivery section. Since the `merge` instruction always merges data from the transmission buffer to the local synopsis, the transmission buffer must be prepared before the `merge` instruction is invoked. The same layout must be used in the transmission buffer as in the synopsis. The initial aggregation state consists of a single line with two 16-bit elements, the ID of the node's parent and the reading from the light sensor. The transmission buffer is prepared in Lines 2–5. The `merge` instruction looks for its parameters on the stack. The first parameter at the top of the stack is the number of grouping expressions (in the example, it is 1, the parent ID). The second parameter, as we proceed down the stack, is the number of aggregate expressions (in the example, it is 1, `MAX(light)`). The third parameter is a numeric code indicating the aggregation operation to perform (1=`COUNT`, 2=`MAX`, etc.). If there are more aggregate expressions (the second parameter is > 1), the stack contains one entry indicating the operation for each aggregate expression. [2] These parameters are pushed onto the stack in Lines 6–8. In order to merge the partial aggregation state received from the children, the `merge` instruction is executed a second time in the reception section (Lines 13–16). A message to the parent containing the aggregation state of a node (i.e., the synopsis) is only sent in the delivery section; incoming messages are merged into the synopsis but never forwarded to the parent. The delivery section is declared with the `epochclear` directive; `epochclear` specifies that the synopsis is to be cleared when the delivery section completes. The bytecode of this program is 15 bytes in size and can be sent in one program message. For comparison, TinyDB, disseminates this query in three messages using 168 bytes.

Managing aggregation. The aggregation example above can be used to illustrate several of the complex system problems associated to in-network processing. The first one is that, as it is done in the example, the synopsis is only sent in the delivery section. This prevents a synopsis being forwarded every time a child sends its data. For the procedure to be correct, when a delivery section of a node is executed, it should have received all the synopses of its children. This leads to a staged execution schedule in which nodes deeper in the tree need to be activated before their parents. In other words, the delivery section must be scheduled *earlier*

[2]The same mechanism is used in C to implement open parameter lists.

the deeper a node is in the tree, such that the execution of the delivery section of a node overlaps with the time its parent is listening for results. This requires synchronized program execution of all nodes in the tree. In SwissQM nodes constantly exchange routing information (see Section 3.2.8). With this information, a node knows the depth of its position in the tree. Each node then uses a simple algorithm for shifting its schedule according to its position in the tree. Such a mechanism is needed by any implementation of the TAG approach [MFHH02] for in-network data aggregation. The advantage of SwissQM is that the timing characterization can be made more precise because of the well characterized building blocks involved.

The second problem is what to do when a node runs out of space and is not able to store the whole aggregation state. In the example of the previous subsection, the aggregation state becomes larger as the aggregation proceeds up the tree. If the network is deep, the whole aggregation state possibly does not fit in the synopsis of a node that is at a high level in the routing tree. As mentioned earlier, SwissQM does not support dynamic memory allocation and that for good reasons: Given the memory constraints of mote-scale sensor nodes, dynamic memory allocation would not solve the problem anyway; instead it would make things worse because the resources consumed of a QM program could not be predicted. The SwissQM solution to this problem is as follows: Once the synopsis of a node is full, the `merge` operation notices this and simply forwards all the synopses it receives without merging. The final aggregation is then performed at the gateway. In such cases, the SwissQM approach to in-network aggregation reduces the number of messages invoked by nodes at the lower layers of the routing tree and SwissQM does as well as possible.

The execution time of the `merge` instruction depends both on the size of the synopsis and the state stored in the transmission buffer. Nevertheless, and in spite of its complexity, the `merge` instruction is not too expensive. Initializing the synopsis (i.e., merging the transmission buffer with a previously empty synopsis) requires about 90 μs depending on the grouping and aggregation expressions. Each additional aggregation row adds another 30 μs to the overall execution time. In other words, merging m rows in the transmission buffer with an n-row local synopsis takes approximately $nm \cdot 30$ μs.

3.2.7 Multiprogramming

Multi-query support is provided on two layers: first, by merging different user queries and second, by multi-programming in the QM. Multi-programming in the QM is done through sequential execution of the programs. The execution duration of a program is typically short compared to the sampling interval. Program execution including data capturing and merging is in the order of microseconds. Data transmission is in the order of milliseconds. Sampling periods are in the order of seconds or larger. Thus, without any further optimization, it is possible to run a number of programs even with the shortest sampling period of one second. For instance, in the examples shown in Section 3.2.5 the execution of the *reception* section takes in the order of 30 ms on Mica2 sensor nodes, with a sampling period in the order of tens of seconds. From a CPU point of view there is room for over 100 such queries. Of course, there is a trade-off between the number of programs that can be executed and the memory available to each program, i.e., the size of the synopsis for storing aggregation state, the size of the stack and the transmission buffer. When combined with query-merging and multi-query optimizations it is possible to support a relatively large number of user queries we run over hundred user queries as we will show in Chapter 4.

3.2.8 Topology Management

The SwissQM routing tree is formed using the *Collection Tree Protocol* (CTP) [GFJ+09] on TinyOS 2 and the *MintRouting* protocol [WTC03] in the implementation for TinyOS 1. The MintRouting protocol is also used in TinyDB. The tree provides the routing structure to send results back to the gateway. Every node in the tree maintains a link to a parent node closer to the root. Thus, a result message is sent to the parent node, which will then forward the message to the next node closer to the root.

A novel aspect of SwissQM is the embedding of clock synchronization information into the routing messages of the network layer. This piggy-backing avoids the cost of separate time-synchronization messages such as those used in TinyDB. Clock synchronization plays a key role in program scheduling, particularly when aggregation is involved. Every routing message is timestamped by the sender. Whenever a routing message is received from its parent, a node adjusts its clock to the time found in the time stamp of the received message minus an average transmission delay (18 ms, a deployment parameter). Experiments show that this simple protocol is accurate enough. An alternative would be to use a more accurate but more complex protocol such as TPSN [GKS03] that requires an explicit two-way message exchange.

3.2.9 Program Dissemination

QM programs are split into several *fragment messages*. Every fragment message contains the identification number of the program as well as an enumeration number of the fragment. The first fragment message contains metadata of the program, i.e., the length of the *init*, *delivery*, and *reception* section, the sampling period, and a number of program flags such as the presence of a synopsis, the synopsis-clear

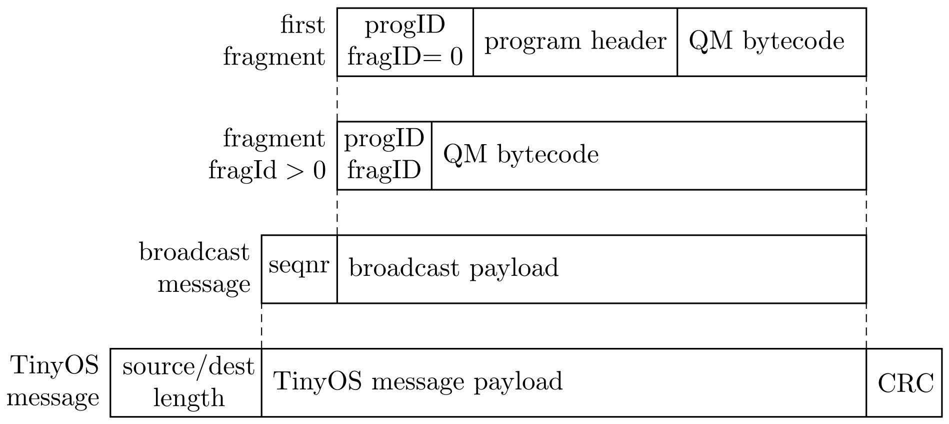


Figure 3.7: Message mapping: fragment message → broadcast message → TinyOS message

mode (*"manualclear"* or *"epochclear"*) as well as the time when the delivery section is invoked the first time. This program header information uses 10 bytes in the first fragment, leaving 16 bytes for the bytecode of the program. Starting with the second and following fragments, the fragment messages only contain the program and the fragment ID (2 bytes), leaving 24 bytes for the program bytecode. The fragment messages are sent as payload over the broadcast layer. The payload size of a broadcast message is 26 bytes. The broadcast layer adds a sequence number to the message. A sequence number in the broadcast message guarantees that a node rebroadcasts a message exactly once. The size of a broadcast message header is three bytes. A broadcast message is stored as the payload of a TOS message, the basic message type provided by TinyOS. The size of a TOS message is 36 bytes (the default message size on TinyOS). The message mapping between the layers is shown in Figure 3.7.

Since wireless ad-hoc networks expose a high bit-error rate, messages are often corrupted and/or lost. SwissQM uses two mechanisms to alleviate the effects of lost messages that contain program fragments. The first mechanism is based on a timeout for program reception and the second on snooping result messages.

When the first fragment message is successfully received, the number of outstanding fragments is computed from the lengths of the program sections contained in the first fragment. Next, the dynamic program structures (stack, transmission buffer, synopsis) are allocated on the heap and a timer is started. When this timer times out after two seconds before all fragments of this program are received, a node sends a *program request* message to its neighbors and restarts the timeout timer. The request message contains the program ID and the missing fragments encoded as a bit-mask (16 bits). A node that receives this message and has the specified program, generates the requested fragments for the program. The node then sends each fragment in a *program reply* message back to the requesting node. The reply message also contains the current epoch count and the relative offset to the start of the next scheduled execution. This allows a node to join in even if the program execution has already started in the meantime.

In addition to this mechanism and in order to accommodate nodes that join late, nodes snoop on the result messages. If a node sees a result message with a program identifier it is not aware of, it sends out a request for all program fragments. Since it is unable to determine the number of fragments of the program, it sets all bits in the 16-bit fragment bitmap of the request message. As soon as it receives fragment 0, it will recompute this mask from the section lengths found in that fragment. This snooping approach is also used in TinyDB. The difficulty of getting a program disseminated to all nodes is the reason why SwissQM places so much emphasis on compact bytecode and the use of highly expressive instructions, such as `merge`.

3.3 SwissQM Gateway

The SwissQM/Gateway system is implemented in Java as a set of OSGi [OSG09] components and services. OSGi allows the life cycle of Java components, the so-called bundles. The use of OSGi has an additional advantage that services can be easily be accessed remotely through *R-OSGi* [RAR07]. This allows a tighter integration into a client than possible by, for example, a traditional JDBC integration. Components can be easily migrated from the gateway to fat clients such as the SwissQM plug-in available for the Eclipse platform, which can be used to control the sensor network, register queries and user-defined functions. Thin clients only need to access the Gateway service to issue a query.

The gateway is designed as a modular unit to allow the user to dynamically add new components such as language parsers or query optimizers even at runtime. The service-oriented architecture also allows to use different optimizers and translation workflows for different queries or users. Like the SwissQM virtual machine the gateway was also designed as an extensible research platform that can be used for a wide range of data-centric applications for wireless sensor network.

3.3.1 Architecture

The architecture of the query front end and the basic query processing workflow is shown Figure 3.8. A user query or a user-defined function (UDF) is submitted by a client through the Gateway Service. SwissQM provides two query interfaces the user can choose: SQL and XQuery. The queries and the UDFs are parsed and

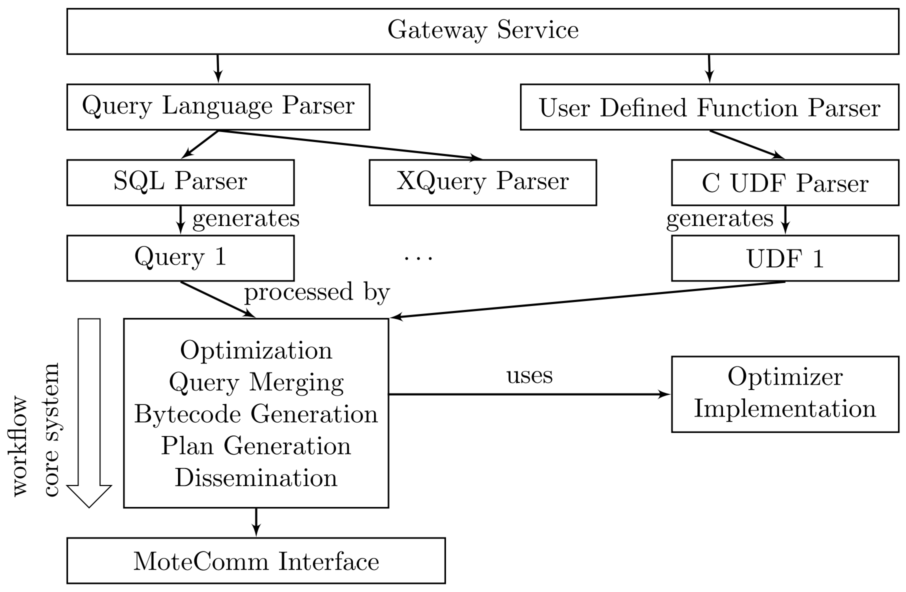


Figure 3.8: Architecture of the SwissQM/Gateway

transformed into a language-independent intermediate representation. OSGi bundles are generated out of the user queries and functions. The bundles are generated through the *ASM Java bytecode manipulation framework* [BLC02]. Wrapping both queries and UDFs into bundles allows the user to make direct use of the life cycle management provided by OSGi.

This translation process performs both merging and rewriting of queries as described in Chapter 4. Virtual queries are finally mapped into network execution plans, which are then disseminated into network in form of bytecode programs through the TinyOS *Mote Communication Interface*. Virtual queries can be reassigned to different network queries, thus, allowing for online changes in the parameter set of the query.

3.3.2 Query Interfaces

The gateway system supports two different query interfaces: SQL and XQuery. Users can submit queries via a webserver-based user interface or through the Eclipse IDE running the SwissQM plug-in. A client application such as a smart home application (Mueller et al. [MRDA07]) can submit a query by invoking the Gateway Service through R-OSGi.

SQL Interface. SwissQM provides the same SQL-like query interface as TinyDB. Queries have the following format.

```
  SELECT  {attributes, aggregates}
    FROM  sensors
   WHERE  {predicate}
GROUP BY  {attributes}
  HAVING  {predicate}
   EVERY  time span e
```

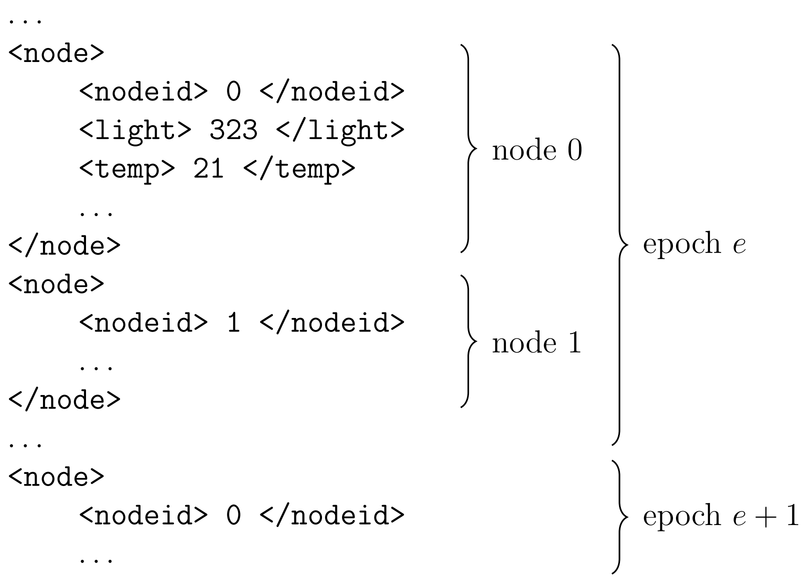
The select-clause can contain sensor attributes and spatial aggregates, but unlike TinyDB it supports general expressions and even user-defined functions. User defined functions have to previously submitted registered in the gateway system. SwissQM supports both spatial and temporal aggregation. Spatial aggregates are evaluated every sampling period (*epoch*). They aggregate data across nodes (spatially). The supported aggregation functions are shown in Table 3.3 (page 54). Grouping of aggregates is also supported through the *group by* clause. Temporal aggregation is provided by additional functions that can be used in the *select* clause. For *sliding windows* the function `SWINDOW`(*expr*, *length*, *aggop*) and for *tumbling windows* `TWINDOW`(*expr*, *length*, *aggop*) are available. *expr* refers to expression whose value is to be aggregated. *length* determines the size of the window. This parameter can be specified in time units or number of tuples. In the context of fixed-interval samples such as in SwissQM, time-based and tuple-based windows are equivalent. Sliding windows are evaluated for each sampled tuple whereas tumbling windows are evaluated every *length* tuples or time. *aggop* determines the evaluator function of the window. SwissQM supports `AVG`, `COUNT`, `MAX`, `MIN`, `VARIANCE`, and `SUM`.

XQuery Interface. The XQuery language accepts XQuery FLWOR expressions. The XQuery front end is based on the XML schema illustrated in Figure 3.9. The concept of sampling is introduced by a function `qm:sample()`. The following example shows a SQL query and the corresponding FLWOR query.

```
SELECT  nodeid
  FROM  sensors
 WHERE  light < 100
 EVERY  4 s
```

```
for $n in qm:sample('4s')/nodeid
where $n/light lt 100
return $n/nodeid
```

Submitted UDFs can be used in both XQuery and SQL queries. In XQuery they are mapped into the `udf:` space. The following example shows spatial aggregation



```
...
<node>
    <nodeid> 0 </nodeid>
    <light> 323 </light>
    <temp> 21 </temp>
    ...
</node>
<node>
    <nodeid> 1 </nodeid>
    ...
</node>
...
<node>
    <nodeid> 0 </nodeid>
    ...
```

Figure 3.9: XML Schema for XQuery front end

query with a UDF *ewma*.

```
  SELECT  parent, AVG(ewma(light)), COUNT(*)
    FROM  sensors
GROUP BY  parent
   EVERY  5 s
```

The corresponding FLWOR query is a bit more verbose. It uses an element constructor in the *return* clause that will explicitly generate a result tuple that contains the group value and the two aggregates.

```
for $n in qm:sample('5s')/nodeid
group $n as $siblings by $n/parent as $parent
return
   <tuple>
      <parent>$parent</parent>
      <avg>{udf:ewma($siblings/light)}</avg>
      <count>{count($siblings)}</count>
   </tuple>
```

3.3.3 User Defined Functions

SwissQM provides an additional language interface for user-defined functions. After a UDF is submitted it is parsed, translated into a bundle, and registered. It can then be used in any query (in SQL or the XQuery front end). The SwissQM C language interface supports a subset of C. The implementation allows only one single data type `int` that maps to the signed 16-bit data type of the SwissQM virtual machine. Due to the lack of subroutine calls (there is no such instruction in SwissQM) recursion in UDFs is not allowed. Non-recursive function calls are inlined by the compiler. Functions can have side effects, i.e., external state. UDF modules can make use of this through global variables and the `static` keyword inside a function. The following example UDF implements the exponential weighted moving average EWMA filter used in the previous example query.

```
int ewma(int u) {
   static int y1 = 0;
   int y;
   y = (4*y1+u)/5;
   y1 = y;
   return y;
}
```

State that is maintained over multiple invocations, such as the variable `y1` in the example, is allocated on the synopsis buffer if the UDF is pushed into the network. This requires exclusive access to the synopsis (the such called "raw" mode), therefore the synopsis cannot be used otherwise at the same time, e.g., for aggregation. The gateway then decides which of the operation is pushed into the network and which is executed at the base station. The expression `AVG`(ewma(*light*)) from the previous example query would use the synopsis for the both aggregation and the UDF. In this case, the gateway decides to push the UDF into the network and perform the aggregation operation in the stream engine at the gateway.

3.3.4 Query Life Cycle Management

Mapping both queries and functions into bundles has the advantage that they can benefit from the life cycle management provided by the OSGi container. Life cycle management is particularly important for long running continuous queries SwissQM was designed for. It is, for instance, possible to temporarily disable a query and restart it after some time or even replace an existing query with a new one. As an example, a query can be updated with a more selective one or by one making use of more sophisticated filtering of the sensor data. This has the advantage over submitting a new query that this modification is transparent to the client that receives the data stream. [3] This extensive query management

[3]Should the schema of the result stream be changed by a query update, the client of course is also affected and the update is no longer transparent.

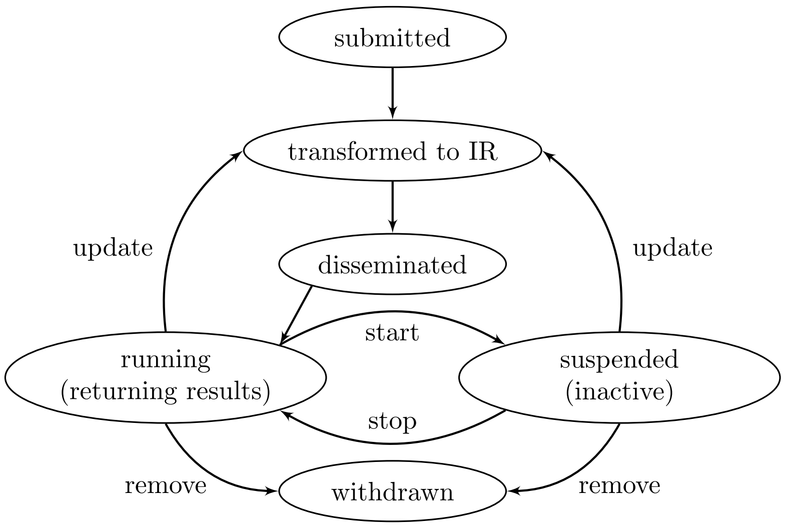


Figure 3.10: Life cycle of a Query

functionality can be implemented due to strict decoupling of network operations from queries.

Figure 3.10 shows the life cycle of a query. A query update can be initiated on a running or a suspended query. The client can update the query by modifying the query string. The query is parsed again and the changes are committed to the network and the execution platform. Exposing the life cycles has a further advantage: the gateway configuration including the queries can be made persistent. SwissQM runs on the Concierge OSGi container [RA07], which persistently stores the bundles. For example, when the SwissQM/Gateway is shutdown all query and UDF bundles in the system are stored onto persistent storage. After are restart of the gateway the bundles are loaded and can be resumed by the user. For the sensor network, the gateway will treat this operation equivalent to a query update that results in a bytecode program that is sent to the nodes.

3.4 Examples

In this section we present a few more complex use cases for the SwissQM virtual machine and illustrate how high-level queries and programs are translated into bytecode sequence by the SwissQM/Gateway.

3.4.1 Conventional Streaming Queries

The following query is specified in a streaming variant of SQL and is used to analyze the correlation between temperature and brightness (light) readings. Assume that the light sensor produces only unprocessed *raw* values from the A/D converter. Further assume that the light sensor has an offset reading that needs to be accounted for. The goal of the query is to average temperature readings of sensor nodes that have similar brightness readings. The groups are built by first removing the offset and then forming bins by applying *integer division*.

$$\begin{array}{rl} \texttt{SELECT} & (light - 512)/10, \texttt{AVG}(temp) \\ \texttt{FROM} & sensors \\ \texttt{GROUP BY} & (light - 512)/10 \\ \texttt{EVERY} & 30\,\mathrm{s} \end{array}$$

This simple query illustrates very well key differences between SwissQM and TinyDB. The query contains expressions to be computed as part of the query evaluation. TinyDB does not encode complete expression trees. Instead, all expressions must match a fixed format of the form:

$$\begin{array}{rl} & \langle attribute \rangle \\ | & \langle aggregate \rangle \big(\langle attribute \rangle \, \langle operation \rangle \, \langle constant \rangle \big) \end{array}$$

This limits the range of expressions supported by the current version of TinyDB. In SwissQM, the query can easily support arbitrarily complex expressions (within the inherent limits of memory available to QM programs) since we do not impose a hard-coded expression format. In order to compare with TinyDB, we simplified the grouping expression from $(light - 512)/10$ to *light*. TinyDB then uses three query messages (120 bytes in total) to disseminate the query. In our case the corresponding QM program requires only 20 bytes (two fragment messages). The bytecode for the original query is as follows:

```
.section delivery, "@30s"
   get_light
   ipushw     512
   isub
   ipushb     10
   idiv
   istore     0      # store group expression
   get_temp
   istore     1      # sum := temp
   iconst_1
   istore     2      # count := 1
```

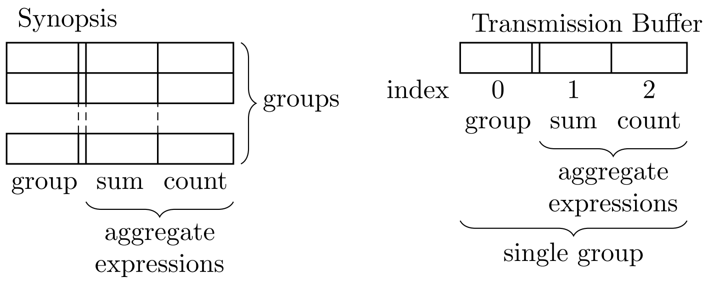


Figure 3.11: Layout of synopsis and transmission buffer (during delivery section) for the grouping aggregation query

```
   ipushb      5       # agg: AVG = 5
   iconst_1            # number of agg expr: 1
   iconst_1            # number of grp expr: 1
   merge
   send_sy

.section reception
   ipushb      5       # agg: AVG = 5
   iconst_1            # number of agg expr: 1
   iconst_1            # number of grp expr: 1
   merge
```

Figure 3.11 shows the layout of the synopsis for this example. A group is identified by the value of the grouping expression. The aggregation state for `AVG` consists of a sum/count pair. Figure 3.11 also shows the layout of the transmission buffer as is used in *delivery* section. The aggregate state contributed by the own sensors is prepared and merged to the local synopsis in the delivery section. When the reception section is executed the transmission buffer contains the partial aggregation state received from the descendant nodes. The size of the program including the complex grouping expression is only 27 bytes (two fragment messages).

3.4.2 In-Network Event Generation

Assume that a sensor network deployed in a building is used to detect potential fire related alarms. It is reasonable to assume that the presence of a fire is correlated with a sudden increase in temperature. In the following example the IDs of the nodes whose temperature reading increased by more than 10 % during the last 10 minutes are returned. The corresponding query is:

```
SELECT s.nodeid
  FROM sensors AS s,
          (SELECT nodeid, SWINDOW(temp, 10 min, MIN) AS mintemp
              FROM sensors
          GROUP BY nodeid
             EVERY 2 min)  AS w
 WHERE s.temp > 1.1*w.mintemp AND s.nodeid = w.nodeid
 EVERY 2 min
```

In the inner query, a sliding-window of the temperature reading is created for every node. The query returns the smallest temperature values observed during the last 10 minutes for every node. These values are then compared with the current temperature reading in the outer query. First, observe that the window is maintained locally for each node, i.e., no aggregation state needs to be exchanged. Second, in contrast to a centralized approach where the event detection is done at the gateway, a message needs to be sent to the gateway only if the predicate in the where-clause is satisfied.

The QM program implementing this query uses a window that contains the last five temperature samples. This window is advanced every two minutes, resulting in a total window width of 10 minutes. The window entries are stored in a five element ring buffer. The buffer is implemented as an array and an index `next_insert` that is used to determine where to insert the next element. Figure 3.12 shows the layout and the initial state of the synopsis where the ring buffer is stored. The array occupies positions 0–4, the `next_insert` index position 5. The bytecode listing of the corresponding QM program is shown below.

The synopsis is initialized in the *init* section. Initially, the ring buffer elements are set to 32767. This is the largest positive value that can be represented using a 16 bit signed integer type.

```
# initialize synopsis
# set syn[0..4]:=MAX (0x7fff)
.section init
    ipushb    4       # i := 4
l1: dup
    iflt      l2      # exit if i<0
    dup
    ipushw    0x7fff  # push MAX
    swap
    istore_sy         # syn[i] := MAX
    idec              # i := i-1
    goto      l1
l2: pop
    iconst_0            # next_insert := 0
    istore_sy 5

.section delivery, "@2min","manualclear"
# find min(syn[i], i=0..N-1)
    iload_sy  4         # establish invariant
    ipushb    4         # setup variant

 # loop variant i on top of stack, invariant below
 # stack content: min(syn[j],j=i..4), i
l3: dup
    iflt      l5        # exit loop if i<0
    dup_x1              # → i,x,i
    iload_sy            # → i,x,syn[i]
    dup2                # → i,x,syn[i],x,syn[i]
    if_icmplt l4        # → i,x,syn[i] jump if x<syn[i]
    swap                # → i, min(x,syn[i]),max(x,syn[i])
l4: pop                 # → i,min(x,syn[i])
    swap                # → min(x,syn[i]),i
    idec                # i := i-1
    goto      l3
l5: pop
# stack content at the end of loop: min(syn[i],i=0..4)

# insert new element into window
    get_temp            # read new sensorvalue
    dup                 # → min_temp,temp,temp
    iload_sy  5         # get next_insert
    istore_sy           # syn[next_insert] := temp

# advance next_insert
    iload_sy  5         # get next_insert
    iinc                # next_insert := next_insert+1
    ipushb    5
    irem
    istore_sy 5         # next_insert := next_insert%5

# current stack content: .., min,temp
```

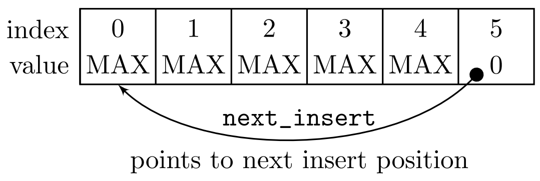


Figure 3.12: Layout and initial state of synopsis for “in-network event generation” example

```
52      swap                # →temp,min
53      dup                 # → temp,min,min
54      ipushb       10
55      idiv                # → temp,min,min/10
56      iadd                # → temp,min+min/10
57      if_icmple    l6     # skip if temp≤min+min/10
58      get_nodeid          # read nodeid
59      istore       0
60      send_tb             # send nodeid
61  l6:
```

The *reception* section is invoked every two minutes, i.e., when the window needs to be advanced. Then the minimum value of the ring buffer elements is determined (lines 18–35). Then the temperature sensor is sampled and the value obtained stored at the next insert position of the synopsis (lines 39–42). Afterwards, the next insert position is advanced (lines 45–49). Lines 52–57 evaluate the predicate $temp > 1.1 \cdot mintemp$. If the predicate evaluates to true, an event is detected and the node sends its ID to the root.

The size of the bytecode is 62 bytes, which can be disseminated in 3 fragment messages. The execution of the *delivery* section takes 40 ms on the Mica2 implementation. The *reception* section is empty as data does not need to be aggregated between nodes.

3.4.3 Adaptive Sampling

In principle, it is sufficient if sensor nodes only send data when the observed physical phenomena change. This allows answering queries directly at the gateway rather than fetching every tuple from the network. A similar idea has already been proposed by Deshpande et al. in the BBQ system [DGM$^+$05] where a statistical model is run at the gateway. Whenever the prediction of the model does not reach the confidence level specified in the query, the gateway actively requests additional data from the sensors in order to update the model parameters and thus increase the quality of the prediction. In BBQ, the model at the gateway decides when to acquire additional data. As an extension of this idea, one can consider running a replica—or at least a simpler, less complex model—at the sensor nodes that allows the node to decide on its own when the model at the gateway is outdated and requires new data to update its parameters. By moving from a pull-based to a push-based mode of operation, the number of messages can further be reduced since no explicit requests have to be sent.

We illustrate how a trivial strategy for *adaptive sampling* can be implemented in SwissQM. The idea is very simple: increase the sampling rate if the phenomenon changes rapidly, otherwise decrease the sampling rate. The following C code snippet illustrates how to implement adaptive sampling.

```
 1 int period = 0;
 2 int count  = 0;
 3 int oldval = getlight();
 4
 5 executeEvery2min() {
 6    int newval;
 7    if (count == 0) {
 8       newval = getlight();
 9   if (abs(newval-oldval)>oldval/10) {
10          // change > threshold → decrease period
11          if (period > 0) {
12             // don't change period if already fastest
13             period = period - 1;
14          }
15       } else {
16          period = period + 1; // increase period
17       }
18       oldval = newval;
19       send(getnodeid(),newval);
20       count = period;
21    } else {
22       // no sampling required
23       count = count - 1;
24    }
25 }
```

As the pseudo code shows, the program applies adaptive sampling on the light sensor. The routine `executeEvery2min` (line 5) is scheduled every two minutes, which determines the shortest possible sampling period. For larger sampling periods a variable `count` is introduced. When the routine is scheduled the variable is decremented (line 23). Sampling is only done when `count` reaches zero (line 8). Thus, all sampling periods are multiples of 2 minutes. The initial value assigned to `count` is the current sampling period stored in `period`. This period is increased or decreased depending on the difference between two consecutive samples `newval` and `oldval`. When the difference is larger than $\pm 10\%$ the `period` will be decremented (line 9 in the pseudo code). Otherwise the period is increased. After each sample the node sends its ID and the sensor value to the root (line 19). The C code snipped can be translated into the following bytecode program.

```
# initialize synopsis
.section init
    iconst_0
    istore_sy 0   # period := 0
    iconst_0
    istore_sy 1   # count := 0
    get_light
    istore_sy 2   # oldval := getlight();

.section delivery, "@2min", "manualclear"
    iload_sy  1   # load count
    ifneq     l1  # jump if count ≠ 0
    get_light     # → light
    dup           # → light,light
    iload_sy  2   # → light,light,oldval
    isub          # → light,light-oldval
    dup           # → light,light-oldval,light-oldval
    ifge      l3  # skip ineg if ≥ 0
    ineg          # → light,abs(light-oldval)

l3: iload_sy  2   # → light,abs(light-oldval),oldval
    ipushb    10  # → light,abs(light-oldval),oldval,10
    idiv          # → light,abs(light-oldval),oldval/10
    if_icmple l4  # within limit then increase period
    iload_sy  0   # → light,period
    dup           # → light,period,period
    ifeq      l5  # skip if already fastest
    idec          # decrement period
    dup           # → light,period-1,period-1
    istore_sy 0   # period := period-1
    goto      l5
l4: iload_sy  0  # increment period
    iinc
    dup          # → light,period,period
    istore_sy 0  # period := period+1

    # stack content: light,period
l5: istore_sy 1  # count := period
    dup          # → light,light
    istore_sy 2  # oldval := light

    istore    1  # send nodeid and light
    get_nodeid
    istore    0
    send_tb
    goto      l2 # go to end of section

l1: iload_sy  1  # sampling skipped
    idec         # count := count-1
    istore_sy 1
l2:
```

The length of the resulting QM program is 64 bytes. It can be sent using three program messages. In the bytecode, the state kept by the program is stored in the three global variables, which are placed in the synopsis: `period` at position 0, `count` at position 1, and `oldval` at position 2. The initialization is done in the *init* section.

3.5 Evaluation

3.5.1 Execution Time

Table 3.2 (page 46) lists the average execution time of each instruction implemented in Mica2 configuration of SwissQM. The execution time of each instruction was measured by instrumenting the SwissQM application code and set of micro benchmarks. For each bytecode instruction executed, the instrumented code sets an I/O register at the start and at the end. Setting a bit in an I/O register takes 2 clock cycles (i.e., 270 ns) on the ATmega128 micro-controller. Thus, the overhead of the instrumentation (which is included in the measurements) is small compared to the execution time of the actual bytecode instruction. The execution duration was measured as the resulting pulse length of the signal observed with a digital oscilloscope connected to the I/O port.

As shown in Table 3.2, most instructions are in the order of tens of microseconds. Exceptions are instructions for sending messages and certain instructions for reading sensor data. Both of these kinds of instructions are in the order of milliseconds. The execution times for the `send_tb` instruction, which sends the contents of the transmission buffer to the network, and for the `send_sy` instruction, which sends the synopsis, depend on the amount of data that needs to be sent. Furthermore, since a contention-based MAC layer for the radio communication is used, additional random delay is introduced. For example, we measured 22–42 ms for sending a full transmission buffer (`send_tb` instruction). The execution time for sensor sampling instructions depends on the type of sensor. For the sensors on the *Mica sensor board*, the execution times for the sampling instructions vary between 0.3 and 3.4 ms. Reading the light and temperature sensors uses a single A/D conversion and requires about 0.3 ms. Accessing the microphone takes longer (3.4 ms) because the A/D converter is accessed 10 times and the largest value obtained is returned. Reading system attributes (e.g., `get_nodeid` and `get_parent`) is faster and in the order of microseconds because no physical sensors need to be accessed.

This evaluation shows that even though SwissQM uses an interpreting VM instead of just-in-time compilation (JIT) the performance penalty is not significant. A frequently occurring stack manipulation instruction such `dup` requires 109 cycles CPU cycles, which is acceptable for embedded computing. Computationally more intensive operations, however, can be added natively as application-specific instructions such that the price of the interpreter overhead does not have to be paid. The approach chosen in the SwissQM makes use of the lower complexity of an interpreting VM and provides solution to mitigate the effects on performance by an extensible instruction set.

3.5.2 Concurrent programs

A direct consequence of the optimized use of resources and the use of a well characterized bytecode to represent programs is that SwissQM can support a higher level of multi-programming than more traditional platforms for programming sensor networks. As part of the evaluation of the platform, we have performed an experiment in which six different programs are concurrently run on a single node. The experiment is performed on a single Mica2 mote. The purpose is to investigate whether the six moderately complex programs can run concurrently on a mote. The space requirements for the six programs can be easily determined by analyzing the bytecode. In this experiment the timing aspect is analyzed.

The programs are characterized in Table 3.4; l_k represents a light reading, t_k

Table 3.4: Six QM programs used for multi-program execution experiment

	Description	Function	Period	Size	#Msgs
p_0	raw light	$y_k = l_k$	2 s	4 B	1
p_1	raw temperature	$y_k = t_l$	2 s	4 B	1
p_2	arith. operation	$y_k = \frac{1}{2}(l_k + t_k)$	4 s	8 B	1
p_3	max. temperature	$y_k = \max_{i=0,\ldots,i} t_i$	4 s	15 B	1
p_4	EWMA filter	$y_k = \alpha y_{k-1} + (1-\alpha)l_k$	4 s	16 B	1
p_5	window average	$y_k = \frac{1}{5}\sum_{i=0}^{4} t_{k-i}$	8 s	46 B	3

a temperature reading. Each program computes a different function of the light and temperature readings. Each program uses a different sampling period, but all programs produce a single value y_k for each epoch k. Although it is possible to compute the results for $p_2, \ldots, p_5$ from the results returned p_0 and p_1 all six programs are run by the motes. The redundancy in the data is used to verify the results.

The experiment is performed on a single Mica2 mote. The experiment ran for four hours in the afternoon until dusk in order to capture a drop in temperature and light. A lamp was turned on and off during two intervals in order to produce clearly identifiable changes in light and temperature. Figure 3.13 shows the data obtained by programs p_0, p_1, p_2, and p_3. The results produced by programs p_4 and p_5 are similar to those of p_0; these results are shown in Figure 3.14 (using a different scale to highlight the differences). The results obtained show 100 % coverage of the data. Note that a since single node is used that is directly connected to the gateway the results are not transmitted over the radio. The reliability of the serial connection is high enough such that packet loss can be ignored. The full coverage indicates that the *delivery* sections of the programs could be scheduled at the right moment. If a program delays execution it prevents running another program, which in the end results in missing tuples.

This experiment also illustrates several important advantages of SwissQM. First, the curves confirm observations made by Jeffery et al. [JAF+06]: There is a great deal of noise in sensor data. This noise can be seen in the graph for p_0 in Figure 3.14. This noise can be reduced by applying filter functions such as p_4 and p_5. The efficient in-network implementation of these functions motivates the design of a powerful and flexible platform for programming sensor networks such as SwissQM. Programs that just capture and send sensor data are not enough. In practice, the need for keeping state and specifying complex control flow in a program is even larger.

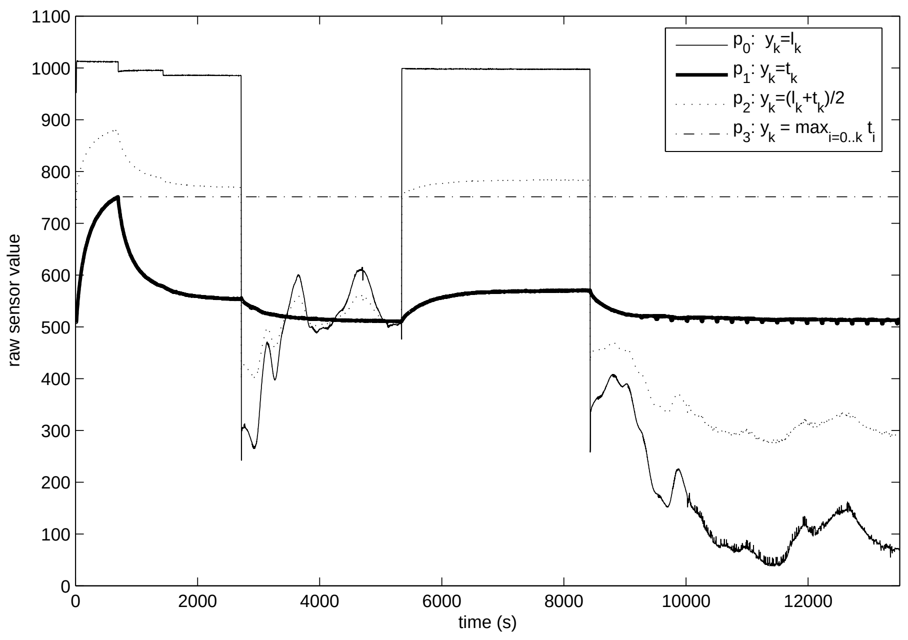


Figure 3.13: Complete results for p_0, p_1, p_2 and p_3

The second advantage illustrated by this experiment is the ability of SwissQM to run programs concurrently. The implementation of the six programs in this experiment was carried out manually (not using a high-level language) and purposefully in a naïve way. As a result, the execution is extremely inefficient: Every program samples the light and temperature sensors separately; the readings are not shared between the programs. The gateway would generate one single program sames sample both sensors every two seconds and compile much more efficient bytecode that allows the sharing of the sensor data in all programs. Nevertheless, despite these inefficiencies, the six programs run perfectly well and without interfering with each other, which shows the robustness of the SwissQM engine.

3.5.3 Program Distribution

This section analyzes the program distribution in a large scale network. In order to perform this experiment under controlled conditions as well as due to the lack of such a large number of sensor nodes, the experiment is run in the *TinyOS Simulator* (TOSSIM) [LLWC03] rather than on real Mica2 nodes. The simulator is run on a standard Intel-based PC. The advantage of TOSSIM and the nesC compiler is that the same software running on the Mica2 nodes can be executed natively on the platform that runs the simulator. TOSSIM is able to accurately

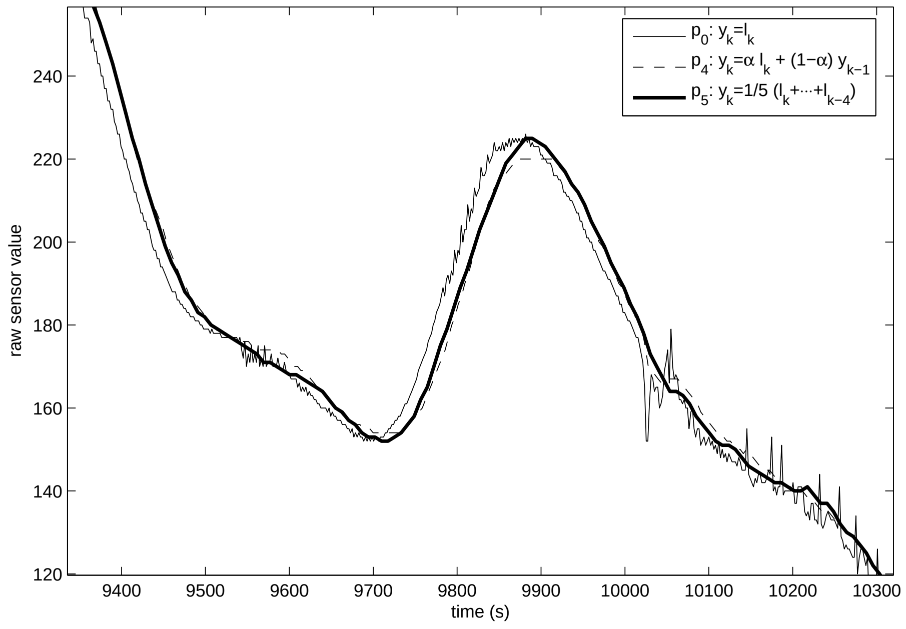


Figure 3.14: Detailed zoom for programs p_0, p_4 and p_5

simulate the network behavior of many nodes. The simulator uses a probabilistic bit error model for the radio links.

The experiment consists of a 10×10 grid of 100 sensors with a 4.5 m spacing between the nodes. The average transmission range is set to 15 m. The bit error rates used in the radio links are computed based on empirical data from a study by Ganesan et al. [GKW$^+$02] and the distance between two nodes. The root node is located in one corner of the grid and injects a program into the network. Figure 3.15 shows gird layout and the program dissemination starting from the lower right corner at different time instants.

In this experiment, we measured how long it takes to distribute a program that fits into a single message. We ran a statistically significant number of experiments and picked the best and worst case. The results are shown in Figure 3.16, which shows the percentage of nodes that received the program as a function of time. The program was disseminated using one single fragment message. In the best case, all nodes get the program in about 5 s. In the worst case, only 2 % of the nodes (the root node and one of its neighbors) successfully received the program during the initial flooding phase. All other nodes must resort to the snooping protocol in order to recognize that they have missed the program and then request it from other nodes. For the program in Figure 3.16 clearly shows the steps in the network coverage after every 5 seconds, i.e., when a result message is generated,

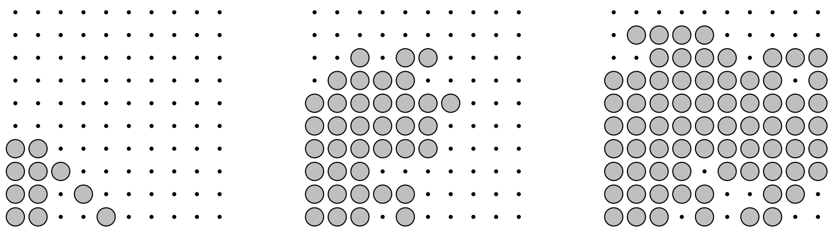

(a) $t = 0.5\,\mathrm{s}$ (b) $t = 1.5\,\mathrm{s}$ (c) $t = 3.0\,\mathrm{s}$

Figure 3.15: Snapshots taken during program dissemination show which nodes have received the program

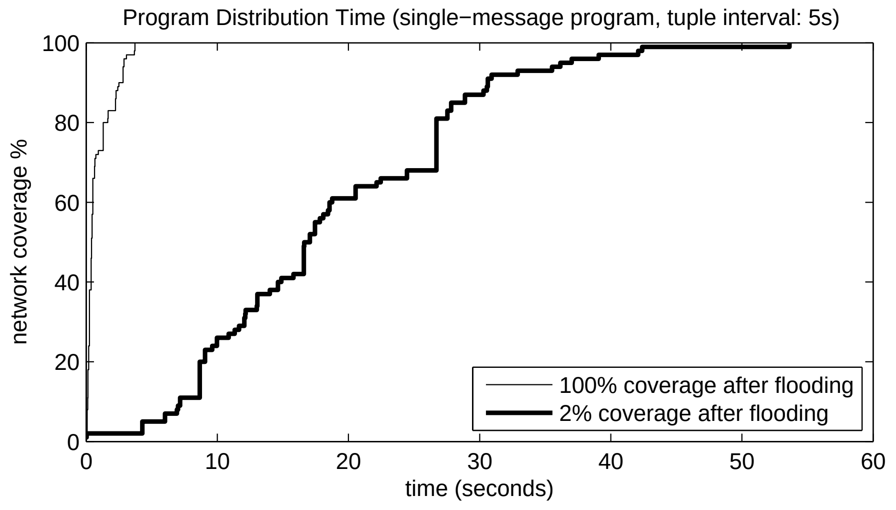


Figure 3.16: Program coverage during program distribution over time

nodes overhear it and realize that they have not received the program. In the worst case, 100 % coverage is reached after 53.6 seconds.

Flooding is an aggressive method to distribute data in networks. A slower approach to information distribution is *Trickle* by Levis et al. [LPCS04]. Trickle was developed for code dissemination in wireless sensor networks. It uses "polite gossip" where nodes periodically broadcast a summary, e.g., of the programs they are currently running, to local neighbors. When a node hears an older summary it initiates a local broadcast and sends an update. Gossip-based approaches typically trade-in latency against transmission cost. Flooding in SwissQM also provides a method for synchronization the query execution.

3.5.4 Comparison with TinyDB

These experiments shown in the previous section illustrate a well known limitation of ad-hoc wireless networks: the difficulty of achieving reliable communication. This lack of reliability is the reason why SwissQM puts so much emphasis on compact bytecode. If the results are so poor for a single message, distribution of a program that requires 2, 4, or even more messages is quite challenging. In fact, in our experiments such programs almost never reached 100 % coverage and often reach only a small fraction of the network. Note that this is a problem of the network. SwissQM (or other platforms) cannot solve it, it can only minimize its effects by minimizing the bytecode and running protocols that are aware of such failures.

To illustrate the advantages of a design like SwissQM over existing systems, we compared the program sizes of TinyDB and SwissQM. The interesting observation about this comparison is that TinyDB, unlike SwissQM, uses a (SQL-like) declarative representation of programs; in theory, a declarative representation should be more compact. In all our experiments, however, the optimized bytecode of SwissQM results in programs that are not only more expressive but also far more compact. TinyDB uses a rather verbose representation format. One message is used for each field and clause expression. Table 3.5 shows the sizes of several TinyDB queries and the sizes of the equivalent SwissQM programs. The queries shown cover the design space of TinyDB. Overall, SwissQM programs use less messages and less bytes than the corresponding TinyDB queries. In addition, TinyDB developers increased the TinyOS message size to 56 bytes in order to support larger packets. SwissQM uses 36 bytes (default on TinyOS). Hence, SwissQM not only uses less messages, but also smaller messages. Larger message sizes have the problem that they increase the message loss rate. Results would be worse for 56 byte messages, since the probability for packet corruption is larger.

At this point it also has to be noted that the current implementation of SwissQM lacks some features that are available in TinyDB. For example, TinyDB allows storing samples in a memory and then issue a query containing a join between the current sensor reading and the history data stored as a relation in memory. This functionality as well as access to flash memory, used to can store log data, can be provided as additional application-specific instructions. Similar to the `merge` instruction for spatial aggregation a `join` instruction can be provided that implements a join of the streaming data from the sensors and the relation stored in flash. In order to generate the corresponding bytecode programs, the query model of the gateway needs to be expanded accordingly.

Table 3.5: Number of messages (and bytes) transmitted for SwissQM programs and TinyDB queries generating the same data

Query	**TinyDB**		**SwissQM**	
	msgs	**bytes**	**msgs**	**bytes**
`SELECT nodeid,light,temp FROM sensors`	3	168	1	20
`SELECT nodeid,light FROM sensors WHERE temp<512 AND nodeid>50`	5	280	2	30
`SELECT MAX(light) FROM sensors`	2	112	1	22
`SELECT parent,MAX(light) FROM sensors GROUP BY parent`	3	168	1	25

3.6 Summary

In this chapter, the SwissQM virtual machine for sensor network was presented. The example queries demonstrated that SwissQM is a flexible execution platform for query processing in wireless sensor networks. The use of a bytecode interface increases the abstraction level at the network border. Furthermore, the expressiveness and, in fact, the Turing-completeness of SwissQM allows performing even complex task in-the network. As motivated in previous chapters the resource constrained sensor nodes alone cannot perform all the query processing work. Instead in the SwissQM approach, the work is shared between the more powerful gateway and the network nodes. The operations are split between the gateway and the nodes depending on the capabilities of the nodes and the communication costs. Having described the execution platform, the next chapter describes the query processor running at the gateway and how the execution platform is used in a cost-efficient manner.

4
Query Optimization for Sensor Networks

4.1 Motivation

Existing work on query processing for wireless sensor networks (WSNs) does consider the possibility of concurrently running more than one query in the system [YG02, MFHH05, GM04]. However, the majority of the deployments and much of the research work focuses on single application (single query) systems (e.g., [MCP+02, LBV06, JOW+02, TPS+05]). Interestingly, the current cost of a sensor network deployment is still very high and it is unlikely that it will significantly decrease in the near future. For instance, Langendoen et al. [LBV06] report on a 100+ node deployment in an agricultural project that requires one full year of planning and a budget in excess of €300,000. In such a deployment, economically, it makes sense to design the system so that it can efficiently support concurrent queries.

The challenge behind multi-user support lies in resolving the trade-off between efficient operation of the network (reduced traffic, sparse duty cycles at the sensors, avoiding redundancy in measurements and messages) and the number of independent user requests for data that need to be supported (each one interested in a potentially different set of sensors and acquisition rate). The difficulty with multi-query optimization is that while in many cases merging queries is a good strategy [MA06, XLTZ07], there are enough cases where alternatives like query splitting, and query rewriting are better options. In this chapter, the different alternatives and approaches are discussed. An important conclusion of this work is that multi-query optimization in sensor networks needs to include more than just query merging. We explore the design space for a query optimizer whose input are SQL-like queries and its output a set of execution plans (e.g., bytecode programs) to be run in a sensor network. Queries can be submitted and withdrawn at any time. We start the discussion of query optimization by using a simple cost metric on the message complexity, i.e., the number of messages transmitted. Later, we expand the discussion to a more general cost model that includes the overall energy consumption.

An additional complexity is the dynamic nature of query processing. Users can submit and withdraw queries at any time, hence, the query processing and optimization problem cannot solved upfront in a static manner. There is an inherent trade-off between an efficient execution of the individual queries and the costs required to account for changes in the query load. This leads to different optimization strategies, such as eager optimization that tries to minimize the total execution cost and lazy strategies that minimize the update cost after a change. In this chapter, these strategies are described and compared for different query workloads. In general, query optimization may use the following mechanisms:

Merging. Combining two or more queries into a single execution plan. Then extracting the results for each query from the combined data stream.

Splitting. Dividing a query into sub-queries so that the sub-queries can either be answered from already existing result data streams or better merged with other queries. Then reconstructing the result data stream from all the different pieces.

Parallelizing. If the overlap of queries is too small it can be better not to merge queries but to run them separately.

4.2 Data Model and Operators

The basic data model is a *virtual stream*, equivalent to that used in existing work by Madden et al. [MFHH05] and Yao et al. [YG02]. The stream contains tuples with time stamp indicating when the tuple was issued and a set of attributes. Traditional stream processing operators are then applied on the data stream.

4.2.1 Data Model

In response to a query, each node produces a data stream, i.e., a ordered sequence of tuples. The tuples contain the values that correspond to the expression listed in the *select* clause of the query. The attributes are typically raw sensor readings although more complex expressions such as aggregates are also possible. Each node periodically samples all sensors that are specified in a query and generates a tuple that is then processed as in a traditional stream processing engine. However, the combination with sampling gives rise to specific properties that cannot be observed in traditional stream processing systems. Essentially, sampling can be considered as filtering along the time axis. The model assumes that all sensors of a node requested in a query are sampled at virtually[1] the same time instant. The model further assumes that the clocks of the nodes are also synchronized such that samples taken on different sensor nodes can be correlated. Similar to TinyDB, the result tuples contain a time stamp, called *epoch*, that enumerates the tuple, and hence, the samples taken from each sensor on that node. The epoch attribute is added implicitly to the result stream even if it is not specified in the query. Additional non-sensor attributes can also be specified in queries. These attributes can be statically assigned values, such as the *nodeid*, a geographic location of an immobile node, a room number where node is deployed, etc. Additional dynamic attributes describe the state of a node, e.g., its current *parent* or *depth* in the routing tree, or the current battery *voltage* of a node.

Example. Consider the following query issued in sensor network, consisting of three nodes with *nodeid* 0, 1, and 2.

$$\begin{array}{rl}\texttt{SELECT} & \textit{nodeid},\ \textit{temp}, \textit{light} \\ \texttt{FROM} & \textit{sensors} \\ \texttt{WHERE} & \textit{temp} > 30\,^\circ\mathrm{C} \\ \texttt{EVERY} & 15\,\mathrm{s}\end{array}$$

In this non-aggregation query the nodes sample their light and temperature sensors every fifteen seconds. The predicate in the *where* clause makes sure that only tuples that contain temperature readings above 30 ℃ are present in the result stream. Assume that after sampling and before applying the predicate in the *where* clause, the stream shown in Figure 4.1(a) is generated. The model assumes that this virtual stream is received at some central location such as the gateway where the additional operators such as selection defined in the *where* clause is applied. In practice, however, the selection is pushed into the network in order to reduce the number of radio transmissions. Note that although the tuples are ordered by the epoch number, no ordering is assumed for tuples within an epoch. This is motivated by properties of the multi-hop tree routing. Tuples from nodes

[1]In practice is not possible to access multiple sensors at the same time, in particular, if they share the same bus (e.g., I^2C). However, the delay between samples of different sensors on a node is in the order of milliseconds. This is negligible compared to a sampling interval in the order of seconds as considered in this work.

materialized readings

epoch	*nodeid*	*temp*	*light*
0	2	23	165
0	1	33	235
0	0	25	173
1	1	29	215
1	2	24	165
1	0	24	129
2	2	25	166
2	1	31	218
2	0	23	165
⋮	⋮	⋮	⋮

(a) Raw sensor readings

result stream

epoch	*nodeid*	*temp*	*light*
0	1	33	235
2	1	31	218
⋮	⋮	⋮	⋮

(b) Result stream

Figure 4.1: Stream of raw sensor readings and result stream after applying *where* clause predicate

closer to the root arrive earlier than tuples from nodes further way. The sampling intervals are assume to be larger than the transmission latency to the root. This guarantees that tuples of a given epoch are received at the gateway when the next epoch starts. Figure 4.1(b) shows the result tuples, that satisfy the predicate in the *where* clause of the query, i.e., those with a temperature attribute above 30 ℃. □

Spatial Aggregation. The stream model also allows aggregation queries. One type of aggregation are *spatial aggregates* that *fuse* data from different sensors captured during the same epoch. The following query computes the maximum temperature sensed by any node during each epoch.

```
  SELECT epoch, MAX(temp)
    FROM sensors
GROUP BY epoch
   EVERY 15 s
```

The same query can also be rewritten without *group by*.

```
SELECT MAX(temp)
  FROM sensors
 EVERY 15 s
```

This deviates from the traditional SQL semantics. We allow this because (1) the virtual stream is assumed infinite, hence, the aggregation operator would block infinitely, (2) the epoch attribute is implicitly added to result stream, and (3) the query model does not allow direct correlation of tuples from different epochs.

Temporal Aggregation. Window-based aggregation can be used to combine tuples from different epochs. Two window modes are considered: *sliding windows* and *tumbling windows*. The windows are specified in a query using the functions SWINDOW for sliding windows and TWINDOW for tumbling windows. The query

```
      SELECT  nodeid, SWINDOW(light, 10, AVG)
        FROM  sensors
PARTITION BY  nodeid
       EVERY  15 s
```

computes the average light value over sliding window that contains the samples from last ten epochs. Note the *partition* clause splits up the virtual stream based on the *nodeid* attribute. This essentially generates substreams, which can be locally computed on the nodes. Since aggregation of locally generated data is often used in practice, e.g., for data cleaning purposes such as outlier removal, a shortcut notation is provided. The *partition-by-nodeid* clause can be dropped. Like in the case for spatial aggregation strictly speaking this notation is incorrect:

```
SELECT  nodeid, SWINDOW(light, 10, AVG)
  FROM  sensors
 EVERY  15 s .
```

4.2.2 Query Operators

Sampling Operator τ. The sampling operator τ is a temporal operator that materializes the virtual stream at the specified sampling intervals. For example, $\tau_{15\,\mathrm{s}}()$ generates a tuple stream that contains readings of all sensors of every node sampled once every fifteen seconds.

Rate Conversion Operator ρ. The rate conversion operator ρ is a temporal operator that changes the sampling rate of a tuple stream. Here, we discuss only down-sampling of the tuple stream (increasing the rate of the tuple stream is possible by interpolation [SY07] or model based sampling [DM06]). Down-sampling in sensor networks is nontrivial due to inaccurate timers and jitter as we will show later in this chapter. Without loss of generality we can assume that down-sampling is implemented as a simple rate-based m:1 sampling, e.g., $\rho_{m:1}$ implies that only every m-th incoming tuple is forwarded.

Shift Operators ζ. The data streams coming from the sensors have time (*epoch*) information attached to them that relate to the network query. The time and epoch shift operators set the counters to 0 when a user query starts and transforms the epoch information of already existing data streams so that to the user it looks like the system just started producing data for that user query.

Projection π and Selection Operators σ. As with traditional query algebra, we use projection π and selection σ operators to remove sensor attributes and to apply filter predicates. There is one significant difference, though. In sensor networks data is generated by sampling and not read as records from disk. Thus, for energy reasons, the sampling operator and the first projection operator are always fused in the execution plan. Consider, for example, the query

$$\begin{array}{rl} \texttt{SELECT} & \textit{nodeid},\ \textit{temp} \\ \texttt{FROM} & \textit{sensors} \\ \texttt{EVERY} & 6\,\mathrm{s}\ , \end{array}$$

which can be represented as $\pi_{nodeid,temp}(\tau_{6\mathrm{s}}())$. The dynamics of selection and projection are different than in conventional query algebras. The reason is that selection and projection are not made over real attributes in a table but over sensors that need to be sampled. Hence, projection and sampling are fused such that only the required sensors need to be sampled.

Window Operator ω. In addition to the standard selection and projection operators, our approach also uses temporal operators that filter on the time axis. The window operator is a temporal operator. It can be implemented either as a *sliding window* (`SWINDOW` $\omega_{s,n}$) or a *tumbling window* (`TWINDOW` $\omega_{t,n}$), with n representing the size of the window. The supported operators over windows include `AVG`, `MAX`, `MEAN`, `MIN`, and `SUM`.

The use of windows in stream setup with an equidistant tuples as generated by the sampling operator time and tuple-based windows can be considered equivalently. Here, we treat them as tuple-based windows. The sliding window operator is evaluated for each incoming tuple whereas the tumbling window operator produces an output tuple only ever every n tuples. As an example, the following two queries

$$\begin{array}{rrl} u_1: & \texttt{SELECT} & \textit{nodeid},\ \texttt{SWINDOW}(\textit{light}, 10, \texttt{MAX}) \\ & \texttt{FROM} & \textit{sensors} \\ & \texttt{EVERY} & 5\,\mathrm{s} \\ \\ u_2: & \texttt{SELECT} & \textit{nodeid},\ \texttt{TWINDOW}(\textit{light}, 10, \texttt{MAX}) \\ & \texttt{FROM} & \textit{sensors} \\ & \texttt{EVERY} & 5\,\mathrm{s} \end{array}$$

can be represented as follows in the operator algebra:

$$u_1 : \pi\Big(\omega_{s,\,10,\,\texttt{MAX}}\big(\tau_{5\,\mathrm{s}}()\big)\Big) \qquad u_2 : \pi\Big(\omega_{t,\,10,\,\texttt{MAX}}\big(\tau_{5\,\mathrm{s}}()\big)\Big) \enspace .$$

The different evaluation strategies of the operators result generate result streams with different tuple rates. Tuples generated by $\omega_{s,\,10,\,\texttt{MAX}}(\tau_{5\,\mathrm{s}}())$ have an interval five seconds while for $\omega_{t,\,10,\,\texttt{MAX}}(\tau_{5\,\mathrm{s}}())$ a node emits a tuple only every 50 seconds.

Fusion Operator μ. The data fusion operator μ performs spatial aggregation, i.e., it aggregates data from several sensor nodes. However, it does not keep history state beyond the current epoch. The μ operator performs traditional in-network aggregation and can be implemented using the TAG approach [MFHH02]. The operator can compute one or more aggregates. As in the *group by* clause of traditional SQL, the tuples can be grouped by one or more attributes. In this work we consider the SQL aggregates `AVG`, `COUNT`, `MAX`, `MIN`, `STDDEV`, `SUM`, and `VARIANCE`.

Summary and Notation. Table 4.1 summarizes the notational conventions as needed for the discussion in this chapter. The operators are designated by Greek symbols. The symbols are defined as they are introduced in the following sections.

4.3 Query Merging

Query merging happens at the gateway of the system. Users can request data from the sensors by posing *User Queries* to the SwissQM/Gateway system, which then can merge multiple of these queries into one *network execution plan* that is sent as a bytecode program into the sensor network and executed by SwissQM virtual machine. Result tuples generated by the network execution plan are extracted tuples the network and fed to a stream processing network engine (see Figure 3.2 on (page 39) query to produce result tuples for all associated user queries.

Example. To illustrate how user queries are merged into a network query and how the result data is extracted, we use a simple example. Consider the following

Table 4.1: Notation summary of operators and symbols used in this chapter

symbol	description
ζ	shift operator
μ	fusion operator
π	projection operator
ρ	rate conversion operator
σ	selection operator
τ	sampling operator
ω_s	sliding window operator
ω_t	tumbling window operator
a	attribute (sensor on a node)
u	user query $u = (u_A, u_s, u_P)$
u_ρ, u_σ	set of stream processing operators for query u
p	execution plan $p = (p_A, p_s, p_P)$
P	predicate
s, u_s, p_s	sampling interval (seconds)
A, u_A, p_A	set of attributes $A = \{a_1, a_2, \ldots\}$
u_P, u_σ, p_P	set of predicates $\{P_1, P_2, \ldots, \}$ used as conjunction $P_1 \wedge P_2 \wedge \ldots$
u_ρ	rate conversion ratio, e.g., $n : 1$
$\mathcal{P}$	set of execution plans $\mathcal{P} = \{p_1, p_2, \ldots\}$
$\mathcal{U}$	set of user queries $\mathcal{U} = \{u_1, u_2, \ldots\}$
C_a	cost of sampling and transmitting attribute a
C_m	cost for sending the message header
$C_r(p)$	cost for removing plan p
$C_s(p)$	setup cost for plan p
C_T	tree topology parameter $C_T = N\bar{h}$
E	expression
$E@r$	annotated expression
$E_i@r\|P(E_j)$	annotated expression E_i with predicate $P(E_j)$
U	update set $U = \{E_1@r, E_2@r, \ldots\}$
$\bar{h}$	average tree depth
$m(p)$	number of program messages used to represent plan p
N	total number of nodes
P_x	power (Watt) spent for operation x
r	tuple rate ($^1/_s$)
$\bar{\sigma}$	average selectivity of a predicate
t_x	duration (seconds) of operation x
T	tuple

four user queries:

$$
\begin{aligned}
u_1 :\ & \texttt{SELECT}\ \ \textit{nodeid}, \textit{light} \\
& \texttt{FROM}\ \ \textit{sensors}\ \ \texttt{EVERY}\ \ 5\,\mathrm{s} \\
u_2 :\ & \texttt{SELECT}\ \ \textit{nodeid}, \textit{light} \\
& \texttt{FROM}\ \ \textit{sensors}\ \ \texttt{EVERY}\ \ 15\,\mathrm{s} \\
u_3 :\ & \texttt{SELECT}\ \ \textit{light} \\
& \texttt{FROM}\ \ \textit{sensors}\ \ \texttt{EVERY}\ \ 50\,\mathrm{s} \\
u_4 :\ & \texttt{SELECT}\ \ \textit{nodeid}, \textit{light}, \textit{temp} \\
& \texttt{FROM}\ \ \textit{sensors} \\
& \texttt{WHERE}\ \ \textit{nodeid} = 1\ \ \texttt{AND}\ \ \textit{temp} > 20\,^{\circ}\mathrm{C} \\
& \texttt{EVERY}\ \ 50\,\mathrm{s}\ \ .
\end{aligned}
$$

In spite of the fact that these queries are requesting different data with different sampling periods, they can be merged into one single network execution plan that generates the necessary data.

$$
\begin{aligned}
p_1 :\ & \texttt{SELECT}\ \ \textit{nodeid}, \textit{light}, \textit{temp} \\
& \texttt{FROM}\ \ \textit{sensors} \\
& \texttt{EVERY}\ \ 5\,\mathrm{s}
\end{aligned}
$$

This network query, will deliver data on *nodeid*, *light* and *temp* every 5 seconds. The stream returned by p_1 however cannot be provided as answer to the user queries. Not all stream operators have been applied yet. For each query a set of operators are configured in a *stream processing plan* that is executed at the gateway will apply the remaining operations. For u_1 and u_2 the *temp* attribute needs to be dropped. In user query u_2 a different sampling interval was specified. Hence, a rate conversion operator $\rho_{3:1}$ is inserted that only forwards one out of every three data points in order to enlarge the sampling interval to 15 seconds. For u_3, the light data must be extracted from one out of each 10 data points to produce light measurements every 50 seconds, i.e., the operator $\rho_{10:1}$ is added. u_4 receives data from all sensors but the predicate still needs to be applied. This can be done by inserting a selection operator $\sigma_{nodeid=1 \wedge temp>20\,^{\circ}\mathrm{C}}$. Before the selection operation, the rate of the stream is matched to u_4 by adding a rate conversion operator $\rho_{4:1}$ that selects every fourth tuple.

Assume that the queries are submitted in sequential order, i.e., u_1 is submitted first, u_4 is the last query. Further assume that only one network execution plan is run. Then the network execution plan needs to be replaced twice. Therefore, the epoch count of the tuples that are returned to the user need to be adjusted such that the first tuple of a new query has epoch number 0 even tough it is connected to an already running plan. This is achieved by adding shift operators ζ. Figure 4.2 shows the resulting operator plan that implements the post processing of the stream generated by p_1 required by four user queries. □

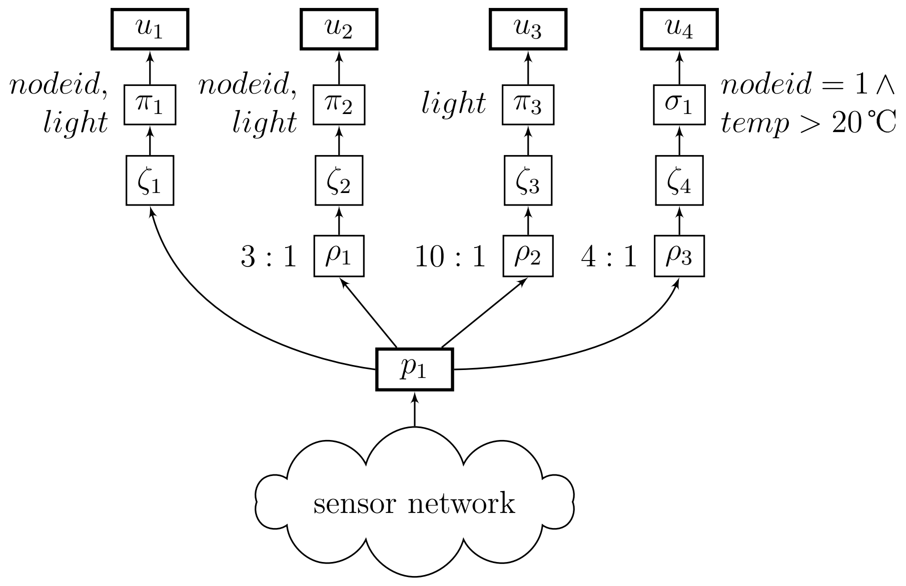


Figure 4.2: Stream execution plan implements post processing of result stream generated by the network execution plan

We now provide a more formal background for the mapping of multiple user queries into a single network execution plan. We will refer to the set of user queries as $\mathcal{U} = \{u_1, u_2, \ldots, u_m\}$. The network execution plan is denoted by p. Let A be a set of attributes $A = \{a_1, a_2, \ldots\}$ where a is a sensor that can be selected by a query. Then u_A is the set of attributes selected by user query u and, similarly, p_A the set of attributes returned by the network execution plan p. The sampling period used in the queries and the plan is u_s and p_s, respectively. We assume that predicates $P_1, P_2, \ldots$ appear in the *where* clause of a query in conjunctive normal form (CNF), e.g., $P_1 \wedge P_2 \wedge \ldots$. The reason for choosing CNF is that each predicate by itself can be considered as a filter stage. Furthermore, since predicates can be applied in any order we can collect them in a set $u_P = \{P_1, P_2, \ldots\}$ for a user query and similarly for a plan p_P. Hence, a query u can be represented as a triple (u_A, u_s, u_P) and a plan p as (p_A, p_s, p_P).

A first requirement to meet is that the set of attributes p_A of the network plan must be a superset of the attribute set u_A of any user query associated with plan p. This leads to the first condition that must hold if a set U of user queries is mapped to a plan p:

$$\forall u \in U \; : \; p_A \supseteq u_A \tag{4.1}$$

The next condition involves the sampling periods. The sampling period p_s must evenly divide the sampling interval u_s of all its user queries. This guarantees that the data stream provided by p can be used to answer all queries u associated with p, thus,

$$\forall u \in U : \exists k_u \in \mathbb{N} \, : \, u_s \, = k_u \cdot p_s \ . \tag{4.2}$$

One way to enforce this condition is to make p_s the greatest common divisor (GCD) of all user query sampling periods. Note that if the sampling intervals of two user queries are relative prime then $p_s = 1$, which is certainly undesirable. Also, in many cases, forcing an exact arithmetic match is too restrictive. Thus, instead of the above condition, we allow for a relative error in the sampling period up to some ε. Instead of using the GCD algorithm for determining the common sampling period we then use a *"Tolerant" Greatest Common Sampling period* (TGCS) such that for all user queries the effective sample period observed is within ε of what the user requested. The TGCS algorithm is described later in Section 4.3.4. Thus, instead of Equation (4.2) we require:

$$\forall u \in U : \exists k_u \in \mathbb{N} \, : \, (1-\varepsilon)u_s \leq k_u \cdot p_s \leq u_s \tag{4.3}$$

Note that in some cases, the application submitting the user query may request data to be delivered exactly at the specified time intervals. This can be achieved through operators that cache the result for a short period of time until it is time to send it to the user. Given the uncertainties in some of the measurements and the lack of precision of most sensor networks today, such time shifts in the measurements should be acceptable in most applications, specially if they can be constrained within a well specified error margin. Also, the formulation just provided adjusts towards the requested period since it is easier to cope with more data than with missing data. The implementation of rate conversion of the tuple stream is described in Section 4.3.4.

Selection queries—as implied by the predicate in the *where* clause—require special treatment. As explained above we require that they must be written in conjunctive normal form to emphasize the filtering property. For u_4 from the previous example we have $u_{4P} = \{\mathit{nodeid} = 1, \mathit{temp} > 20\,^{\circ}\mathrm{C}\}$ A query can be represented in general as a selection over a conjunction of predicates p_i, p_j followed by a projection on a set of attributes A from the set of sensor attributes u_A. So for any two queries u_i and u_j in the algebra:

$$\begin{aligned} u_i : &\quad \pi_{A_i}\left(\sigma_{P_{i_1}\wedge P_{i_2}\wedge\ldots\wedge P_{i_n}}\left(\tau_{u_{is}}()\right)\right) \\ u_j : &\quad \pi_{A_j}\left(\sigma_{P_{j_1}\wedge P_{j_2}\wedge\ldots\wedge P_{j_m}}\left(\tau_{u_{js}}()\right)\right) \ . \end{aligned}$$

In order to allow sharing of common operations, the selection predicates of any two queries must be brought in relation. We define the relation "u_j is at least as selective as u_i" as $u_i \preceq u_j$ where we use

$$u_i \preceq u_j \ := \ P_{i_1} \wedge P_{i_2} \wedge \ldots \wedge P_{i_m} \Rightarrow P_{j_1} \wedge P_{j_2} \wedge \ldots \wedge P_{j_n} \ .$$

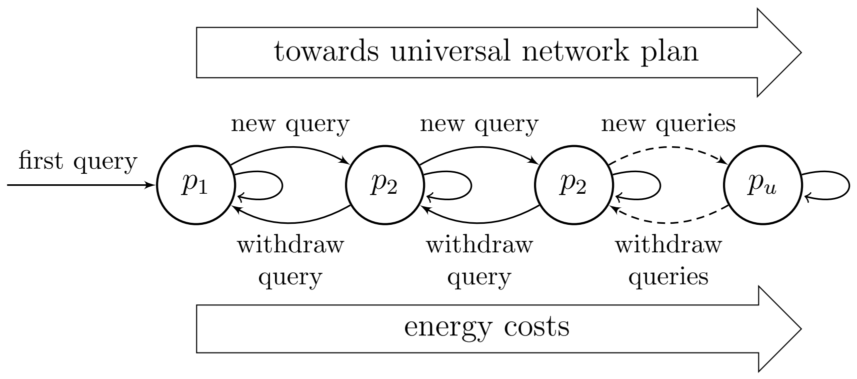


Figure 4.3: Moving between specific network plans and the "universal" network plan as user queries are submitted and withdrawn

This leads to the last condition that must be met by the network query. The network plan p must be "at least as selective" as any of its user queries u, i.e., $u \preceq p$, or in other words, return at least as many tuples as requested by any user query, thus,

$$\forall u \in U \; : \; u \preceq p \; . \tag{4.4}$$

Summarizing Equations (4.1), (4.3), and (4.4) we obtain the following rules for mapping a set of user queries to a network execution plan:

$$\forall u \in U \; : \; p_A \supseteq u_A \quad \wedge \quad u \preceq p \quad \wedge \quad \exists k_u \in \mathbb{N} \; : \; (1-\varepsilon)u_s \leq k_u \cdot p_s \leq u_s \tag{4.5}$$

4.3.1 Universal Network Execution Plan

The idea behind merging queries into a common execution plan arises from the following observation. There is always a limit to the maximum amount of data that can be obtained from a sensor network: sampling at the highest possible frequency, i.e., at the lowest possible interval, and capturing data from all the sensors of every node. We refer to a such request as the *Universal Query* as it materializes the complete virtual stream such that it contains all the data ever needed to answer any user query. The universal query

```
SELECT *
  FROM sensors
 EVERY ε
```

translates into the *universal network plan.* It has the highest possible *selectivity* since it returns all available data. We use ε to describe the shortest possible sampling interval that is supported. SwissQM schedules programs on a granularity of 1 millisecond. Hence, theoretically, we have $\varepsilon = 1\,\text{ms}$. In practice, however, the high network traffic that is generated at high data rates leads to a significant amount of loss as we explained earlier in Section 2.2.4 (page 23) and illustrated in Figure 2.5. Due to the low reliability the networks exhibit under heavy load, we set the minimum sampling period to one second. Operating the sensor network at such high rates without knowing whether the data is used at all is highly inefficient. Nevertheless, the intuitive notion of a universal query seems to indicate that it should be possible to merge a set of user queries into a single plan that will produce the data needed to answer all of the outstanding user queries.

In order to reduce the amount of data that flows through the network, the goal is to find the least selective plan that allows to answer all outstanding user queries. Thus, as new user queries arrive, the resulting network plan is progressively expanded (it is made more selective) according to the new user queries. Ultimately, the system could end up expanding to the universal network plan. Conversely, when user queries are withdrawn, the system must in turn decrease the selectivity of the network plan so that it captures only the necessary data. This process of increasing selectivity (or moving toward the universal plan as new user queries arrive) and decreasing selectivity as user queries are removed is shown in Figure 4.3. Performing such a process dynamically and in an efficient manner is the main challenge in SwissQM/Gateway.

4.3.2 Adding a new User Query

Condition (4.5) states when a network query can be used to answer a given user query. The challenge however is how to dynamically manage a network query as user queries arrive (later on we will discuss what to do when user queries are withdrawn). For the moment we assume in order to simplify the discussion that data acquisition system (SwissQM or TinyDB) has limited support for multiple concurrent execution plan. Further assume that we are forced to map all user queries to a single network plan. A second network plan is used during the transition phase, i.e., when a new user query is added or withdrawn, and the network query has to be replaced. The detailed procedure for adding a new user query is shown in Figure 4.4. On line 2, if no network query is running, the system makes the user query the network query. Otherwise the system checks if the network query that currently delivers data matches the condition (4.5) described above. If this is the case, the existing network plan p can be used to answer u. The query processor then adjust the rate conversion operator u_ρ in the stream execution plan for u and inserts selection operators for all predicates that are not part of the network plan p (line 8). In this case, the new user query can be accommodated without any change in the sensor network.

Now assume that the current network plan does not satisfy condition (4.5).

```
   input: new user query $u$
1  if no network plan running then
2      create new network plan $p$ with $p_A := u_A$;   $p_s := u_s$;   $p_P := u_P$ ;
3      inject plan $p$ into the network ;
4      set operators for queries $u_\rho := 1 : 1$;   $u_\sigma := \emptyset$ ;
5      $\mathcal{U} := \mathcal{U} \cup \{u\}$ ;
6  else
7      if $u_A \subseteq p_A \wedge u \preceq p \wedge \mathrm{TGCS}(u_s, p_s) = p_s$ then
8          $u_\rho := {}^{u_s}/_{p_s} : 1$;   $u_\sigma := u_P \backslash p_P$ ; // connect to existing plan
9          $\mathcal{U} := \mathcal{U} \cup \{u\}$ ;
10     else
           // setup new plan and migrate queries
11         create new network plan $p'$ ;
12         $p'_A := \bigcup_{u_j \in \mathcal{U}} u_{jA}$ ;
13         $p'_s := \mathrm{TGCS}(\{u\} \cup \mathcal{U})$ ;
14         $p'_P := \bigcap_{u_j \in \mathcal{U}} u_{jP}$ ;
15         inject $p'$ into the network;
16         $u_\rho := \frac{u_s}{p'_s} : 1$;   $u_\sigma := u_P \backslash p'_P$ ; // connect to new plan
17         wait until $p'$ has generated tuples ;
18         foreach $u_j \in \mathcal{U}$ do
19             $u_{j\rho} := {}^{u_{js}}/_{p'_p}$;   $u_{j\sigma} := u_{jP} \backslash p'_P$; // migrate query to new plan
20         $\mathcal{U} := \mathcal{U} \cup u$ ;
21         remove old plan $p$ ;
22         $p := p'$ ;
```

Figure 4.4: Adding a new user query

This happens if the sampling period of the plan does not match the one specified in the new user query, the new user query request additional sensor attributes or has fewer predicates. The system now has to replace the current network plan p by a new plan p' such that condition (4.5) is met (lines 11–14). The sampling period p'_s of new network query is determined by computing the "tolerant" greatest common sampling period on all user queries (including the new query u). The attributes and the selection predicates of p' are also chosen based on sets present in the user queries. The attribute set is the union of the attribute sets of all user queries and the selection predicate is the largest common set, i.e., the intersection set of all user query predicates. Next, the down-sampling and selection operators of the new user query u are configured for new network plan p', which is then injected into the network. It is immediately used to return tuples for user query u. While p' is being disseminated the old plan p continues to deliver tuples. As soon as p' has been set up (detected by counting the received tuples, line 17) the remaining user queries are migrated from the old plan p to p' (line 19). This requires reconfiguration of the rate conversion operators $u_{j\rho}$ to the new common sampling period. Additionally, the insertion of selection predicates into $u_{j\sigma}$ between the user query and the new plan p' might be necessary since the new network plan can be more selective. In order to simplify the discussion we do not explicitly list attribute projections for the queries. These would be necessary as the new attribute set p'_A can be larger than p_A. Afterwards, the old plan p is removed and p' becomes the new network plan.

4.3.3 Withdrawing a User Query

What we have discussed so far allows new queries to be submitted to the system. It remains to be seen how to deal with user queries that are withdrawn. The algorithm shown in Figure 4.5 performs this step. This routine is called periodically. It is possible to call the routine as soon as a user query is stopped, however changing the network query too often would create too much traffic in the network. By calling the routine periodically with a sufficiently large interval the overhead is minimized. Recall that the query executed by the sensor network can be changed in two different ways. First, the network query can be replaced by a different query. Second, the sampling period of the running network query can changed directly. Since any modification is associated with a cost, the algorithm first analyzes whether it is cost effective to change the network query. The algorithm is based on a *penalty function* that indicates how well the current network execution plan matches the user queries. The first penalty function $f_r(p,\mathcal{U})$ compares the sampling periods of all user queries from the set U with the one of network plan p. The shorter the sampling period p_s of the network query compared to the "tolerant" common sampling period of the user queries, the larger is the penalty value.

```
1  if $f(p,\mathcal{U}) > \phi_m$ then
2      create new network plan $p'$ ;
3      $p'_A := \bigcup_{u\in\mathcal{U}} u_A$ ;
4      $p'_s :=$ TGCS($\mathcal{U}$) ;
5      $p'_P := \bigcap_{u\in\mathcal{U}} u_P$ ;
6      inject $p'$ into the network;
7      wait until $p'$ has generated tuples ;
8      foreach $u \in \mathcal{U}$ do
9          $u_\rho := {}^{u_s}/_{p'_s}$;   $u_\sigma := u_P \backslash p'_P$; // migrate query to new plan
10     remove old plan $p$ ;
11     $p := p'$ ;
```

Figure 4.5: Strengthening the network execution plan after withdrawing a new user query

We define this penalty function as follows

$$f(p,\mathcal{U}) = \left(\frac{\text{TGCS}(\mathcal{U})}{p_s} - 1\right) + \alpha \cdot \left| p_A \setminus \bigcup_{u\in\mathcal{U}} u_A \right| + \beta \cdot \left| \bigcap_{u\in U} u_P \setminus p_P \right| .$$

The first term the sum is zero if and only the actual sampling period of the plan p corresponds to the best common sampling period as determined by TGCS from the current set of user queries. The second term in the penalty function determines the overall matching of the selected attributes. It counts the number of attributes selected by the network query that are no longer needed by any user query. This is done by computing the difference set of the network query's attribute set and that of every user query. The third term describes the matching of the selection predicates by counting the predicates all user queries have in common that do not appear in the network query. For example, these predicates might have been missing in a previous user query, thus preventing their inclusion in the network query. The value of penalty function $f(n, U)$ is a weighted sum of these three terms.

The parameters α and β are tuning parameters that are initially chosen to reflect the characteristics and the desired behavior of the sensor network. Replacing a network plan and migrating the user queries to the new network query is associated with a migration cost threshold ϕ_m. If the value of the penalty function is larger than the cost threshold, the change is performed. The cost threshold is set at configuration time and is chosen to reflect the characteristics of the network and the desired behavior of the system. For example, if the sensor network is not very reliable, i.e., many messages are lost, and the energy consumption for setting

```
input  : user queries $u_1, \ldots, u_m$, resolution $R$, min. period $s_{\min}$, tolerance $\varepsilon$
output: common sampling interval $pR$
1 $p := \left\lfloor \frac{\min_{u_i} u_{is}}{R} \right\rfloor$;
2 while $pR > s_{\min} \;\wedge\; \neg\big(\forall u_{is} : \exists k_i \in \mathbb{N} : (1-\varepsilon)u_{is} \leq pRk_i \leq u_{is}\big)$ do
3     $p := p - 1$ ;
4 common sampling interval is $pR$ ;
```

Figure 4.6: Algorithm to compute the "tolerant" greatest sampling period (TGCS)

up a new query is large, larger values for the cost threshold ϕ_m are chosen, such that updates are performed less frequently.

4.3.4 Combining Sampling Rates

In the previous sections, the effects of different sampling periods specified in the user queries were discussed. When assigning different user queries to the same network plan a common sampling rate needs to be determined. This is done using the TGCS method. The data stream that is generated by that plan does not necessarily have to match the rate of every user query, hence, rate-conversion needs to be performed on the stream before it can be returned to the user. In this section we describe TGCS and the implementation of the down-sampling operator ρ.

A "tolerant" algorithm for determining the greatest common sampling period. The "tolerant" greatest common sampling period determination algorithm (TGCS) is related to the Euclidean greatest common divisor algorithm (GCD). The algorithm is shown in Figure 4.6. The problem is to find the greatest common sampling period for a set of user queries $\mathcal{U} = \{u_1, u_2, \ldots, u_m\}$ with sampling periods $u_{1s}, u_{2s}, \ldots, u_{ms}$ milliseconds each.

In embedded systems, scheduling is typically performed in discrete time steps. This granularity is given by the resolution R of the underlying hardware timer. For example, TinyOS provides a millisecond timer for the Mica2 and Tmote Sky platforms. This timer is driven by an external 32.768 kHz clock crystal and runs at 1,024 Hz. Hence, one clock tick approximately corresponds to 0.977 ms. SwissQM supports scheduling of queries at the timer resolution $R = 0.977$ ms. TinyDB uses a resolution of $R = 256$ ms. The resulting common sampling period therefore has to be quantized to integer multiples of the timer resolution.

The key idea of a "tolerant" version is to allow an error in the effective sampling period in anticipation of a higher common sampling period (even if some $\lfloor \frac{u_{is}}{R} \rfloor$ are relative prime). The largest relative error that can be tolerated is specified by a constant ε. In SwissQM, this parameter is used for all queries but could just as well be specified for every query individually. We enforce that the effective common sampling period p_s of the network query remains within the error bounds such that condition (4.3) holds. Note that the constraint allows the common sampling interval to be less but never larger than any u_{is} because the user application is more likely to be able to handle a surplus of data than missing tuples. Condition (4.3) states that there must be a sampling period within the interval $[(1-\varepsilon)u_{is}, u_{is}]$ that is evenly divisible by p_s. Since the "tolerant" greatest common sampling period p_s cannot be larger than the shortest user interval, $p_s \leq \min(u_{is})$ must hold, resulting in at most $\lfloor\frac{\min(u_{is})}{R}\rfloor$ "candidate" lengths for p_s. The TGCS algorithm then iterates over all possible candidate lengths and checks if condition (4.3) is satisfied, as sketched in Figure 4.6. In the worst case, the largest candidate length is returned because the algorithm starts with the largest "candidate" interval and then stepwise reduces the period until the shortest common sampling interval $s_{\min}$ is reached, which can be specified as additional system parameter. It prevents the system from entering an inefficient mode of operation due to congestion of the network when the sampling interval of a network query is chosen too small.

The TGCS algorithm requires at most $\lfloor\frac{\min(u_{is})}{R}\rfloor - \frac{s_{\min}}{R}$ iterations. During each iteration, condition (4.3) has to be checked for all m user queries. The time complexity (for the range checks) therefore is $O(m \cdot \min_{u_i}\{u_{is}\})$. Although the algorithm is in principle expensive, our experimental evaluation has shown that the overhead is acceptable and completely hidden behind other costs (e.g., sending a network query to the sensor network).

Implementing down-sampling. Due to merging of different queries a network execution plan typically generates data at a higher rate than specified by the queries. The tuple stream received from the network must be down-sampled before being delivered to the users. This is done by down-sampling by the rate-conversion operator ρ operator that is placed in stream execution plan between the network and the user query (see the earlier example in Figure 4.2 on page 92). Two different intervals are distinguished: the *incoming sampling period*, i.e., the sampling period of the tuples originating from the network plan and the *outgoing sampling period*, i.e., the sampling period at which the user requests tuples. A valid implementation of a down-sampling operator must solve two problems: (1) how forwarded tuples are selected from the incoming stream and (2) how the epoch value of the forwarded tuples is (re-)computed. The approach presented here uses time stamp information obtained at the arrival of previous tuples to determine which element is to be forwarded.

Tuples sent by the sensor nodes do not contain fine-grained time stamps. The coarse-grained epoch value only allows to associate a tuple to a specific sampling epoch. As the sampling period chosen by the user can be arbitrary large, the time information in the epoch number too coarse and insufficient for down-sampling. A time stamp can be added when a tuple arrives at the gateway node. The down-sampling operator remembers the time stamp t_k of the last forwarded tuple and then forwards the first tuple that arrives at a time t_l such that $t_l - t_k \geq u_{is}$ where u_{is} is the sampling period specified in user query u_i. The operator also remembers the epoch value e of the last tuple forwarded. It then increments e and assigns this value to the forwarded tuple, yielding a correct epoch value for the user query stream. However, as simple as this idea appears at first, it has two problems:

1. If there is more than one sensor mote that generates tuples, the down-sampling does not work as expected. Assume there are N sensor motes. If no tuples are lost or filtered by a predicate in the *where* clause, N result tuples are received per epoch. Thus, the arrival time between tuples from the same epoch is less than period of the network query. Therefore the down-sampling operator as sketched above would only return at most one tuple out of an epoch consisting of N tuples.

2. Slightly too short sampling periods caused by clock drift lead to a large error in the effective sample period seen by the user. The problem is that tuples that arrive too early are simply dropped. The next tuple forwarded will then be late, introducing a large error in the effective sampling period.

There are remedies for these problems. The difficulty in the first problem is that the time stamp and epoch count of the last forwarded tuples has to be stored for *every* node in the network. This modification introduces two new difficulties. First, it requires additional storage for each node in the network. Two integer numbers need to be kept for each node, the epoch value and the time stamp of the last forwarded tuple. Second, this approach requires that the *nodeid* attribute is always present in the result field. If the network query does not select the *nodeid* field, then there is no way for the operator to associate a result with a node. A solution is to include the *nodeid* by implicitly on every network query the same way as the *epoch* attribute was defined.

The second problem can be alleviated by caching the latest tuple that arrives before its expected time instead of dropping it. The tuple found in the cache, i.e., the latest tuple, is then forwarded when the tuple is due, as specified in the user query. After being forwarded, the tuple is removed from the cache. Caching reduces the number of tuples that are intentionally dropped.

The effect is shown in Figure 4.7. The histogram shows the measured inter-tuple intervals for user query tuples and for the corresponding network execution

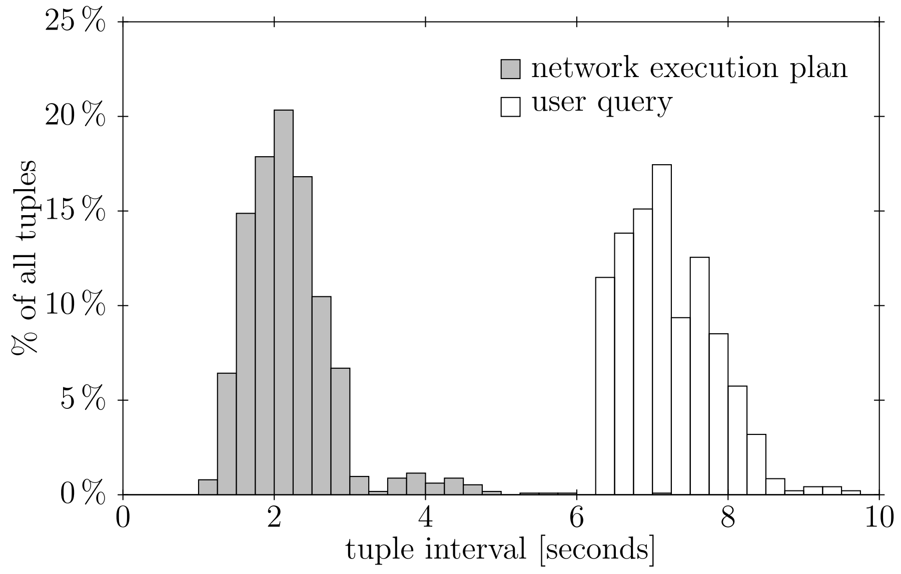
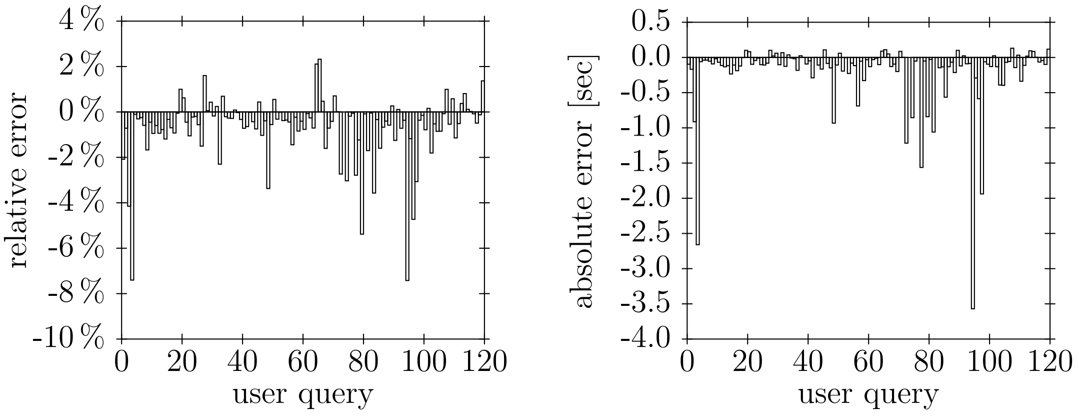


Figure 4.7: Histogram of measured inter-tuple intervals of a network execution plan with a sampling period 6,144 ms and the user query after 3 : 1 down-sampling (2,048 ms)

plan. The sampling interval specified in the network execution plan is 2,048 ms. The down-sampling operator implements a 3 : 1 rate conversion and uses caching. This translates into a user sampling interval of 6,144 ms. It can be seen that the specified user query sample period indeed is the lower bound of the measured tuple interval. Unfortunately, even the caching implementation of the operator cannot fully compensate jitter present in the incoming data stream. This is because tuples are dropped, i.e., overwritten in the cache, if they arrive within the same user query period. A dropped tuple typically results in a longer period for the next epoch.

4.3.5 Evaluation Merging Techniques

In this section evaluate the multi-query merging strategy on top of TinyDB running on small deployment consisting of three Mica2 nodes. The goal is to measure the effect of the TGCS algorithm on the resulting inter-tuple intervals seen by the user. As mentioned earlier, at the moment, we merge all queries onto one single network execution plan. This restriction is removed in the next sections of the chapter. A second network execution plan is used during the transition phase when replacing plan.



(a) Relative Error

(b) Absolute Error

Figure 4.8: Error in sampling period (measured from 120 random queries)

Experiment Setup. The shortest sampling period the network reliably delivers data is 1,024 ms. Thus, this period was configured as limit s_{min} for the TGCS algorithm. Its relative error tolerance ε was chosen as 10 %. In order to model the behavior of the users, i.e., submitting and canceling queries, we implemented a random query generator. The size of the attribute set was uniformly distributed and well as the selection of the attribute fields. The user queries were created from a Poisson process at with an arrival rate of 1 query/min with an exponentially distributed sampling interval (average 30 s) and an exponentially distributed execution duration (average 10 min). Data was gathered for 120 queries, then the average sampling interval between consecutive tuples as seen by the user was measured for each query individually.

Sampling Interval Error. The average relative error in the sampling interval, measured as the interval between two tuples with consecutive epoch numbers is shown in Figure 4.8(a). The figure shows the average for each of the 120 queries. Similarly, Figure 4.8(b) shows the absolute sampling errors in milliseconds. The largest error for a measured query interval was -7.42 %. This demonstrates that the sampling period error indeed was bounded by the specified 10 % error margin specified and that the system is able to process the given query load.

Sampling Interval of Network Plan. Figure 4.9 shows the sampling rate of the network execution plan throughout the experiment. Starting with the fourth user query, the network query contained all five sensor attributes. In total 22 different network execution plans were used. The changes of the sampling period of the network query are depicted in Figure 4.9. In the figure the highest sampling

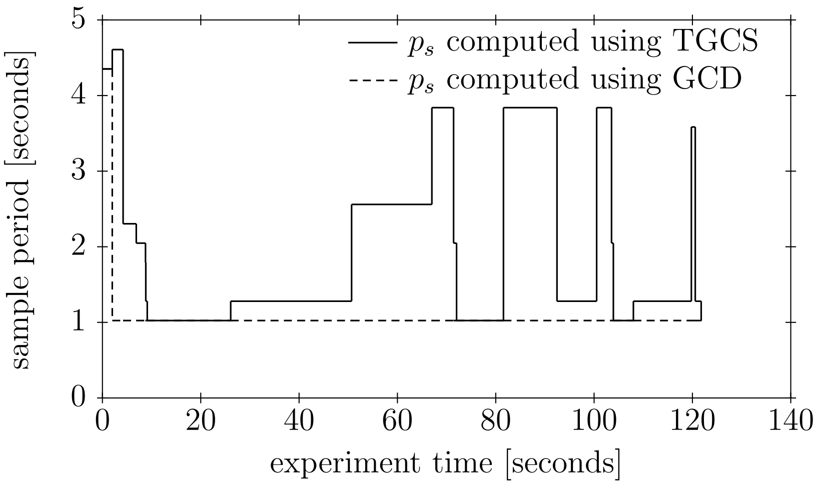


Figure 4.9: Sample period p_s of the network execution plans during experiment (120 random queries) with TGCS and GCD algorithm

rate with 1,024 ms period was only used during 26.4 % of the experiment time (when the TGCS algorithm is used to determine the common sampling frequency). The largest sampling period reached during the test is 4,608 ms. This corresponds to an improvement by a factor 4.5 compared to using the sampling period of the universal network query.

For evaluating the TGCS algorithm against the basic GCD algorithm, the sampling period of the network plan computed using GCD is also depicted in Figure 4.9. Starting from the second user query, GCD always returned the sampling period of the universal network query. This has a significant impact on the number of messages that are sent.

Message Count. Figure 4.10 represents the number of messages that were sent through the network. The bar on the left in Figure 4.10 indicates the number of messages that would have been sent if the user queries had been directly broadcast into the network without merging. Since TinyDB itself is not capable of executing a large number of queries concurrently, the message count for the non-merging case was computed based on the queries' sample period and execution duration. The right bar in the figure shows the message count for the merged queries that were sent into the network using merging and TGCS. It can be seen that by applying merging and the TGCS algorithm the number of result messages is reduced by 35 %.

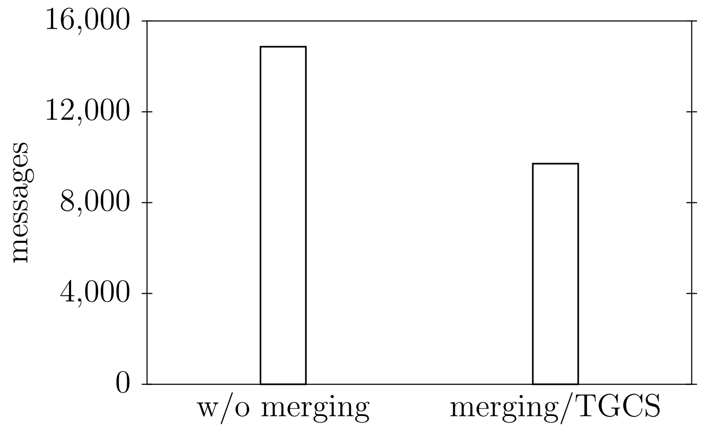


Figure 4.10: Number of tuple messages sent during the experiment

Further Optimizations. The moderate reduction of number of messages by 35 % shown above is due to the fact in this evaluation all user queries are merged into a single network query. This is not always the best option. Consider, for example, two user queries with sampling periods $u_{1s} = 7\,\mathrm{s}$ and $u_{2s} = 5\,\mathrm{s}$ and the timer resolution $R = 1\,\mathrm{s}$. The common sampling period for the network query determined by the TGCS algorithm using $\varepsilon = 0.1$ is then $p_s = 1\,\mathrm{s}$ as any other choice would violate the specified error boundaries. Hence, one message per second is sent by every node in the network. However, when executing the two queries separately only $\frac{12}{35} \approx 0.34$ messages/s are sent per node. Thus, merging queries does not always reduce the message count. As a consequence alternative strategies have to be developed. In the next section, we introduce a detailed cost model that includes not only the number of messages sent but also the overall energy consumption in the sensor network, such as the energy required to access the sensors. Using this cost model, we develop more complex optimization strategies that are able to build optimal groups of user queries. User queries within a group are then merged using the algorithms described in this section and sent into the network.

4.4 Energy-based Cost Model

In the previous analysis the cost metric used was the number of messages that generated in the network. The simple model does not consider the individual size of the messages, e.g., transmitting shorter messages is cheaper than longer messages, the cost for accessing the sensors, and the network topology. The topology plays an important role for non-aggregation queries if the generated tuples are sent as separate messages. We introduce the cost model and identify the parameters for the Tmote Sky platform by measurements. Using these parameters we define a tuple-based cost model, which we then extend to an overall cost model that includes both network topology as well as the query types, i.e., aggregation and non-aggregation queries.

4.4.1 Power Consumption of a Tmote Sky Sensor Node

The power required by each component, such as the CPU, radio transmitter and receiver as well as sensors must be considered when planning the power budget of an application. Hence, the power consumption of each component must be quantified beforehand. The power consumption usually cannot be measured in real time. For example, the Tmote Sky sensor platform does not have hardware support to measure the supply current. We measured the power consumption of the *Tmote Sky* sensor node using lab equipment. The sensor platform contains a Texas Instruments MSP430 16-bit micro-controller. The radio is implemented by a monolithic Chipcon CC2420 chip. Two Hamamatsu light sensors are available. The first sensor has a larger sensitivity range that corresponds to the total solar radiation. The sensitivity of the second is in the photosynthetically active range. Both sensors are photodiodes directly connected to the A/D converter of the micro-controller. A Sensirion SHT11 sensor is available to measure relative humidity and temperature. This sensor is connected to the general purpose inputs of the controller. The Sensirion sensor chip uses a binary I^2C-like protocol for communication with the processor.

Measurement Setup. A $1\,\Omega\,(\pm 1\,\%)$ metal film resistor placed in the power supply circuit, i.e., between the laboratory power supply and the Tmote Sky sensor node. This shunt resistor measures the overall supply current used by the node. The voltage drop across the resistor is measured using a Tektronix TDS2014 digital storage oscilloscope. For synchronization, e.g., in order to detect when a new message is received, a general purpose output of the node is connected to the trigger input of the oscilloscope.

The current consumption of a Tmote Sky sensor is varies between 1–20 mA. This gives rise to a 1–20 mV drop across the shunt resistor. There is a considerable amount of noise in the current measurements (even when using a metal film resistor instead of a carbon film resistor). As a countermeasure the acquired samples are averaged, that is, rather than taking a single waveform 128 waveforms are acquired and averaged. A precise synchronization of the subsequent waveform acquisitions is realized by using the synchronization signal as a trigger source. The single noisy waveform is shown in the left plot of Figure 4.11. The right plot shows the average

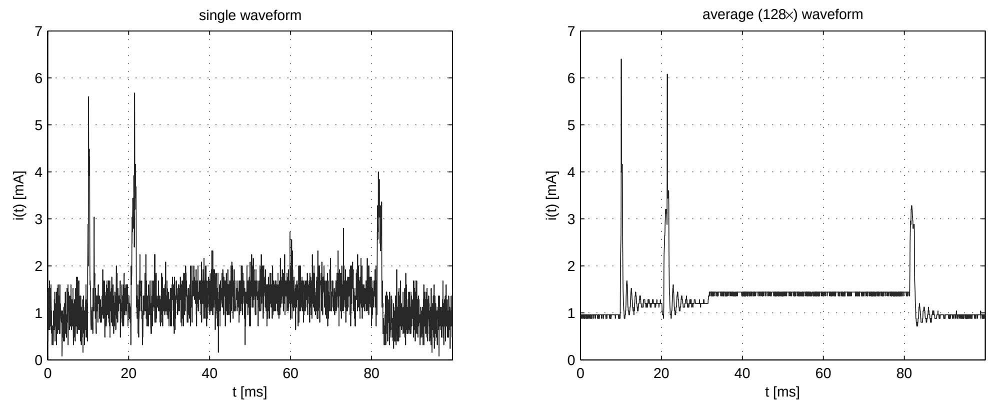


Figure 4.11: left: single wave form, right: average of 128 waveforms

computed out of 128 waveforms. It can clearly be seen that the zero-mean noise is successfully removed.

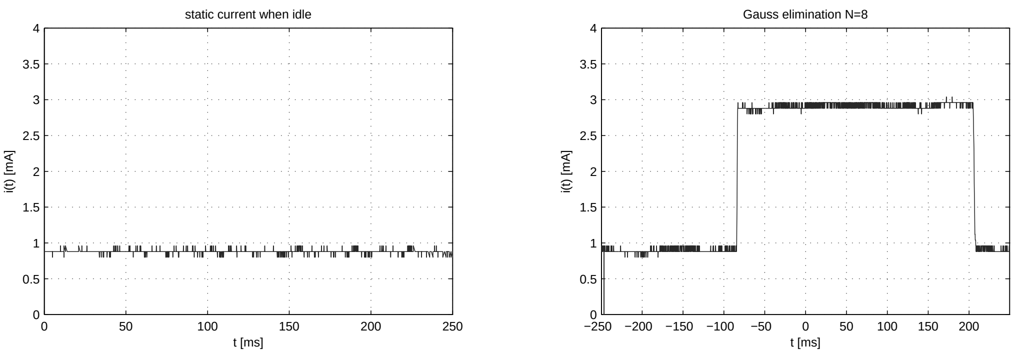
Based on the current measurements $i(t)$, the instantaneous power consumption $P(t)$ can be computed as $P(t) = U \cdot i(t)$ from constant supply voltage ($U = 3\,\mathrm{V}$). The energy (in Joules) E_k of operation can be computed by integrating over the duration time

$$E_k = \int_0^{t_k} P(t)\,\mathrm{d}t = U \int_0^{t_k} i(t)\,\mathrm{d}t$$

MSP430 Micro-controller. For measuring the power consumption of CPU intensive workloads on the Tmote Sky node a solver for a linear system (Gauss elimination) of equations is used as a representative work-load. The actual CPU workload does not have a large influence on the power used due to absence of out-of-order execution and caches. All calculation are performed in 32-bit single precision arithmetic (`float`) that is emulated in software due to the lack of a floating input unit.

Whenever the micro-controller is not used, i.e., the TinyOS task queue is empty and no interrupts have to be served, no communication over the UART takes place, and no sensor is currently being sampled, the processor is put into low-power mode. It is awakened by the next interrupt. The power consumed in low-power mode is dominated by the static power required by the components that remain active. The measurements where obtained by a simple application that disables the interrupts and puts the controller into low-power mode. Figure 4.12(a) shows the static power consumption. The supply current is constant at a level of 0.88 mA, resulting in a static power consumption of 2.64 mW.

Figure 4.12(b) shows the current consumption when execution the Gauss elim-



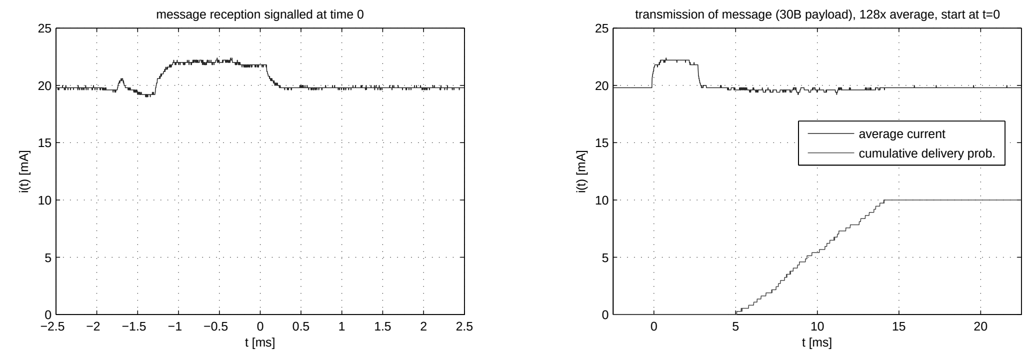
(a) Static supply current when idle

(b) Supply current when executing Gauss elimination

Figure 4.12: Supply current of the MSP430 microcontroller

ination for an 8×8 system. The algorithm starts at time $t = 0$. The current increase for $t < 0$ is due to the fact that the CPU is also active when preparing the system matrix and the constant vector. It takes requires 84 ms to initialize the system in addition to the 207 ms for solving it (resulting in an emulated floating-point performance of 2.3 kflops). The supply current peak is 2.92 mA, resulting in an active CPU power consumption of 8.76 mW.

Radio Receiver. These measurements show the amount of power used when receiving a TinyOS message with 2 byte payload (total size 13 bytes). When the receiver is turned on the CC2420 chip draws 18.8 mA according to the datasheet. This such called idle listening cost occurs independent whether a message is currently received. As the current measurements in Figure 4.13(a) show this current is only exceeded by a small amount when a message is actually received. When the receiver is idle the overall supply current drawn is 19.8 mA (this includes the 0.88 mA static current identified earlier), resulting in a idle power of 59.4 mW. In Figure 4.13(a) the reception of a single message is shown. The reception is signaled to the application at time $t = 0$. The surge in current consumption of approximately 2.2 mA starts at $t = -1.75$ ms. The peak disappears at $t = 0.27$ ms. This gives a lower bound time it required to receive a message $t_{rx} \geq 2.02$ ms where the receiver must be active. Of course, in order to compensate clock skews in the synchronization of sending and receiving node, as well as MAC back-off (and retransmit if available) the receiver usually operated at a much larger duty cycle. In the interval $[-1.75\,\text{ms}, 0.27\,\text{ms}]$ the charge transport, i.e., the integral is $42.7\,\mu$As. This leads to a lower bound of the energy for receiving a single message $E_{rx/msg} \geq 128.1\,\mu$J. Averaging that over t_{rx} gives a mean power consumption of



(a) Supply current when receiving a radio message. Receiver permanently activated

(b) Supply current when transmitting a 30 bytes message.

Figure 4.13: Supply current of CC2420 radio chip

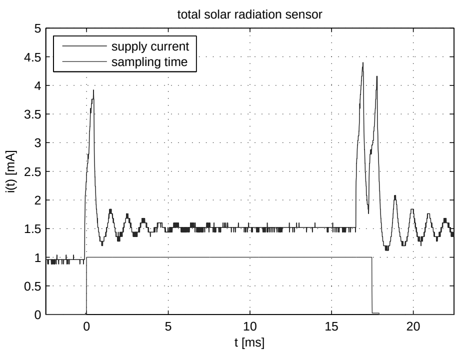
$P_{rx} = 63.4\,\text{mW}$.

Radio Transmitter. In order to the measure the power consumption of the radio transmitter, a TinyOS application was written that periodically sends a single 30 byte message (41 bytes including header). The CC2420 radio is configured to operate at the highest output setting 0 dBm (1 mW).

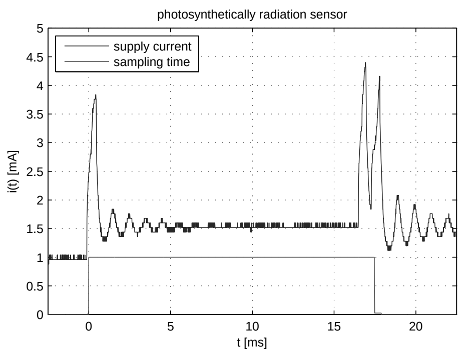
The average current is shown in Figure 4.13(b). Since the message transmission is non-deterministic, for example, channel assessment, back-off, etc. the cumulative probability density function ($F(t)$ is shown (ramp curve in the figure) that indicates the probability that the transmission has been completed by then. Completion is signaled to the application, which then sets a general purpose output connected to the oscilloscope. The message transmission is initiated by the application at time $t = 0$. Since $F(t) = 0$ for $t < 5\,\text{ms}$ it follows that at least 5 ms are required for the transmission. On the other hand, after at most 14.07 ms the transmission is complete since $F(t) = 1$ for $t \geq 14.07\,\text{ms}$. From the density function $\frac{d}{dt}F(t)$ the expected transmission duration can be determined.

$$\begin{aligned}\bar{t}_{rx} &= \int_{t=0}^{14.07\,\text{ms}} t \cdot \frac{d}{dt}F(t)\,dt \\ &= 14.07\,\text{ms} - \int_{t=0}^{14.07\,\text{ms}} F(t)\,dt \approx 9.63\,\text{ms}\end{aligned}$$

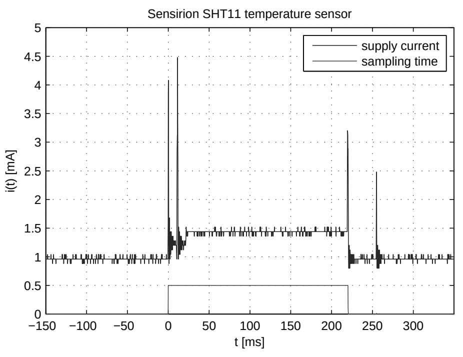
In average, the transmission is completed after $\bar{t}_{tx} = 9.63, \text{ms}$. The average charge used for a transmission can be computed as the integral of current and the density function.



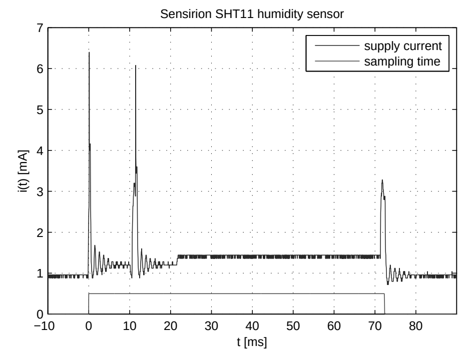
(a) Supply current when sampling the total solar radiation sensor



(b) Supply current when sampling the photosynthetically active radiation sensor



(c) Supply current when sampling the Sensirion SHT11 temperature sensor



(d) Supply current when sampling the Sensirion SHT11 humidity sensor

Figure 4.14: Supply current of sensors

$$\int_0^{5\,\mathrm{ms}} i(t)\,dt + \int_{5\,\mathrm{ms}}^{14.07\,\mathrm{ms}} \frac{d}{dt}p(t)\left[\int_{5\,\mathrm{ms}}^{t} i(\tau)\,d\tau\right] dt \approx 196.11\,\mu\mathrm{As}$$

This results into an energy cost of $E_{tx/msg} = 589\,\mu$J per message. Divided by $\bar{t}_{tx}$ the average power consumption for sending is $P_{tx} = 61.07$ mW.

Light Sensors. The light sensors are implemented as photodiodes that are operated in zero bias mode. The dark current of the diode generates a voltage drop over a measurement resistor. This drop is directly measured by the A/D converter of the microcontroller. The cost for accessing the light sensors is therefore only given by the cost of operating the A/D converter. The readings for the two light sensors are thus identical (Figure 4.14(a) and 4.14(b)). In both cases the sampling requires 17.48 ms. In order to determine the latency of reading the sensors the general purpose output is used for signaling again. The pin is set immediately before the sampling is initiated. The pin is cleared as soon as the sensor data is ready.

From the two Figures 4.14(a) and 4.14(b) is can be seen that after the sampling is initiated the circuit is oscillating. This behavior is difficult to explain. In order to compute the charge used for the sampling only the interval from $t = 0$ until $t = 17.48$ ms is considered ignoring the oscillations. For the total solar sensor the parameters are as follows:

$$\begin{aligned} Q_{tsr} &= 28.80\,\mu\text{As} \\ E_{tsr} &= 86.43\,\mu\text{J} \\ t_{tsr} &= 17.48\,\text{ms} \\ P_{tsr} &= 4.94\,\text{mW} \end{aligned}$$

For the photosynthetically active radiation sensor the parameters are:

$$\begin{aligned} Q_{tsr} &= 29.18\,\mu\text{As} \\ E_{tsr} &= 87.53\,\mu\text{J} \\ t_{tsr} &= 17.48\,\text{ms} \\ P_{tsr} &= 5.01\,\text{mW} \end{aligned}$$

Temperature and Humidity Sensor. The Sensirion SHT11 temperature and humidity is connected to the general purpose I/O pins of the processor. For communication a protocol similar to the serial I^2C-bus is used. Hence, reading this sensor requires much more time than sampling the light sensors as the Figures 4.14(c) and 4.14(d) show. Two large transients can be seen both at the beginning of the sample acquisition and at the end. As in the previous case, the general purpose I/O pin is used to measure the duration of the sampling process. For the temperature sensor (Figure 4.14(c)) the parameters are as follows:

$$\begin{aligned} Q_{tsr} &= 317.0\,\mu\text{As} \\ E_{tsr} &= 951\,\mu\text{J} \\ t_{tsr} &= 220.4\,\text{ms} \\ P_{tsr} &= 4.32\,\text{mW} \end{aligned}$$

For the humidity sensor (Figure 4.14(d)) the identified parameters are:

$$\begin{aligned} Q_{tsr} &= 103.2\,\mu\text{As} \\ E_{tsr} &= 310\,\mu\text{J} \\ t_{tsr} &= 72.36\,\text{ms} \\ P_{tsr} &= 4.28\,\text{mW} \end{aligned}$$

Power Parameters. The results confirm current assumptions about sensor networks:

1. Power consumed by the CPU/microcontroller is almost negligible, at least compared to the power spent for radio communication. In absence of timing constraints CPU work can be considered free.

2. Sampling, at least using the primitive sensors built-in on the Tmote Sky node is free.

3. Power consumption is dominated by the radio. In particular, having the receiver turned uses most of the power. Transmission overhead is negligible compared to receiving.

A summary of the power parameters of the Tmote Sky node is shown in Table 4.2. The sensors approximately require the same amount of power, although the time required to acquire a sample varies considerably. Processing costs and sensor access are both negligible compared to the radio cost, which is dominated by the receiver. Whereas for transmission an average transmission time $\bar{t}_{tx}$ can be specified, the duration the receiver is activated depends on the activation. A longer activity increases the overlap between sender and receiver and hence can increase the data yield. This duty cycle needs to be chosen carefully.

4.4.2 Per-Tuple Costs

The ultimate optimization goal is to minimize the total power consumption over all nodes. We start by calculating the cost of acquiring and transmitting a tuple to the base station. The energy consumption E_T of a sensor node for a single tuple is given by the following expression.

$$\begin{aligned} E_T &= \underbrace{\sum_{s_k \in S} t_{s_k} P_{s_k}}_{\text{sampling}} + \underbrace{t_{rx} P_{rx}}_{\text{reception}} + \underbrace{t_{tx} P_{tx}}_{\text{transmission}} \\ &\quad + \underbrace{t_{cpu} P_{cpu}}_{\text{CPU active}} + \underbrace{t_i P_i}_{\text{idle}} \end{aligned} \tag{4.6}$$

Table 4.2: Measured parameters for energy model for the Tmote Sky platform

Parameter		**Value**	**Unit**
Broad-band light sensor	P_{tsr}	4.94	mW
	t_{tsr}	17.5	ms
Narrow-band light sensor	P_{par}	5.01	mW
	t_{par}	17.48	ms
Temperature sensor	P_t	4.32	mW
	t_t	220.4	ms
Humidity sensor	P_h	4.28	μW
	t_h	72.36	ms
Radio transmitter	P_{tx}	61.1	mW
Sending message (41 byte)	$\bar{t}_{tx}$	9.63	ms
Radio receiver	P_{rx}	63.4	mW
Receiving message (min)	$t_{rx} \geq$	2.02	ms
CPU active	P_{cpu}	8.76	mW
Idle power	P_i	2.64	mW

The expression captures the cost of sampling, receiving and transmitting, running the CPU, and the power consumed while idle. The cost of sampling a sensor is the product of the time t_{s_k} to take a sample and the power consumption P_{s_k} of each sensor. The overall sampling cost is the sum over all sensors sampled. For message transmission and reception we use the time the transmitter and receiver circuits are active multiplied by the power consumption for sending and receiving. The same applies to the CPU. The idle power consumption P_i is the power used by the node when the radio is powered-off and the CPU is in the lowest power mode.

Table 4.2 shows the values for the parameters we have obtained through shunt-measurements on the Tmote Sky sensor platform as described in the previous section. For some of these parameters, what we have measured is the minimum value. The real values of these parameters are affected by the software running on the node. For instance, the receiving time t_{rx} is application dependent (it depends on the duty cycle, message length, and size of the network). In SwissQM we have measured the real value to be 50 ms.

Table 4.2 indicates as it also noted throughout the literature that energy consumption is dominated by the transmission costs. In fact, the major power drain occurs in the radio receiver.[2] For the model we assume a MAC layer that performs the necessary duty-cycling of the receiver. Thus, without any significant loss of accuracy, we can simplify the model by considering only the cost of transmitting tuples and disseminating network execution plans. Although our implementation uses messages of different sizes and supports tuple bundling in one message, in the cost model we assume each tuple costs at least one message. Some tuples are too long to fit into one message, hence, the cost $c(T)$ for sending a single tuple T that contains n attributes is given as follows:

$$C(T) = C_m \left\lceil \frac{n}{8} \right\rceil + C_a n \tag{4.7}$$

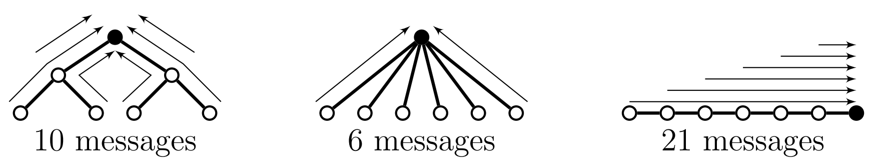
C_m is the fixed cost for sending a message. In our system the fixed costs are the 25 header bytes of for each full 41 byte result message. C_a is the cost of sending a single 16-bit attribute. The fraction $^1/_8$ comes from the maximum number of attributes we can fit into a message (in other systems and configurations this value may be different). C_m and C_a can be inferred from the bit-transmission cost. With our hardware we measured $C_m \approx 359\,\mu\text{J}$ and $C_a \approx 28.7\,\mu\text{J}$. In the following, we take the simplifying assumption that a tuple is not split across multiple messages, i.e., $n \leq 8$. As an approximation we use $C_m \approx 12C_a$. All energy calculations are done in units of C_a.

4.4.3 Overall Cost Model

Sensor networks typically resort to multi-hop routing to transmit messages to the base station. This is the basis for in-network data aggregation algorithms, e.g., [MFHH02].

There are several strategies to define the routing topology. For instance, in Directed Diffusion [IGE+03] the routing depends on the data and task at hand. To calculate the cost model we assume a tree routing topology that is independent of the queries and data collected. There are many possible tree topologies. The actual topology determines the number of messages required to get a tuple to the basestation. Figure 4.15 illustrates different topologies that each consist of six sensor nodes and a basestation. For the balanced tree, 10 messages are exchanged per epoch. The star requires with six messages the least number of transmissions. The chain topology has the worst message complexity of 21 messages. In general, for N-node topologies in Figure 4.15, the total number of messages is of the order of $O(N^2)$ for the chain, $O(N \log(N))$ for the binary tree, and $O(N)$ for the star. This overhead changes if aggregation is performed in the network. If the

[2]The sensors of the Tmote Sky node platform we are using do not have a significant energy consumption. For different sensors (e.g., gas sensors) sampling cost may no longer be negligible. In this case, the sampling cost has to be included in the parameter C_a.



(a) Balanced Tree (b) Star Graph (c) Chain Graph

Figure 4.15: Different collection tree topologies

aggregation state, i.e., the summary data that is used to compute the value of the aggregate, is of constant size, which is the case for non-holistic aggregates as described in [MFHH02], the number of messages that have to be sent is proportional to the number of edges in the routing graph, thus, $O(N)$. Ideally, only one message needs to be sent over an edge in the tree.

To derive a cost model for the average power consumption over all nodes, we need to distinguish between spatial aggregation and non-aggregation queries. In the following, t_p is used for the sampling interval, C_T is a parameter describing the topology, and N is the number of nodes in the network. With Equation (4.7) as a basis, Equation (4.8) describes the costs for the non-aggregation case, whereas Equation (4.9) is used for aggregation queries. The execution cost $C(p)$ for a given plan is the total electrical power used by the sensor nodes (physical units mW) to execute that plan:

$$C(p) = C_T \frac{1}{t_p}\left[C_m + C_a n\right] \tag{4.8}$$

$$C(p) = (N-1)\frac{1}{t_p}\left[C_m + C_a n\right] \tag{4.9}$$

In [XLTZ07] similar equations are proposed. However, because they cannot determine the actual topology, the cost model they use is based only on Equation (4.9) (the lower bound). In SwissQM we can determine the value of the C_T parameter as follows. A node h hops away from the basestation leads to h messages being sent towards the base station. For the average energy consumption one can consider the average hop-distance $\bar{h}$ of a node. Together with the number of nodes, the topology parameter is then $C_T = N\bar{h}$, i.e., the average number of hops to the base station multiplied by the number of nodes. For example, for the tree graphs shown in Figure 4.15 $\bar{h}$ is $5/3$ for the balanced tree, 1 for the star graph, and $7/2$ for the chain graph. In modern sensor networks the topology is typically dynamic. In our system we can obtain the current value $\bar{h}$ directly from the network by executing the following query in the background at a sufficiently

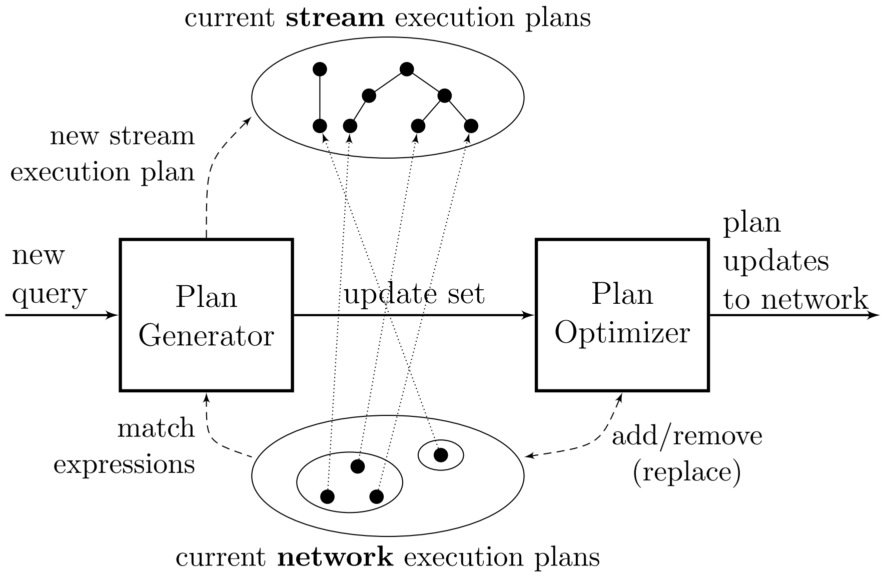


Figure 4.16: Query processor consisting of a plan generator and plan optimizer

large frequency to react properly to changes:

```
SELECT  AVG(depth)
  FROM  sensors
 EVERY  1 min
```

This value is used by the query optimizer to constantly keep the cost model up to date. In Section 4.5.5 we extend this cost model to include query selectivity.

4.5 Query Processor

Submitted queries are passed to the *query processor*, which then generates *network execution plans* and *stream execution plans*. Network execution plans are submitted to the sensor network and each one of them produces a data stream. These result data streams are processed by the stream execution plans that extract the final result streams for the user queries.

The query processor consists of a *plan generator* and a *plan optimizer*. Figure 4.16 illustrates these two components and how they operate. New queries arrive at the plan generator. The plan generator uses the new query and the set of *current plans* (the plans currently running in the network) as input and produces a stream execution plan and an *update set*. The update set specifies what additional data needs to be requested from the network to answer the new query (note that the update set can be empty). The update set is given as an input to the plan optimizer. The plan optimizer takes the update set and the set of current plans and produces a new set of current plans. It also forwards any changes to the current plans to the sensor network for execution.

Query optimization happens in two stages. The plan generator tries to answer the new query by using the result data streams already being produced. If it cannot, it generates the update set. The plan optimizer uses different cost model based strategies (Section 4.5.2) to minimize the cost of propagating changes and running the set of current plans.

The plan optimizer does not need to generate network execution plans that acquire exactly the data asked by the queries. Depending on the optimization strategy, there might be *redundant data* (two separate plans return overlapping data sets) and *orphan data* (data obtained from the network for which there is currently no user query). The latter reduce the cost of removing plans if they are no longer needed. Leaving them around for a while may better than removing them and having to re-insert them shortly afterwards.

4.5.1 Plan Generator

A new query is sent to the plan generator after it has been parsed. The generator looks up all possible subexpressions of the query in the set of current plans. When no match is found, the corresponding expression is added to the update set, which is then passed to the plan optimizer.

Ignoring selective queries (queries with filter predicates are described later in Section 4.5.5) a query can be regarded as a collection of expressions $\{E_1, E_2, \ldots\}$. Such an expression can either be a stream operator (as presented in Section 4.2.1) or an arithmetic expression. Every expression has an additional property in our stream processor, the *tuple rate* r. It determines how often the operator is expected to produce tuples. We explicitly note the rate r together with the expression as an *annotated expression* $E@r$.

Matching expressions from a query with the ones from the execution plans involves comparing both expression subtrees and their rates. We define a binary matching relation $\preceq$ for annotated expressions:

$$E_i@r_i \preceq E_j@r_j \quad \Longleftrightarrow \quad \big(E_i = E_j\big) \ \wedge \ \big(r_j \mod r_i \equiv 0\big)$$

According to this definition $E_j@r_j$ *matches* $E_i@r_i$ if, and only if, it contains the same subexpression and produces tuples at an integer multiple rate of $E_i@r_i$, i.e., a superset of the tuples of the left hand side, hence the $\preceq$ symbol. Note that $\preceq$ is not symmetric.

We can now provide a more formal definition of a query and a network execution plan. A query q is a set of annotated expressions: $q = \{E_1, E_2, \ldots, E_k\}@r$.

```
input  : new query $q$, set of existing network execution plans $\mathcal{P}$
output: update set $U$
1 $U := \emptyset$ ;
2 forall $E@r \in q$ do
3     $(M_i, U_i) :=$ MatchSubExpr$(E@r, \mathcal{P})$ ;
4     subscribe $q$ with all plans $p \in M_i$ ;
5     $U := U \cup U_i$ ;
6 return update set $U$ ;
```

Function ComputeUpdateSet

Figure 4.17: Algorithm to compute the Update Set

This notation emphasizes that all expressions of the query have the same rate r, determined by the *every* clause. The set is computed from the operator tree and contains one element for each expression in the *select* clause.

Similarly, a network execution plan is also a set of annotated expressions: $p = \{E_1, E_2, \ldots, E_k\}@r$. The difference to a query is that each execution plan maintains a list of queries that currently use data generated by the plan. This list can be used as a reference counter. When the list is empty a plan may be removed. We use $\mathcal{P} = \bigcup p$ to denote the set of all plans p.

Figure 4.17 shows the function ComputeUpdateSet that determines the update set for a query q considering the set $\mathcal{P}$ of currently executing plans. The function begins with an empty update set U and then iterates over all expressions in the given query q and calls MatchSubExpr to recursively match the subexpressions.

MatchSubExpr returns a pair (M, U) where M contains the plans associated to those subexpressions that could be matched by any expression from the current set of plans. The function is listed in Figure 4.18. U contains the unmatched subexpressions. The function first checks if the entire tree $E@r$ is matched by some plan (line 1 in Figure 4.18). If not, the function is recursively applied on the subexpressions (line 6). The matching set M_i and update set U_i obtained from each subexpression tree are combined (line 7). If no match was found ($M = \emptyset$) for any subexpression, the entire subexpression is added to the update set and returned (line 9). Otherwise, the union of the matching plans M and the update set U is returned (line 11). In ComputeUpdateSet the query q is registered with all matching plans $p \in M$. The update set passed to the query optimizer is the union of the update sets U_i obtained for each expression.

While matching the expressions of the update set with the current plans, the plan generator also creates a stream execution plan. For each match found, the attribute in the result stream of the corresponding network plan is extracted and,

```
   input  : annotated expression $E@r$, set of plans $\mathcal{P}$
   output: pair $(M, U)$ of two sets. Set $M$ contains plans with matching
           expressions. Set $U$ contains unmatched annotated expressions.
1  if $\exists p \in \mathcal{P} : \exists E_j@r_j \in p : E@r \preceq E_j@r_j$ then
       // match found in plan
2      return $(p, \emptyset)$ ;
3  else
4      $M := \emptyset$;   $U := \emptyset$ ;
5      forall subexpressions $E_i@r_i$ of $E@r$ do
           // recursively match subexpressions
6          $(M_i, U_i) :=$ MatchSubExpr$(E_i@r_i, \mathcal{P})$ ;
7          $M := M \cup M_i$;   $U := U \cup U_i$ ;
8      if $M = \emptyset$ then
9          return $(\emptyset, \{E@r\})$ // no subexpression matches
10     else
11         return $(M, U)$ ;
```

Function MatchSubExpr

Figure 4.18: MatchSubExpr recursively matches an annotated expression in a set of plans $\mathcal{P}$

if necessary, some additional processing applied, e.g., down-sampling or applying a filtering operator. In order to join two result streams, e.g., computing E_1+E_2 from two streams that provide E_1 and E_2, the tuples are combined using an equi-join. For spatial-aggregation queries the join is performed on the grouping attributes (specified in the *group by* clause). In all other cases, tuples that originate from the same node are joined, i.e., the join is performed on the *nodeid* attribute. We can save communication cost by extracting the *nodeid* attribute from the *origin* field of a result tuple message through the routing layer. Alternatively, the *nodeid* attribute could be added implicitly to every execution plan. However, this enlarges the size of a result tuple.

Example. As an illustration of ComputeUpdateSet consider the example shown in Figure 4.19. Given is a query consisting of a single expression $E_0@r = E_1@r + \omega_{t,10}(a_1@10r)$. Let $E_1@r$ be an unspecified expression tree, $\omega_{t,10}$ a tumbling window operator with a window size of 10 tuples, and a_1 a sensor attribute. Assume that there are currently two execution plans p_1, p_2 in the system.

First, the possible subexpressions are compared with the plans. Since for $E_0@r$ no match was found, the subexpressions $E_1@r$ and $E_2@r = \omega_{t,10}(a_1@10r)$ are

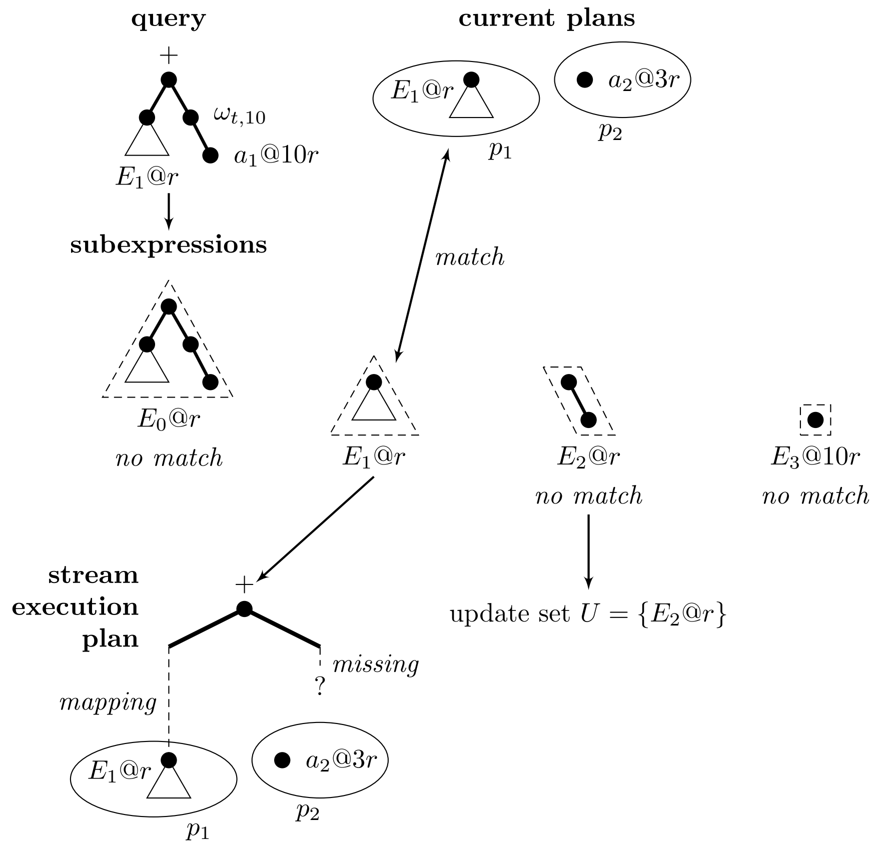


Figure 4.19: Plan Generator Example

analyzed (line 6). A match for the $E_1@r$ is found in p_1, hence, p_1 is added to the set M and the recursion stops at this point. For $E_2@r$, no match can be found. Thus, recursion continues to $E_3@10r = a_1@10r$. Also in this case, no match is found. The algorithm considers the entire branch $E_2@r$ to be unmatched and adds it to the update set (line 9). This update set needs to be passed to the optimizer, which then modifies $\mathcal{P}$ such that results for $E_2@r$ are retrieved from the network. This is explained in the next section. The streams from $E_1@r$ and $E_2@r$ are joined in the stream execution plan by adding an addition-operator. In that step the query also subscribes with the existing plan p_1. □

4.5.2 Plan Optimizer

Unmatched expressions that are passed to the optimizer by the plan generator need to be included into some execution plan (see Figure 4.16, page 116). The optimizer has two choices how to deal with the update set. Either it can add it as a new plan or merge it with an existing plan. All changes in the set of execution plans are associated with a cost. Hence, not only the execution of the plans but also the set up and removal costs of the plans have to be considered. There are different strategies the optimizer can apply, depending on the cost-aspect.

4.5.3 Cost of Submitting Queries

Before we describe the possible optimization strategies we first state our assumptions on network execution plans:

(1) Execution plans are immutable, i.e., they cannot be modified after they have been generated. The reason is that the execution platform does not support modifications of plans. A modification (besides changing the sampling interval, which is possible in, e.g., TinyDB) could in principle consist of large changes. The coordination of these distributed updates is difficult (global snapshot semantics on the state of a plan is required), therefore, we decided not to support updates on plans. The immutability implies that in order to update a plan it has to be replaced.

(2) The execution of a plan results in a power drain $C(p)$ in mW (Section 4.4.3). The actual energy consumption (Joules) depends on how long a plan is executed.

(3) Setting up a new execution plan gives rise to an energy cost C_s (in Joules). This energy is spent in disseminating the plan through radio messages. These costs depend on the size of the network and the complexity of an execution plan. Our implementation uses a flooding mechanism, hence, a given plan message is retransmitted once by every node. This results in $m \cdot N$ transmissions for a plan that is disseminated with m messages. We measured an energy consumption of $589\,\mu$J for sending a full radio message. Let $m(p)$ denote the number of messages required for representing plan p. Then the setup cost is

$$C_s(p) = 589\,\mu\text{J} \cdot N m(p) \quad .$$

(4) For stopping and removing an execution plan an additional energy cost C_r (in Joules) is defined. In our implementation a plan is removed when a *stop* message is received. C_r is the cost for broadcasting one single message and is independent of the plan. Thus, $C_r = 589\,\mu\text{J} \cdot N$.

(5) The number of execution plans that can be run concurrently in the sensor network is limited. This limitation is not only due to the limited bandwidth available for the transmission of result tuples but also to the limited CPU/memory on the sensor node. Our implementation on the Tmote Sky sensor nodes can execute up to 8 plans concurrently.

(6) Finally, as already mentioned in the previous section, all annotated expressions $E_i@r$ of a given plan must have the same rate r.

Obviously, there is a trade-off between cheap execution costs, i.e., “good” plans that best match the current set of queries, and a low update cost due to less frequent changes of the plan. The goal that is consistent with maximizing the battery life time is to minimize the energy used to process a given query load. The query load is characterized by the set of queries it consists of and the submission and withdrawal times of each query. In a realistic scenario we cannot assume that withdrawal times are known beforehand. Nevertheless, the execution time of a query plays an important role, e.g., for a long running query it might be worthwhile to update execution plans as the update costs are amortized by the long running queries.

4.5.4 Optimizer Rules

We study different strategies for the optimizer. The optimization strategy determines how the optimizer modifies annotated expressions from the set $\mathcal{P}$ of execution plans and the update set U. In general, the optimizer can modify $\mathcal{P}$ based on following rules:

1. If an expression $E_j@r_j$ is not matching expression $E_i@r_i$, i.e., $E_i@r_i \not\preceq E_j@r_j$, because only their rates are incompatible while $E_i = E_j$, the rate of both expressions can be adapted to the *least common multiple* $\text{lcm}(r_i, r_j)$. The expressions can then be merged.

$$E@r_i, E@r_j \longrightarrow E@\text{lcm}(r_i, r_j)$$

   In order to adapt the rate of the result streams rate conversion operators ρ have to be added to the stream execution plan. For the stream of $E@r_i$ the down-sampling ratio is $\text{lcm}(r_i, r_j)/r_i : 1$.

2. If $E@r$ is a composite expression, i.e., an operator op, then the composite expression $E@r = E_k@r_k \,\text{op}\, E_l@r_l$, it is replaced by the subexpressions, i.e., $E_k@r_k$ and $E_l@r_l$. The operator op is added to the stream execution plan such that the two streams are joined into a stream for $E@r$.

3. If $E@r$ is an aggregate expression, remove the aggregate. This is a special case of Rule 2. Example: $\max(E_1 + E_2)@r$ is expanded into $E1@r + E2@r$. This transformation has a profound impact on the execution cost as now the aggregate MAX has to be computed at the base station and therefore a tuple from every node needs to be sent to the base station. If a stream for $(E_1 + E_2)@r$ is already available no additional costs occur. In fact, the costs are reduced as no processing is required in the network.

4. Combine common subexpressions in $\mathcal{P}$ and U. If, e.g., $E_1@r \,\mathrm{op}\, E_2@r$ is in a current execution plan p and $E_1@r \in U$, then plan p can be replaced by a plan p' that contains $\{E_1@r, E_2@r\}$ instead.

4.5.5 Queries with Predicates

In this section, we extend the expression annotation introduced in Section 4.5.1 to include filter predicates, i.e., queries with σ operators. Just as with the sampling rate r, we also annotate the selection predicate $P(E_j)$ with the expression. Multiple predicates are represented in conjunctive normal form. The resulting annotated expression is:

$$E@r|P(E_j) \wedge \ldots \wedge P(E_k)$$

For example, for the single query the annotated expressions in the execution plan are:

$$\begin{array}{rl} \texttt{SELECT} & \mathit{nodeid}, (\mathit{light} + \mathit{lightpar})/2 \\ \texttt{FROM} & \mathit{sensors} \\ \texttt{WHERE} & \mathit{humidity} < 50\,\% \\ \texttt{EVERY} & 1\,\mathrm{s} \end{array} .$$

$$p = \{\mathit{nodeid}@1\,\mathrm{s}|(\mathit{humidity} < 50\,\%), (\mathit{light} + \mathit{lightpar})/2@1\,\mathrm{s}|(\mathit{humidity} < 50\,\%)\}$$

When the matching relation is redefined appropriately, the plan generator does not have to be changed. The matching operator $\preceq$ for predicates is redefined as:

$$\begin{aligned} E_i@r_i|P(E_k) \preceq E_j@r_j|P(E_l) \quad \Longleftrightarrow & \\ & E_i = E_j \ \wedge \ r_j \mod r_i \equiv 0 \ \wedge \ P(E_k) \Rightarrow P(E_l) \end{aligned}$$

The superset condition still holds. If $P(E_k) \Rightarrow P(E_l)$ the stream $E_j@r_j|P(E_l)$ is a superset for $E_i@r_i|P(E_k)$. For the plan optimizer, we add the following rewrite rule.

5. If an expression $E_i@r_i|P(E_j)$ is bound to a predicate $P(E_j)$ the predicate is removed. The predicate $P(\cdot)$ is then computed at the base station, i.e., it is added as a σ operator to the stream execution plan. In order to evaluate the predicate at the base station, a stream for the predicate expression E_j has to be available, thus, it also has to be added to the expression set together with E_i:

$$E_i@r_i|P(E_j) \longrightarrow E_i@r_i,\ E_j@r_i$$

For the query above the *humidity* attribute is also added to the rewritten annotated expressions.

$$nodeid@1\,\mathrm{s},\ (light + lightpar)/2@\,1\mathrm{s},\ humidity@1\mathrm{s}$$

The plan generator needs to first decide on the evaluation order of a list of predicates $E@r|P(E_j) \wedge \ldots \wedge P(E_k)$, and whether a predicate is applied in the network execution plan at all, i.e., whether rule (5) should be applied.

Again, the cost model is used. However, since execution plans now can contain selective expressions, the tuple cost depends on the average selectivity $\bar{\sigma}$ of a predicate. Hence, the cost Equations (4.8) and (4.9) are redefined as

$$C(p) = C_T \frac{\bar{\sigma}}{t_p}\left[C_m + C_a n\right] \tag{4.10}$$

$$C(p) = (N-1)\frac{\bar{\sigma}}{t_p}\left[C_m + C_a n\right] \quad . \tag{4.11}$$

In order to apply this model, the optimizer needs to know the selectivity of the predicates in the execution plans. As stated earlier, we use the *origin* field in the result message to associate a tuple with a sensor node, hence, the selectivity of a predicate can be estimated at the base station for every sensor node by counting the number of received tuples. This estimate is inherently affected by message loss, as missing tuples (due to lost messages) cannot be distinguished from filtered tuples. The optimizer uses the average $\bar{\sigma}$ over all nodes, thus reducing the influence of individual losses. Additionally, the decision on whether it makes sense to continuously use a selective plan, and thereby reducing reuse for different queries, is made if the selectivity is very low, e.g., $< 10\,\%$. This makes mispredictions due to message loss negligible as the delivery probability of a message is sufficiently high $(0.8 - 0.95)$. In order to determine $\bar{\sigma}$ for selective in-network aggregation plans, the number of fused readings needs to be explicitly counted (e.g., the number of values that contributed to a MAX aggregate). This statistical data has to be sent along with the aggregate state. Only for AVG and COUNT this information can be directly deduced from the aggregate state. SwissQM/Gateway currently does not estimate the selectivity of aggregation plans. The optimizer assumes always assumes $\bar{\sigma} = 1$ leading to worst-case cost estimates.

4.6 Optimizer Strategies

How optimization rules are applied is determined by the optimization strategy. We study the following strategies:

Min-Execution Cost Strategy. (Section 4.6.1) The optimizer aggressively reorganizes the current execution plan so that the execution cost $\sum_{p \in \mathcal{P}} C(p)$ is minimized. Costs for the updates, i.e., setting up and removing plans are not considered. This strategy is motivated by the fact that update costs are negligible compared to the execution costs for long running queries, or for query workloads with low arrival rates.

Delta-Plan Strategy. (Section 4.6.2) For each update set a new plan is created. When a new plan is added, the existing plans are not reorganized. This, of course only works up to a maximum number of plans that can be executed concurrently. A plan is removed if no query has subscribed to the plan. It obviously minimizes the update-costs, however, it can lead to orphan data being returned as a long running query might be subscribed with several query plans that, therefore, cannot be removed. These query plans can also return data not required by this query.

Single-Plan Strategy. (Section 4.6.3) leads to a single execution plan, such that all queries are answered from a single stream. This approach corresponds to the merging approach described in Section 4.3. As new queries are added, the selectivity of the network execution plan increases until it corresponds to the *Universal Query.* From this point on, inserting additional queries does not require any updates in the network. In order to include queries with different sampling intervals, the period of the resulting plan has to be set equal to the greatest common divisor of the queries' sampling interval. As the evaluation in Section 4.7 will show, the resulting sampling interval quickly approaches rate of the universal query, leading to redundant data.

TTMQO Strategy. This strategy was described by Xiang et al. [XLTZ07]. We have implemented TTMQO for comparison purposes. In the TTMQO strategy, instead of merging all queries to one single plan, a query q is merged with the "most beneficial" existing plan. The benefit of a plan p is defined as $C(p)+C(q)-C(\{p,q\})$, i.e., the savings when merging p with the query q compared to creating a new plan for q and running it together with p. In Xiang et al. only propose merging. Splitting a query and assigning it to different plans is not considered. Thus, their approach essentially bypasses the plan generator such that the update set contains all expressions of the submitted query. In the TTMQO strategy, a plan is not immediately removed as soon as the last query that uses data from it is withdrawn. Instead, a plan is kept, and produces orphan data. The idea is to save update costs as a new query might be submitted that could make use of that plan. When such a plan is removed is determined by an aggressiveness parameter $0 \leq \alpha \leq 1$. Xiang et al. do not describe how to choose that parameter. Their experiments indicate that the choice for parameter α does not affect the results significantly.

4.6.1 Min-Execution Cost Strategy

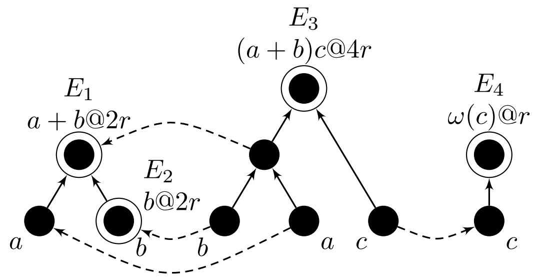
In this section we describe how to map queries such that the execution costs are minimized. The min-execution cost strategy tries to aggressively minimize the execution costs $\sum_{p \in \mathcal{P}} C(p)$ of the plans. Costs for starting and removing plans are not considered. As soon as query is added or withdrawn the set of current plans is recomputed and plans are replaced where necessary. This strategy works best if the query is in the system for a sufficiently long time as long execution times amortize update costs.

Before describing the strategy we motivate why the update costs can be ignored in a first approximation. When comparing the cost for updating a plan with the execution duration, it can be observed that in the worst case a plan is amortized soon after it has produced data as a simple calculation shows (ignoring selective plans and retransmissions). A plan update (broadcasting a *stop* command message and disseminating the new plan p using $m(p)$ messages) leads to $N(m(p)+1)$ transmissions, assuming that each node forwards each message exactly once. A worst case plan is a plan with the lowest possible execution cost, which is either a spatial aggregation plan, or a non-aggregation plan in a star topology (see Figure 4.15(b)), leading both to N transmissions per epoch. Hence a plan is amortized after $> m(p) + 1$ epochs. In SwissQM the plan size $m(p)$ is typically less than 5 messages even for complex queries with simple user-defined functions. Thus, update costs can considered being amortized after having produced data for about 10 epochs. In the common case, the average depth of the tree is $\gg 1$. Then the update cost is amortized even earlier.

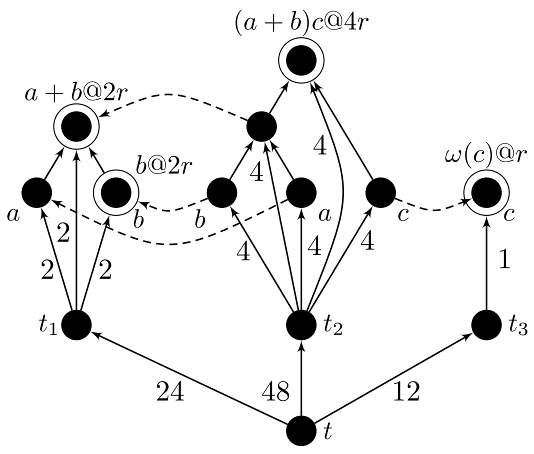
Example. Consider a set of queries that leads to the following *annotated expression set*:

$$\begin{aligned}
E_1 &: \quad (a+b)@2r \\
E_2 &: \quad (a+b)c@4r \\
E_3 &: \quad \omega_{10}(c)@r \\
E_4 &: \quad b@2r
\end{aligned}$$

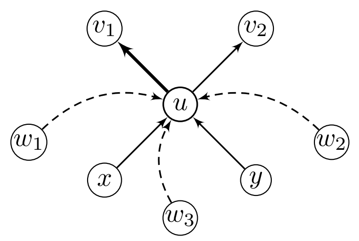
In this example a, b, and c correspond to sensor attributes whereas $@r$ specifies the requested tuple rate (frequency). The goal is to find the cheapest execution plans for the expression set. This also includes deciding how to split the expressions into execution plans p_i, that each produces a stream of tuples T_i at one particular rate r_i. The tree for the expressions E_i can be presented as a graph (Figure 4.20(a)). In the graph we add an edge (dashed arrows) starting



(a) Graph representing set of annotated expressions



(b) Weighted graph model for ILP



(c) Construction of ILP inequalities

Figure 4.20: Graph representations for ILP construction for the example

from every *matching expression* to all expressions it matches. For example, since $(a+b)@2r \preceq (a+b)@4r$, we add an edge from $(a+b)@4r$ to $(a+b)@2r$. The doubly encircled nodes are the root nodes of the expressions E_i. Values corresponding to these root nodes are requested by some user query. □

Graph Modeling. The graph in Figure 4.20(a) can be extended as shown in Figure 4.20(b). We refer to call this extended digraph $G = (V, E)$ consisting of a set of vertices V and edges E. An artificial root node t is added to the graph. The expressions are grouped by their rate r_i. Also, for each group a new node t_i is introduced. Since for each rate (corresponding to a network execution plan or bytecode program) a fixed header cost has to be spent regardless of the number of selected attributes, a weighted edge (t, t_i) is added between the artificial root node t and the common group nodes t_i. The edge weight is set as $c(t, t_i) = r_i C_m$ and accounts for the header cost C_m when using a plan at rate r_i. Additional edges (t_i, u) are added between the group nodes t_i and the remaining nodes u of the expressions. These edges are weighted by the costs $w(t_i, u) = r_i C_a$. The weights reflect the cost of a field in the result message. Note, since $C_m \gg C_a$, as soon as a rate is selected, adding fields to a tuple is relatively cheap. All other edges, including the "matching expression" edges have zero weight. The idea is that evaluating an expression from an expression that is reached over a zero-weight edge is free. The operand values can be retrieved from the tuple that is sent to the base station. When following dashed lines (see Figure 4.20(b)), costs are actually saved as redundancy in the expression set is exploited.

The goal is to select a subset $S \subseteq E$ of edges such that the corresponding induced graph is connected, and all expression roots E_i can be reached from t. For optimality, the minimum cost subset S is of interest. This leads to the following constraint optimization problem:

$$\min_S \sum_{(u,v)\in S} c(u,v) \tag{4.12}$$

0-1 Integer Linear Programming. The optimization problem can be mapped to an *0-1 Integer Linear Programming Problem* (ILP). A 0/1 variable x_e is introduced for each edge e. The edge $e \in S$ if and only if $x_e = 1$. The objective function is expression (4.12). Connectivity and reachability from t is modeled using the constraint inequalities. The procedure is as follows:

Consider a node u (shown in Figure 4.20(c)). This node represents a binary operator, e.g., $x + y$. Hence, it is connected to the two operand nodes x and y. In order to evaluate the expression the edges (x, u) and (y, u) have to be selected, i.e., the operands x and y have to be available. The dashed edges from w_1, w_2, and w_3 are from "matching expressions" that provide $x + y$, at a rate that is an integer multiple of the one requested by u. Node u also has two outgoing edges to v_1, and v_2. The connectivity rules for the inner nodes are given by the following condition.

$$\begin{aligned}(u, v_i) \in S \qquad &\Longleftrightarrow \\ &((x, u) \in S) \wedge ((y, u) \in S) \vee \bigvee_{w_j} ((w_j, u) \in S)\end{aligned} \tag{4.13}$$

In other words, an outgoing edge (u, v_i) is in S if and only if the operand edges $(x, u), (y, u)$ are in S or u is reachable by at least one "matching expression" edge $(w_j, u) \in S$. Note that the condition must hold for each outgoing edge (u, v_1) and

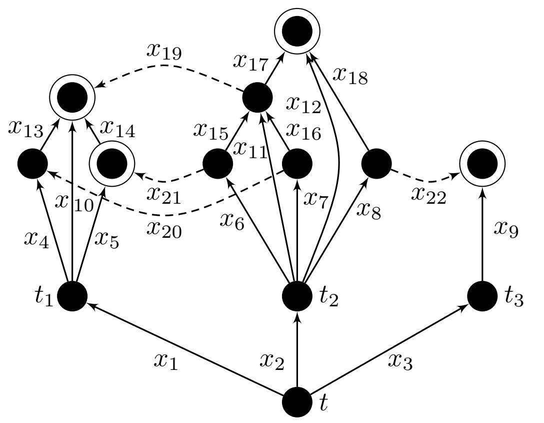


Figure 4.21: Variables $x_e = 1$ correspond to selected edges $e \in S$ in ILP

(u, v_2). The constraints for expression roots (double circled nodes in Figure 4.20) introduce a similar constraint: either all operand edges have to be in S or at least one “matching expression” edge.

In a second step, the set of conditions (4.13) can be rewritten as linear inequalities by introducing a structural variable $x_{(u,v)}$ for each edge (u, v). The equation corresponding to expression (4.13) is:

$$x_{(x,u)} + x_{(y,u)} \; + \; 2 \sum_{w_j} x_{(w_j,u)} - 2x_{(u,v_i)} \geq 0 \tag{4.14}$$

It can easily be verified that inequality (4.14) corresponds to (4.13) by substituting the structural variables x by values 0 and 1. The number of variables is equal to the number of edges $|E|$. For each out-bound edge there is a constraint inequality. An additional inequality is obtained for each root node of the expressions. The objective function of the ILP problem to be minimized is expression (4.12).

Example Continued. The assignment of variables x to edges is shown in Figure 4.21. For the example, 22 variables $x_1, \ldots, x_{22} \in \{0, 1\}$ are introduced. The weights w_i are defined as indicated in Figure 4.20(b). The objective function to be *minimized* is

$$z(x_1, \ldots, x_{22}) = \sum_{i=1}^{22} x_i w_i \qquad .$$

The constraints for the expression roots E_i are:

$$\begin{aligned}
x_{13} + x_{14} + 2x_{10} + 2x_{19} &\geq 2 \\
x_5 + x_{21} &\geq 1 \\
x_{17} + x_{18} + 2x_{12} &\geq 2 \\
x_9 + x_{22} &\geq 1
\end{aligned}$$

The remaining constraints are:

$$\begin{aligned}
x_1 - x_4 &\geq 0 \\
x_1 - x_5 &\geq 0 \\
x_1 - x_{10} &\geq 0 \\
x_2 - x_6 &\geq 0 \\
x_2 - x_7 &\geq 0 \\
x_2 - x_8 &\geq 0 \\
x_2 - x_{11} &\geq 0 \\
x_2 - x_{12} &\geq 0 \\
x_3 - x_9 &\geq 0 \\
x_4 + x_{20} - x_{13} &\geq 0 \\
x_5 + x_{21} - x_{14} &\geq 0 \\
x_6 - x_{15} &\geq 0 \\
x_6 - x_{21} &\geq 0 \\
x_7 - x_{16} &\geq 0 \\
x_7 - x_{20} &\geq 0 \\
x_8 - x_{18} &\geq 0 \\
x_8 - x_{22} &\geq 0 \\
x_{15} + x_{16} + 2x_{11} - 2x_{17} &\geq 0 \\
x_{15} + x_{16} + 2x_{11} - 2x_{19} &\geq 0
\end{aligned}$$

The problem is solved, e.g., using the *GNU Linear Programming Kit (GLPK) solver*, and the following optimal integer solution can be found.

$$\begin{aligned}
x_2 = x_6 = x_7 = x_8 = x_{13} = x_{14} = \ldots = x_{21} = x_{22} &= 1 \\
x_1 = x_3 = x_4 = x_5 = x_9 = x_{10} = x_{11} = x_{12} &= 0 \\
z(x_1, \ldots, x_{22}) &= 60
\end{aligned}$$

The selected edges S and the induced graph are shown in Figure 4.22. The solution contains the two "matching subexpression" edges x_{20} and x_{21} for the operands of $a + b@2r$ as well as the edge x_{19} for the expression itself. This can be considered as a sort of redundancy, but since the "matching subexpression" edges have zero

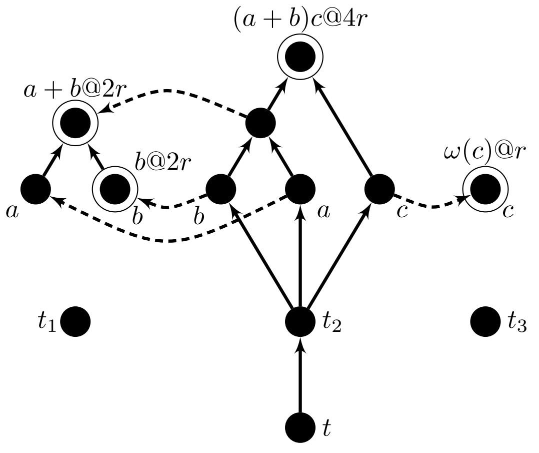


Figure 4.22: Induced graph given by selected edges S, i.e., edges e where $x_e = 1$

weight they do not add a cost. The solution is plausibly minimal in terms of the cost z.

In a second stage the graph is traversed from t in order to determine the expressions that are executed in the network, i.e., the expressions for the tuples fields. This is done by traversing the graph from t and stopping at a node that either is the root of an expression E_i or has an outgoing "matching expression" edge $\in S$. The expression rooted at this stop node is then used for that particular tuple field. The expressions are then assigned to plans by their common rate r_i. Concluding the example, the expressions $E_1,\ldots,E_4$ can be computed using a single plan $p = \{a, b, c\}@4r$. $\square$

Spatial Aggregation Queries. For aggregation queries the mapping from the annotated expressions to the graph has to be extended. As shown by Equations (4.8) and (4.9) (in Section 4.4.3) spatial aggregate plans have smaller costs, as only a single message has to be sent over an edge in the data collection tree. Therefore, both the fixed costs as well as the attribute cost depend on whether the spatial aggregation is performed in-network or at the base station.

The graph model has to reflect these two choices. The model is extended as follows: First, the cost in the graph, i.e., the edge weights, are specified as the total cost including the topology parameter C_T, rather than just r_iC_a and r_iC_m (topology independent). Second, for each rate group r_i that contains at least one spatial aggregation expression, the group node t_i is split into two nodes t_i and $t_{i,\text{agg}}$. If the latter is reachable over an edge $\in S$ the plan will be executed using a spatial aggregation plan. The weights for the edges (t, t_i) and $(t, t_{t,\text{agg}})$ represent the header cost for running a non-aggregation plan at rate r_i and, respectively, an aggregation plan at the same rate. Thus, using the cost metrics from Equations (4.8) and (4.9) the weights are:

$$\begin{aligned} w(t, t_i) &= rC_T C_m \\ w(t, t_{i,\text{agg}}) &= r(N-1)C_m \end{aligned}$$

Next, the aggregation expressions are duplicated in the graph. The second level edges (t_i, a) to attributes model the costs of a field in a result tuple, which is also different for spatial aggregation expressions agg(a). In an aggregation plan, leaf nodes are always aggregation expressions agg(a) and never the single attribute a, as attributes cannot be part of an aggregation plan by definition. The weight costs are as follows:

$$\begin{aligned} w(t_i, a) &= rC_T C_a \\ w(t, f(a)) &= r(N-1)s_f C_a \end{aligned}$$

Here s_f is used to denote the size of the aggregation state of aggregate f. For example: 1 for MIN, MAX, SUM, 2 for AVG, 3 for VARIANCE, and STDDEV, etc. Finally, the graph is extended by the corresponding edges from the non-aggregation component to the aggregation component. As an example consider the following two expressions:

$$\begin{aligned} E_1 &: \quad \max(a)@r \\ E_2 &: \quad (a-b)@10r \end{aligned}$$

In this example, E_1 is a spatial aggregation expression. The optimizer has to decide whether the expressions are to be computed using two separate plans $\max(a)@r, \{a, b\}@10r$, or whether they are merged into a single non-aggregation plan $\{a, b\}@10r$. The graph model is shown in Figure 4.23. The aggregation expression E_1 the group nodes t_r is split into t_r and $t_{r,\text{agg}}$ (dotted rectangle). Also observe the different weights for (t, t_r) and $(t, t_{r,\text{agg}})$. A “matching expression” edge from $a@10r$ to the $a@r$ node of the non-aggregation component of E_1 is added as well. As it can be easily verified that as long as

$$C_T > \frac{1}{10}(N-1)\left(1 + \frac{C_m}{C_a}\right)$$

it is more efficient to run the two plans $\max(a)@r, (a-b)@10r$ than to run a single non-aggregation plan $\{a, b\}@10r$.

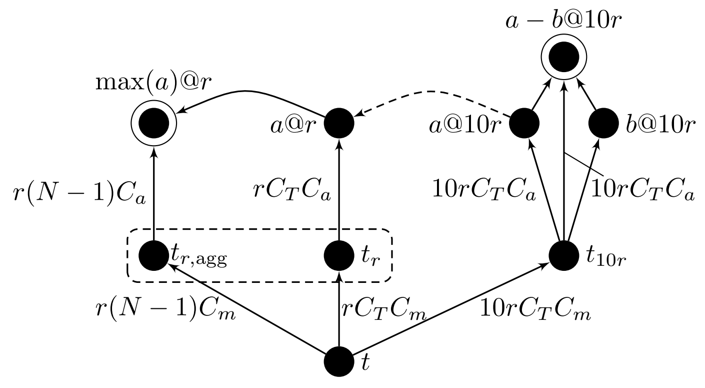


Figure 4.23: Graph for spatial aggregation queries

Limit on the number of concurrent plans. The number of concurrent plans that are generated is equal to the number of selected edges $(t, t_i) \in S$. If the sensor network imposes a limit of at most N_{Pmax}, this limit can be enforced by adding another inequality

$$\sum_i x_{(t,t_i)} \leq N_{\text{Pmax}} \qquad .$$

If this constraint is added, it is in principle possible that the linear program has no feasible solution, i.e., the constraints lead to an empty feasible region. In this case, the optimizer learns that a cost-optimal solution is not possible. Instead, a sub-optimal solution is sought that uses fewer plans but inevitably leads to data not requested by any query. The optimizer iteratively reduces the number of rate groups and, hence, possible plans, by combining groups t_i and t_j until a feasible solution is found. Assume that in group i the set of selected sensors attributes is A_i and A_j for group j then the amount of superfluous data obtained when merging the groups is estimated as

$$\frac{\text{lcm}(r_i, r_j)}{r_i}|A_i \backslash A_j| + \frac{\text{lcm}(r_i, r_j)}{r_j}|A_j \backslash A_i| \quad .$$

The optimizer now greedily combines groups that introduce the least amount of superfluous data until a feasible solution is found.

The ILP-based solution for the min-execution cost optimization problem is NP-hard [Kar72]. We believe that there is no efficient solution for this problem. Nevertheless, for the number of queries that is reasonable to run in a wireless sensor network, this approach is feasible in practice.

4.6.2 Delta-Plan Strategy

For each update set U that is generated when a query is added a new plan will be created. Plans are not reorganized once they are started. The idea for this strategy is to include the update set with as few updates as possible, which is clearly the case if the entire set is added as a new plan. A plan is always removed as soon as no more query is subscribed to an expression of this plan.

This strategy has two problems. First, a plan might produce large amounts of orphan data. Consider, for example, a plan producing tuples that consist of ten attributes. Assume that all queries are removed until the last query, which only extracts one single field of the tuples. This results in resulting in nine orphan attributes in the plan. Unfortunately, the plan is not removed and orphan data can significantly increase the cost. In TTMQO [XLTZ07] Xiang et al. consider this as an advantage, as new queries might request one of the orphan fields. In that case, the queries can be answered directly without any update costs. Xiang et al., unlike in the delta-plan approach, even propose not to strictly remove that contain only orphaned attributes. They are intentionally left in the sensor network for this reason. TTMQO provides a tuning parameter that allows adjusting how aggressively orphaned plans are removed. The second problem of the delta-plan strategy is large number of plans in the system. Because each update set leads to a new plan the number of concurrent execution plans rapidly increase. The approach works only up to the maximum number of concurrent plans. If the limit is reached, the plans have to be reorganized, e.g., using the min-execution cost strategy.

4.6.3 Single-Plan Strategy

This strategy is based on the query merging introduced in Section 4.3 and is motivated by the notion of the *universal query* that returns all attributes of the sensor nodes in the entire network at the highest possible rate. As new queries arrive, the selectivity of the single execution plan is gradually increased towards the universal query (see Section 4.3.1). If a query is withdrawn, a plan is gradually made less selective, i.e., more specific to the queries.

This approach works well for a workload with many concurrent queries, in particular if there is large overlap in the requested attributes. The disadvantage is that the result rate is the least common multiple of all user queries. For example, if the sampling intervals specified in the queries are relative prime, the greatest common divisor is one, leading to a plan with the shortest possible sampling interval. The "tolerant" greatest common sampling period algorithm described in Section 4.3.4 introduces a slack in the resulting sampling interval (a tolerant sampling interval). The optimizer then can freely pick any sampling intervals within the tolerance window specified such that the sampling rate of the resulting execution plan is minimized.

4.7 Evaluation of Strategies

In order to assess the performance of the different optimization strategies a simulation setup is used. Queries are continuously submitted to the query processor and withdrawn at predefined times. Based on the strategy the query processor generates a sequence of network execution plans, which are inserted and withdrawn from the network. The execution cost of each plan is estimated using the cost model introduced in Section 4.4.3. The energy spent for issuing and stopping network execution plans is also included in the model. The cost of all execution plans issued during the evaluation run is summed up into a total execution cost. Lacking a benchmark for query processing in sensor networks, we use a set of randomly generated queries. For each query in the workload, a submission time and execution duration is given.

4.7.1 Query Workload

The queries are selected from a considerably large query space consisting of approximately $2 \cdot 10^{13}$ possible queries. The expressions of the queries are composed using elements that are randomly picked from a predefined set. In this analysis we are not considering selective queries as results depend on the actual sensor values. Furthermore, choosing useful predicates that are repeatable despite the varying nature of the sensor readings is difficult. The workload set consists of three different types of queries: (1) spatial aggregation queries, (2) temporal aggregation queries and (3) non-aggregation queries. Not all queries from the available space are equally likely to occur in practice. A reasonable assumption is that exotic queries such as

$$\begin{array}{rl} \texttt{SELECT} & (temp + light/3)*nodeid \\ \texttt{FROM} & sensors \\ \texttt{EVERY} & 17\,\mathrm{s} \end{array}$$

occur less frequently and that sampling intervals will most likely take values from an "even" range, such as 1 s, 5 s, 15 s, 30,s, 1 min, 5 min, 30 min, 1 h, etc. The query components, number of selection expressions, sensor attributes, and subexpressions are thus selected using a non-uniform distribution. The sampling intervals take one of 17 possible values. We are aware of the fact that sampling intervals that are integer multiples of each other greatly increases reuse of execution plans for multiple queries. Constraining the possible sampling intervals actually corresponds to applying TGCS on unconstrained intervals.

Table 4.3: Query Workloads

Query Workload	**WL1**	**WL2**
non-aggregation queries	50 %	20 %
spatial aggregation queries	30 %	60 %
temporal aggregation queries	20 %	20 %

As shown in Section 4.4.3 the cost for running spatial aggregation queries is very different from the cost for non-aggregation queries. We use two different query workloads with different query mixes (Table 4.3) to investigate these effects. In workload WL1 non-aggregation queries dominate, whereas WL2 is dominated by spatial aggregation queries. For illustration purposes the first 16 queries of the two randomly generated workloads are shown in Table 4.4 (page 145) for WL1 and Table 4.5 (page 146) for WL2.

The submission and withdrawal times of each query are determined using two random models. First, we rely on the common assumption that the query arrivals are Poisson distributed. Second, the time a query is run until it is withdrawn by the user is exponentially distributed. The expected number of queries present in the system in steady state can be estimated by Little's Law. Assuming a query arrival rate λ and an average execution duration D there are λD queries expected in the systems. This does not consider the ramp-up phase at the beginning and the ramp-down phase at the end after the last query has been submitted. A simple calculation shows that the expected number of queries in the ramp up phase is $\lambda D(1 - e^{-\frac{t}{D}})$. For $t \to \infty$ this leads to the asymptotic case of Little's Law. For the ramp-down phase started at t_0 the number of queries decrease as $\lambda D(1 - e^{-\frac{t_0}{D}})e^{-\frac{t-t_0}{D}}$. The system is ergodic during steady state phase, hence, it is possible to estimate statistic characteristics (averages) using a time average, i.e., by choosing a sufficiently large execution window. In the following, it is thus sufficient to consider a single (large) query set for a given set of parameters.

4.7.2 Few Concurrent Queries

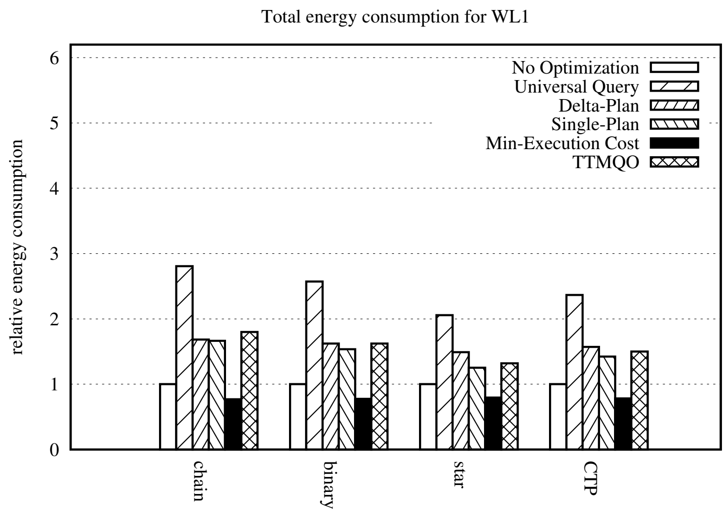
We consider a workload of 10 concurrent queries in average. We select the average arrival rate $\lambda = 1/100$ queries/s and the average execution time $D = 1000$ s. This leads to 10 concurrent queries on average. The number of queries in a workload set is chosen sufficiently large such that the duration of the steady state phase is large compared to ramp-up and ramp-down times. When choosing workload sizes of 200 queries the expected duration of the three phases is distributed as follows: 19 %

for ramp-up, 62 % for steady state, and the remaining 19 % for ramp-down phase. We consider steady state after > 9.9 (expectation) queries are in the system.

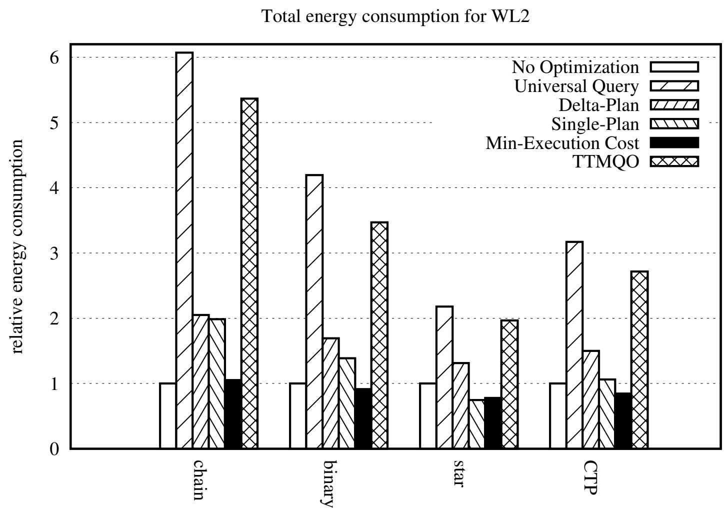
We determine the execution costs for the workloads for four different topologies in a sensor network consisting of 32 battery powered sensor nodes and one base station. The first three are the topologies shown in Figure 4.15. The average hop-distance in the chain is $\bar{h} = 16.5$ graph, in the binary tree ($\bar{h} = 3.375$), and in the star graph $\bar{h} = 1$. The fourth topology is obtained from the dynamic tree routing protocol CTP (see Section 2.2.4, page 23) on the sensor testbed deployment at ETH (Figure 2.1, page 20). The structure established by the protocol with $\bar{h} = \frac{59}{32} \approx 1.8$ is close to the star topology.

We implemented the min-execution cost, delta-plan, and the single-plan strategy, as well as the TTMQO by Xing et al. [XLTZ07]. For the latter we set the aggressiveness parameter to $\alpha = 0.5$. We also determined the execution cost when no optimization would be used by plugging-in the duration and topology into the cost model. We use these values as a baseline for comparison. Additionally, we computed the cost of running the universal query `SELECT * EVERY 1s` for the processing duration of entire workload set. For this evaluation we are considering all costs, i.e., costs for execution, setup and removal. Figure 4.24 shows the total energy relative to the non-optimization scenario for both workloads. A first observation is that all strategies except min-execution cost lead to a higher energy consumption than running each query with a separate plan. As further analysis showed that for the delta-plan and TTMQO strategies few active plans and many orphan expressions remain in the network. For the single-plan strategy, the resulting network plan does have orphan expressions but it runs at a very short interval (1–5 s), already close or equal to shortest possible interval. From Figure 4.24 one can see that switching to the universal query is not beneficial. Compared with the single-plan strategy, the universal query not only runs at a too high sampling rate, it also returns sensor attributes not asked by any query. The figure also shows the influence of the mixture of the query load as well as the topology. The energy costs of suboptimal are worsened by in a deep network, e.g., the chain graph compared to a star.

As expected, in WL2, dominated by spatial-aggregation queries, the improvement of the min-execution cost algorithm is smaller than for WL1. For the chain topology the min-execution cost strategy actually requires slightly (5 %) more energy than the no-optimization strategy. The reason is not completely clear to us. Obviously, the update costs are higher for min-execution cost strategy. In the no-optimization strategy each query is mapped into a separate plan, which is never replaced. For the chain topology, if a spatial aggregation query is executed by a non-aggregation plan, $\bar{h} = 16.5$ more messages are generated than by an aggregation plan. Thus, deciding to do aggregation outside of the network is most heavily



(a) Workload 1



(b) Workload 2

Figure 4.24: Total execution costs for workloads WL1 and WL2 using different strategies and topologies. The costs is shown relative to the no optimization case

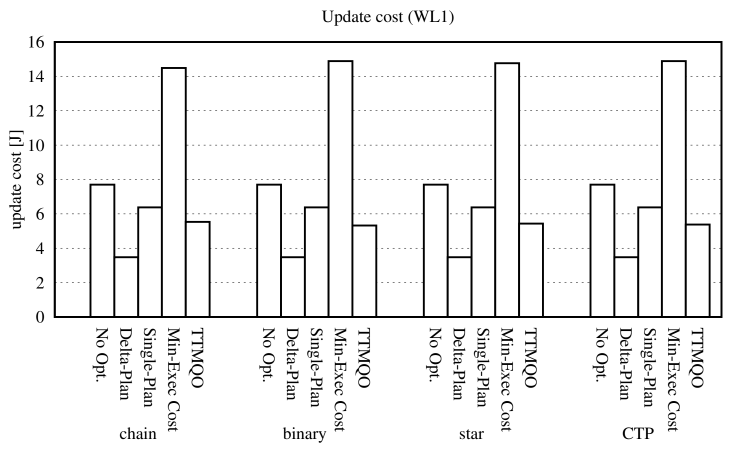


Figure 4.25: Absolute update costs for WL1

penalized in the chain topology. Figure 4.24 also shows that the TTMQO has a very high energy consumption, in particular for WL2.

Figure 4.25 shows the absolute update costs for plan submission and removal when executing WL1. [3] As expected, the aggressive updating of plans by the min-execution cost strategy leads to largest fraction of update costs. The other strategies have lower update costs than the non-optimization case. This is due to the orphan expressions and plans, which suppress creation of new plans when new queries arrive.

Figure 4.26 shows the maximum number of concurrent plans produced by the individual strategies. As described earlier, the delta-plan strategy may lead to a large number of concurrent plans as they are not reorganized. In TTMQO there are relatively few plans in the network but for a long time, which also leads to orphan data and, hence, high energy costs (as seen in Figure 4.24).

4.7.3 Many Concurrent Queries

The results for WL1 and WL2 with 10 concurrent queries on average indicate that the min-execution cost strategy performs well. In order to study the behavior for smaller and larger sets the number of concurrent queries is varied within the interval 1–50. In this run, the 200 queries from WL1 and WL2 with the same execution duration D are used except that the arrival rate λ is chosen such that on average λD queries are in the system.

[3] The plot for WL2 is omitted as it has similar update cost, although, as Figure 4.24 indicates, different total costs.

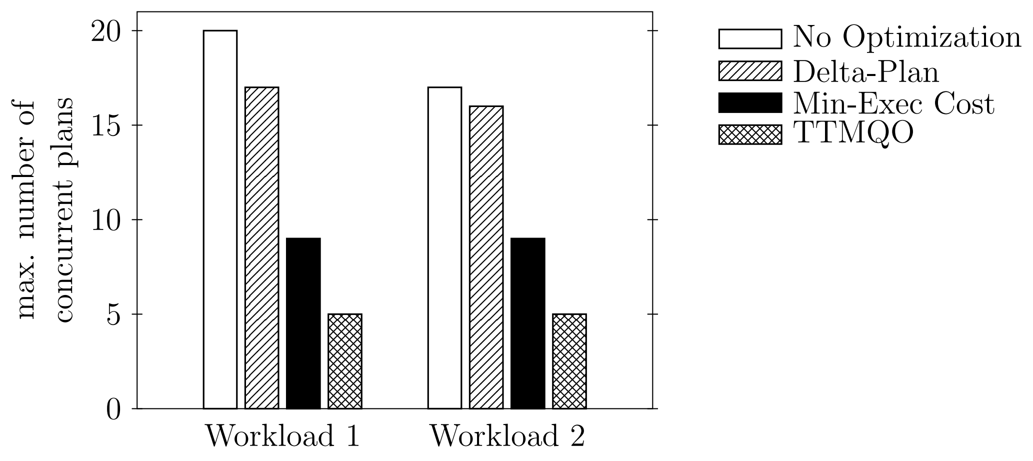
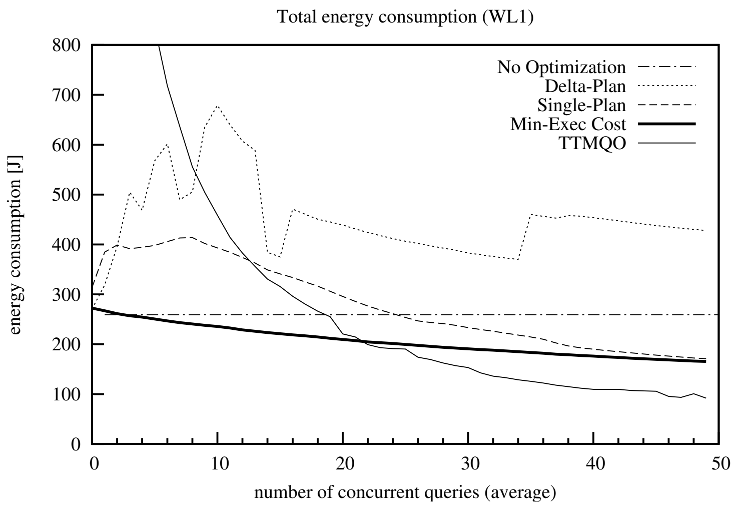


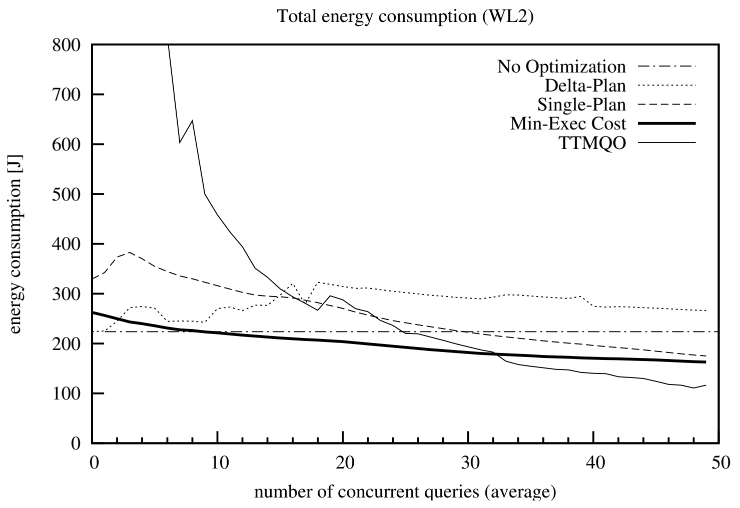
Figure 4.26: Maximum number of concurrent plans

Figure 4.27 shows the total execution costs for each of the optimization strategies for query loads WL1 and WL2. For just a few queries (around less than 5), running the optimizer does not pay off. The queries should be run independently. The delta-plan strategy obviously has a bad performance over the entire interval, although it works better for WL2. Thus, a strategy that solely tries to minimize update costs is not sufficient. The single-plan strategy becomes very efficient as the number of queries increase. The TTMQO strategy performs exponentially bad for less than 20–25 concurrent queries. Starting with $n \geq 23$ queries for WL1 and $n \geq 33$ for WL2 TTMQO outperforms the min-execution cost strategy. This an artifact of the experiments that actually favors TTMQO as the workloads used in the benchmark generate a constant query load. In this case, orphan expressions can quickly be reused by new queries and safe update costs that have to be paid for the min-execution cost strategy. TTMQO will perform worse on bursty loads or when the query arrival rate is not high enough to quickly reuse orphans. When comparing the graphs from Figure 4.27 only the min-execution cost plan and the single-plan strategies show the same characteristics for both loads.

Optimizer Heuristics. As a consequence from these measurements, the optimizer should choose between (1) no optimization, (2) min-execution cost, and (3) eventually the single-plan strategy. The decision can be made based on the number of queries that are present in the system. For a few concurrent queries it should generate a separate plan for each query. For $5 < n \leq 30$ the min-execution cost strategy should be applied. For $n > 30$, the best strategy is probably the single-plan as it does not involve any optimization overhead. TTMQO can be used for a large number of queries if the load is of the right type, but its effectiveness is



(a) Workload 1



(b) Workload 2

Figure 4.27: Total energies for different numbers of concurrent queries (arrival rates) using CTP topology

highly dependent on the query arrival rate, the aggressiveness factor and the speed at which orphan expressions can be reused. It is not a general solution. Switching from min-execution to a different strategy for larger n has the advantage that it prevents the optimization problem from being intractable. For large n the ILP problem that results from the min-execution cost strategy is hard to compute. By avoiding large problem instances difficulties of the NP-hardness of ILP are not observed in practice.

4.8 Related Work

Processing of continuous queries is a well known problem in data streaming systems. Madden et al. [MSHR02] propose an adaptive multi-query processing system CACQ for streams that is based on Eddies [AH00] from conventional database systems. Our approach is different, as in contrast to traditional data streaming systems, data streams in sensor networks are pulled into the stream processing engine rather than pushed. In our case, execution plan generation also involves creating the data stream at the source nodes in the sensor network.

Crespo et al. describe query merging [CBGM03] in a publish-subscribe system in a multi-cast environment. They provide a solution for a “battlefield awareness and data dissemination” scenario where several client submit subscriptions for events with geographically overlapping ranges. Although completely different in nature, there is a similar trade-off between processing inside the network and outside (at the clients) for query areas.

Multi-query optimization for sensor networks for spatial aggregation queries is shown by Trigoni et al. [TYD$^+$05]. Query execution is performed in rounds, where the plans are recomputed and disseminated every round. Queries and sensors are represented in a vector space. This representation leads to a reduction of the amount of data transmitted if the concurrent queries are linearly dependent. The number of queries that are actually processed is equal to the number of dimensions of the subspace spanned by the query vectors. The linearity property leads to a reduction of transmission costs for aggregation queries. The evaluation in the paper is limited to queries having the same aggregate and the same sampling frequency. Additionally, the representation chosen makes it difficult to use predicates other than on node IDs.

Xiang et al. [XLTZ07] propose a two-way multi-query optimization system called TTMQO. The first stage is performed at the base station by query rewrite in order to reduce redundancy in the executed queries. The second stage is performed in the sensor network using query-aware routing and the properties of the broadcast medium. The work does not consider the network topology. They state that the topology is hard to predict and the authors opt for a lower bound for transmitted messages, essentially assuming in-network aggregation or a star topology. Additionally, they do not consider splitting queries. We implemented the TTMQO strategy. The results using our implementation indicate that this strategy is only applicable when the query load contains many concurrent queries and there is a constant stream of new queries. Otherwise, the strategy does not pay off and leads to worse performance than if no optimization is used.

Silberstein and Yang [SY07] discuss the efficient evaluation of multiple spatial aggregates on a subset of source nodes is discussed. In contrast to our setup the aggregate values are not sent to the base station. Instead, they are sent to destination nodes inside the network, e.g., to control actuators. The proposed solution minimizes communication costs while the sharing of partially aggregated values is maximized. In their work, they assume a multi-cast tree rooted at every destination node that connects the corresponding source nodes. This is a much stronger requirement for the routing layer than the single collection tree.

4.9 Summary

In this chapter, we introduced a data model for pull-based streaming queries in sensor networks and defined corresponding data processing operators. The model supports both aggregation (spatial and temporal) and non-aggregation streaming queries. We tackle the problem of efficiently executing multiple concurrent queries from a dynamic workload by performing multi-query optimization. An energy-based cost model was presented, which served as the optimization metric for multi-query optimization. The parameters for the model were identified from a real sensor platform.

Multi-query optimization for sensor networks is different than in traditional streaming systems where data from external sources is pushed into the system. In our discussion we explicitly include data generation by sampling sensors. A merging approach is used to combine related queries in order to reduce the number of queries that are executed by the resource constrained sensor nodes and avoid the acquisition of redundant data. Query merging, however, also introduces problems. Depending on the sampling periods specified in the individual queries, the resulting period after merging can be substantially higher than for the original queries. We proposed TGCS, a method for computing the common sampling period when a bounded error on inter-tuple timing can be tolerated by the user.

The two-tiered query processing of SwissQM generates two execution plans. One is executed in the network and contains the operations that are pushed into the network while the other runs in a traditional stream processing engine at the base station. Based on the cost model, several strategies are presented that partition the work between gateway and sensor network. We showed that different

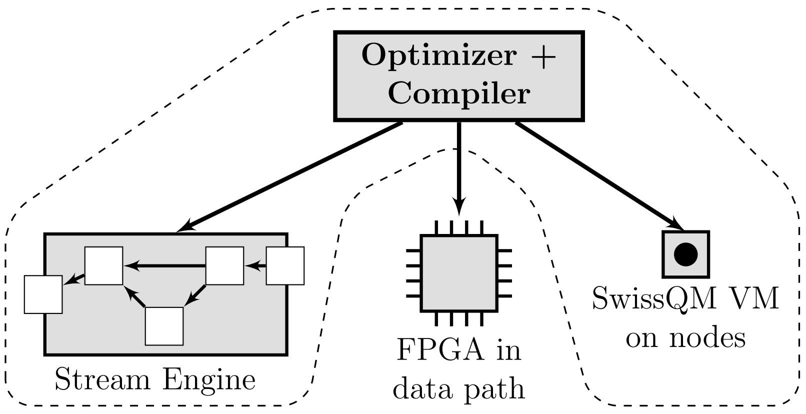


Figure 4.28: Heterogeneous stream engine consisting of SwissQM virtual machine and gateway component extends query processing to the data source

strategies have different effects on the total execution cost. Furthermore, there is no single strategy that performs best for any number of concurrent queries. Instead, the query optimizer should choose the best strategy based on the multi-programming level. A set of heuristics was presented to guide the optimizer to pick the best strategy. We discussed the min-execution cost strategy, which aggressively reorganized the plans using the cost model and 0-1 integer linear programming. This strategy performs well across a wide range of multi-programming levels and takes into account important factors like network topology.

The SwissQM virtual machine and gateway consisting of a compiler that uses optimization techniques described in this chapter represents a complete query processing platform as shown in Figure 4.28. Queries including user-defined functions can be submitted and are partitioned onto the sensor network and the stream engine. In the second part of the dissertation these concepts are extended onto the FPGA platform.

Table 4.4: First 16 queries of Workload WL1

ID	Query
0	`SELECT humidity/temp, temp, light*temp, light EVERY 2min`
1	`SELECT TWINDOW(temp, 3, MEAN) EVERY 60s`
2	`SELECT SWINDOW(voltage, 20, MIN) EVERY 30s`
3	`SELECT light/temp EVERY 5min`
4	`SELECT temp+light, light-temp, light EVERY 2min`
5	`SELECT lightpar*parent EVERY 60s`
6	`SELECT depth/light,temp,parent,humidity/lightpar EVERY 2min`
7	`SELECT SWINDOW(parent, 3, SUM) EVERY 5min`
8	`SELECT depth EVERY 2s`
9	`SELECT parent*parent, humidity EVERY 30s`
10	`SELECT COUNT(*) EVERY 30s`
11	`SELECT AVG(light) EVERY 5min`
12	`SELECT SUM(humidity) EVERY 2min`
13	`SELECT MIN(humidity) EVERY 2min`
14	`SELECT MAX(depth) EVERY 2min`
15	`SELECT temp/humidity, lightpar EVERY 5s`

Table 4.5: First 16 queries of Workload WL2

ID	**Query**
0	`SELECT TWINDOW(humidity, 5, SUM) EVERY 15s`
1	`SELECT light EVERY 15s`
2	`SELECT MIN(parent) EVERY 10s`
3	`SELECT COUNT(*) EVERY 30s`
4	`SELECT COUNT(*) EVERY 15s`
5	`SELECT COUNT(*) EVERY 30s`
6	`SELECT TWINDOW(depth, 5, SUM) EVERY 5s`
7	`SELECT AVG(light) EVERY 60s`
8	`SELECT nodeid, depth-depth EVERY 30s`
9	`SELECT temp, temp EVERY 30s`
10	`SELECT MAX(nodeid) EVERY 4s`
11	`SELECT parent, depth, depth EVERY 2s`
12	`SELECT parent*parent, humidity EVERY 30s`
13	`SELECT COUNT(*) EVERY 30s`
14	`SELECT AVG(light) EVERY 5min`
15	`SELECT SUM(humidity) EVERY 2min`

Part II

Data Stream Processing on FPGAs

5
Field Programmable Gate Arrays

Taking advantage of specialized hardware has a long tradition in data processing. Some of the earliest efforts involved building entire machines tailored to database engines such as the DIRECT database machine by DeWitt [DeW79]. More recently, graphic processors (GPUs) have been used to efficiently implement certain types of operators, e.g., sorting [GGKM06] or general query processing [GLW+04].

Parallel to these developments, computer architectures are quickly evolving toward heterogeneous many-core systems. These systems will soon have a (large) number of processors, e.g., Intel's Single-chip Cloud Computer (SCC) [HDH+10]. Not all processors will be identical. Some will have full instruction sets, while the instruction set of others will be reduced or contain specialized instructions. The heterogeneous cores may operate at different clock frequencies or exhibit different power consumption. Floating point and arithmetic-logic units will be available in different numbers on the cores. Some cores will not have a floating point unit, while others may have additional units such as specialized vector units. An example of such a heterogeneous system is the Cell Broadband Engine [GHF+06], which contains, in addition to a general-purpose core, multiple special execution cores (synergistic processing elements, or SPEs).

It is possible that reconfigurable hardware will be available in the form of highly specialized cores [GS08,MVB+09]. Reconfigurable hardware provide a number of configurable logic primitives and a flexible interconnect in-between. These resources can be used to implement very efficient application-specific processing cores. Essentially, the same ideas already discussed in the context of application-specific virtual machines in Chapter 3 are also applicable. Reconfigurability may be offered at different levels of granularity. *Field-programmable gate arrays* (FP-GAs) provide reconfigurability at the gate level. *Massively parallel processor arrays* (MPPAs) such as the Ambric Am2045 [But07] offer hundreds of very small CPU cores and memory elements on a single chip. The CPUs can be programmed independently and connected through a reconfigurable communication interconnect. MPPAs offer reconfigurability at a coarser level than FPGAs. The processor arrays, however, have not been widely use so far. In contrast, the FPGA market has a significant size. McGrath estimates the FPGA market to exceed $2.7 billion by 2010 [McG06]. FPGA have been used since the 1990s in the embedded system space. A few years ago they started being used for high-performance computing applications.

Another reason besides the investigation of the potential the FPGAs offer for general purpose computing are the current limitations of modern CPU architectures. The limitations are well known: high *power consumption*, *heat dissipation*, *network bottlenecks*, and the *memory wall*. These problems add up when the CPU is embedded in a complete computer. For instance, if applications are not carefully designed, CPUs can spend much of their time waiting for data from memory or disk. Getting data in and out of the system often results in high *latency*, to the point that any algorithmic advantages may become irrelevant. In addition, a modern server CPU consumes over 100 Watts of *electrical power*, not counting necessary peripherals such as memory, disks, or cooling equipment. In the search for possible solutions, FPGAs have been proposed as a way to extend existing computer architectures. They add processing elements that help alleviate or eliminate some of these problems. FPGAs are particularly interesting today because they can be either added as additional processing cores in heterogeneous multi-core architectures [GS08, Kic09] and/or embedded in critical data paths (network-CPU, disk-CPU) to reduce the load and amount of data that hits the CPU [Net09].

What makes FPGAs interesting for designing data processing systems is that they are not bound to the classical von Neumann architecture. Thus, they can be used to avoid the memory wall, to implement highly parallel data processing, and to provide support that would be very expensive otherwise, *e.g.*, content-addressable memory. They can also guarantee extremely low latencies and high throughput rates. For instance, they can process data from the network at *wire-speed*, without having to bring it to memory and the CPU first. In addition, and not least important these days, FPGAs feature a far lower power consumption than CPUs, making them ideal complements to general-purpose CPUs in many-core architectures.

In the second part of this dissertation we focus our attention on FPGAs and their use as a data processing platform. It is as yet unclear how the potential of FPGAs can be efficiently exploited. In this chapter we introduce FPGAs. The next chapter discusses properties of the FPGAs as a computing platform. We use sorting networks that are well suited for an implementation in hardware to illustrate the trade-offs in designs for FPGAs. Chapter 6 also provides a set of guidelines for how to make design choices. In Chapter 7 we present a compositional approach to translate streaming queries into digital circuits for FPGAs. Essentially, we provide a similar tool as the query-to-bytecode compilation in SwissQM. The chapter will describe *Glacier* a hardware component library for stream operators and a compiler that translates queries into hardware circuits by instantiating components from the library. Chapter 7 also presents a new operator for window-based stream joins and its implementation on FPGAs.

5.1 Related Work

A number of research efforts have explored how databases can use the potential of modern hardware architectures. Examples include optimizations for cache efficiency (e.g., MonetDB [MBK00]) or the use of vector primitives ("SIMD instructions") in database algorithms [ZR02]. The QPipe [HSA05] engine exploits multi-core functionality by building an operator pipeline over multiple CPU cores. Likewise, stream processors such as Aurora [ACc+03] or Borealis [AAB+05] are implemented as networks of stream operators. An FPGA with database functionality could directly be plugged into such systems to act as a node of the operator network.

The shift toward an increasing heterogeneity is already visible in terms of tailor-made graphics or network CPUs, which have found their way into commodity systems. Govindaraju et al. demonstrated how the parallelism built into graphics processing units can be used to accelerate common database tasks, such as the evaluation of predicates and aggregates [GLW+04].

The use of network processors for database processing was studied by Gold et al. [GAHF05]. The particular benefit of such processors for database processing is their enhanced support for multi-threading. We share our view on the role of FPGAs in upcoming system architectures with projects such as Kiwi [GS08] or Liquid Metal [HHBR08]. Both projects aim at off-loading traditional CPU tasks to programmable hardware. Mitra et al. [MVB+09] recently outlined how FPGAs can be used as co-processors in an SGI Altix supercomputer to accelerate XML filtering.

The advantage of using customized hardware as a database co-processor is well known since many years. For instance, DeWitt's DIRECT system comprises of a number of query processors whose instruction sets embrace common database tasks such as join or aggregate operators [DeW79]. Similar ideas have been commercialized recently in terms of database appliances sold by, e.g., Netezza [Net09],

Table 5.1: Selected characteristics of FPGAs chips used in this dissertation

	Virtex-5		Virtex-6	
	LX110T	FX130T	LX550T	LX760
lookup tables (LUTs)	69,120	81,920	343,680	474,240
flip-flops (1-bit registers)	69,120	81,920	687,360	948,480
slices (4 LUTs, 4/8 flip-flops)	17,280	20,480	85,920	118,560
block RAM (36 kbit blocks)	148	298	632	720
25 × 18-bit multipliers	64	320	864	864
PowerPC cores	–	2	–	–
I/O pins	800	840	1,200	1,200
release year	2006	2006	2009	2010

Kickfire [Kic09], or XtremeData [Xtr09]. All of them appear to be based on specialized, hard-wired acceleration chips, which primarily provide a high degree of data parallelism. Our approach can be used to exploit the *reconfigurability* of FPGAs at runtime. By reprogramming the chip for individual workloads or queries, we can achieve higher resource utilization and implement data *and* task parallelism. By studying the foundations of FPGA-assisted database processing in detail, this work is an important step toward the goal of building such a system.

5.2 Overview of FPGAs

Field-programmable gate arrays (FPGAs) are re-programmable hardware chips for digital logic. FPGAs provide logic gates on a 2D array that can be configured to construct arbitrary digital circuits. FPGAs, informally sometimes referred to as "programmable logic", are general-purpose hardware chips. In contrast to ASICs (application-specific integrated circuits), FPGAs have no pre-determined functionality. Rather, they can be configured to implement arbitrary logic by combining gates, flip-flops, and memory elements. Initially, FPGA were designed to be used for prototyping ASICs. By using FPGAs instead custom silicon chips (ASICs) design costs and the time-to-market can be significantly reduced.

The circuits are specified using either circuit schematics or hardware description languages such as Verilog or VHDL. A logic design on an FPGA is also referred to as a *soft IP-core* (intellectual property core). Existing commercial libraries provide a wide range of pre-designed cores, including those of complete CPUs. More than one soft IP-core can be placed onto an FPGA chip.

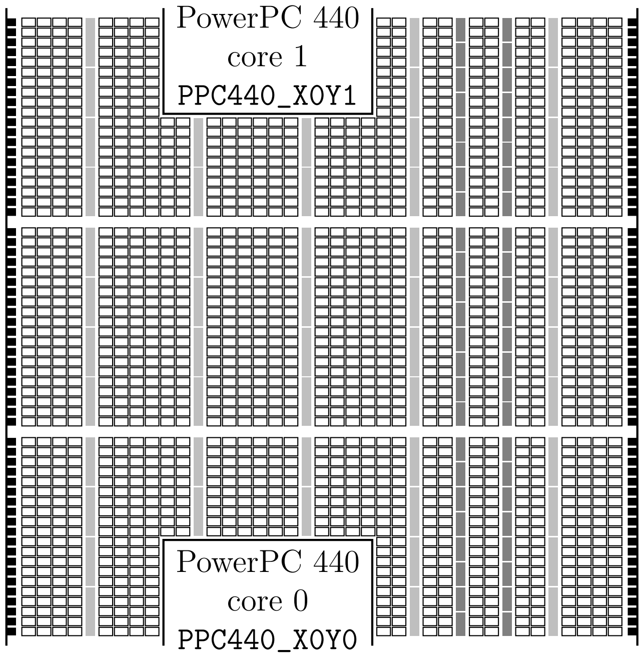


Figure 5.1: Simplified FPGA architecture: 2D array of CLBs ▭ each consisting of 2 slices. IOBs ■ connect the FPGA fabric to the pins of the chip. Additionally available in silicon are: two PowerPC cores, BRAM blocks ▮ and multipliers ▮ .

5.2.1 FPGA Architecture

Figure 5.1 sketches the chip layout of the Xilinx Virtex-5 FX130T FPGA [Xil09a, Xil09b]. The FPGA is a 2D array of *configurable logic blocks* (CLBs). Each logic block consists of 2 *slices* that contain logic gates (in terms of lookup tables, see below) and a switch box that connects slices to the FPGA *interconnect fabric*.

In addition to the CLBs, FPGA manufacturers provide frequently-used functionality as discrete silicon components, called *hard IP-cores*. Such hard IP-cores include *block RAM* (BRAM) elements, each containing 36 kbit fast dual-ported memory cells, as well as 25×18-bit multiplier units. A number of *input/output blocks* (IOBs) link to pins, e.g., used to connect the chip to external RAM or networking devices. Two on-chip PowerPC 440 cores are directly wired to the FPGA fabric and to the BRAM components. Each PowerPC core has dedicated 32 kB data and instruction caches. The caches have similar latency as the BRAM memory and are intended to speed-up accesses to external memory with longer latency. The superscalar cores implement the 32-bit fixed-point subset of the PowerPC architecture. The embedded PowerPC 440 cores are also used in the IBM Blue Gene/L supercomputer where they perform all non-floating point operation.

Table 5.1 shows a summary of the characteristics of the FPGA used in this dissertation. The table shows both Virtex-5 and Virtex-6 devices. The Virtex-6 is the latest Xilinx device family. For experiments only Virtex-5 devices were used due to the lack of Virtex-6 devices. Designs that used the Virtex-6 chips where simulated using the Xilinx simulation tools. In VLSI design it this is a acceptable method to estimate the performance characteristics of a circuit. Table 5.1 also illustrates the improvements between devices and device families. The largest chip of the Virtex-6 family the LX760 provides 6.9× more lookup tables and 13.7× more flip-flops than the four-year-old Virtex-5 LX110T used in the experiments.

Configurable Logic Blocks (CLBs) are further split into *slices*. On the Virtex-5 each CLB is made up of two slices. Each slice contains four *lookup tables* (LUTs) and four *flip-flops*. Figure 5.2 depicts one of the four LUT–Flip-flop configurations a Virtex-5 slice. LUTs can implement arbitrary Boolean-valued functions that can have up to six independent Boolean arguments. Traditionally, a LUT has one output. On the Virtex-5 a LUT has two outputs (identified as O5 and O6 in Figure 5.2). A Virtex-5 LUT either implements a single function on output O6 that uses up to six inputs or, alternatively, two functions on O5 and O6 that in total use five inputs. The outputs O5 and O6 are fed to two multiplexers that configure which signals appear on the output of the slice and are fed to the flip-flop. The flip-flop acts as a register that can store one single bit. The design of a slice provides dedicated elements for carry logic that allow an efficient implementation of, e.g., adders and comparators. The carry logic connects the LUTs inside a slice and different slices in an FPGA column.

Certain LUTs on the Virtex-5 can also be used as 16- or 32-element shift registers or as 32 × 1-bit or 64 × 1-bit RAM cells. Memory instantiated through LUTs configured as RAM cells is referred to as *distributed memory*. In contrast to the aforementioned block RAM (BRAM) distributed memory can be instantiated on finer scale, however, at a significantly lower density.

5.2.2 FPGA Design Flow

Digital circuits for are typically specified using a *hardware description language* (HDL) although there are also specialized languages, for example, Lola [Wir98, Wir96] by Niklaus Wirth used in education or Lava [BCSS99] by Per Bjesse et al. to express layouts. The most commonly used languages are *Verilog* and *VHDL*. There are also high-level programming approaches to FPGAs such as Kiwi [GS08] and various C-to-hardware compilers. In this work we are studying the direct impact of the algorithms on the resource consumption on the chip. We therefore preferred the low-level HDL programing approach to better understand the behavior of the designs and the FPGA.

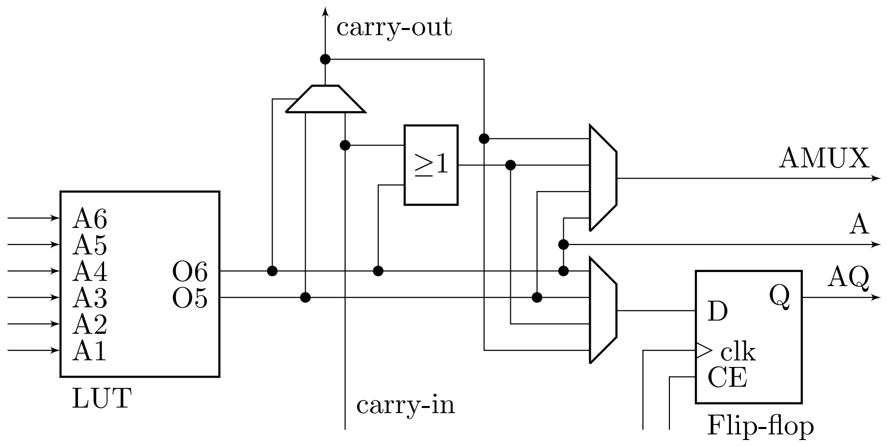


Figure 5.2: Simplified LUT–Flip-flop combination of a Virtex-5 slice. A slice contains four of these structures.

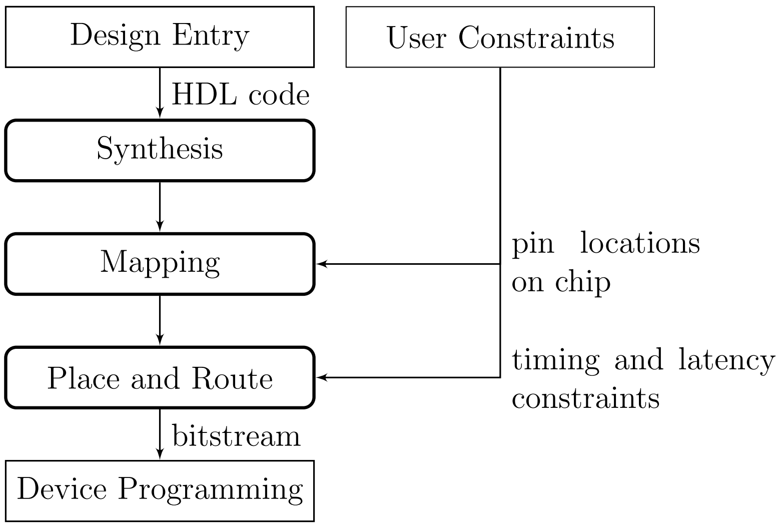


Figure 5.3: FPGA design workflow

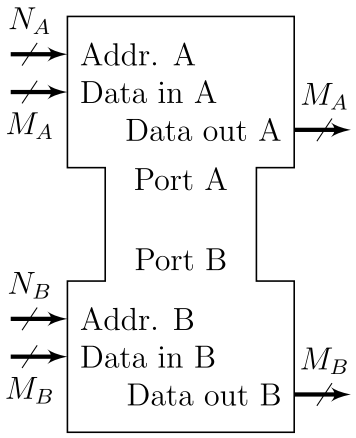
The HDL sources are compiled through a tool chain to a *bitstream* file that is the programmed onto the chip and defines the function of the configurable hardware components, such as LUTs and the interconnect fabric of the chip. The workflow is depicted in Figure 5.3. The HDL sources are compiled into a device-independent format. The following *mapping* stage maps the device-independent primitives into device resources. During *place and route* (PAR) the resources are mapped onto chip locations and the signal routing in the interconnect determined. Finally, the fully routed design is serialized into a binary format, i.e., bitstream. At different stages in the design flow non-functional specifications can be added. These external *constraints* are considered in the workflow and include, for example, pin locations, in order to make sure that the I/O ports of the FPGA circuits match the hard-wired signals to the chip pins. Timing constraints are added to force the tools to produce a design that meets well-specified clock and latency requirements. In the following chapter we will make heave use of timing constraints.

5.2.3 Hardware Setup

FPGAs are typically available pre-mounted on a circuit board that includes additional peripherals. Such circuit boards provide an ideal basis for the assessment we perform in this work. Quantitative statements in this dissertation are based on two different hardware platforms.

Xilinx XUPV5-LX110T Development Board. This development board is distributed as part of the Xilinx University Program. The board design is based on a Xilinx ML505 evaluation board. For the XUPV5 edition the LX50T chip of the ML505 board is replaced by a larger LX110T chip. The chips are architecture-wise identical but the LX110T provides more than twice as many logic gates and thus can be used for larger designs. The FPGA is mounted on a PCI-Express printed circuit board and contains an on-board 256 MB DDR2 RAM module. For communication, one gigabit Ethernet port and a serial RS-232 UART interface are available. For universities the board is sold for $750.

Xilinx ML510 Development Board. The ML510 development board contains a Virtex-5 FX130T FPGA chip. It not only provides more configurable logic gates than the LX110T but also contains two embedded PowerPC cores. The board has an ATX form factor and a DDR2 DIMM socket, which we populated with two 512 MB RAM modules. For terminal I/O of the software running on the PowerPC, a RS-232 UART interface is available. The board has two gigabit Ethernet ports. The ML510 board costs $3,995.



(a) 36 kbit Block RAM

Address width N	**Data width** M	
15	1	32 k × 1
14	2	16 k × 2
13	4	8 k × 4
12	9	4 k × 9
11	18	2 k × 18
10	36	1 k × 36

(b) Port configurations

Figure 5.4: Dual-ported Block RAM (BRAM) element on Virtex-5

Both boards are clocked from a 100 MHz crystal. From this external clock additional clocks are generated for the various clock regions on the chip and for the external I/O connectors, such as the DDR RAM. The PowerPC cores are clocked at 400 MHz. The RAM module, gigabit port and the UART interface are hard-wired to a set of FPGA pins. The FPGA designs are volatile on the chip, i.e., they are lost after a power loss. The designs thus have to be loaded into the chip after each power-up of the board. The bitstreams can be downloaded to the chip through a JTAG programming interface or loaded from a Compact Flash card.

5.2.4 Dual-Ported Block RAM

In several places we will exploit the high configurability of the available Block RAM (BRAM). On Virtex FPGAs, each BRAM block provides 36 kbit of on-chip memory. Unlike in commodity systems, the *word size* of each block can be configured: the 36 kbit of BRAM can be partitioned into, for instance, 1,024 words of 36 bit, 4,096 words of 9 bit, or 32,768 single-bit words. The possible configurations are shown in Figure 5.4(b). In some configurations the full 36 kbit cannot be used. Multiple BRAM blocks can be wired together to obtain larger memories and/or larger word sizes.

All BRAM blocks are *dual ported*. Figure 5.4(a) shows the two ports (A and B) of the BRAM block. The two fully independent ports provide access to the same physical data as truly concurrent operations.[1] Moreover, the word size of both ports can be configured independently; data might be written, *e.g.*, as two 8-bit

[1]The semantics for two conflicting write operations is undefined.

words on one port, later accessed as a single 16-bit word using the other port. We will make use of this feature to implement *content addressable memory* (CAM) in the next section.

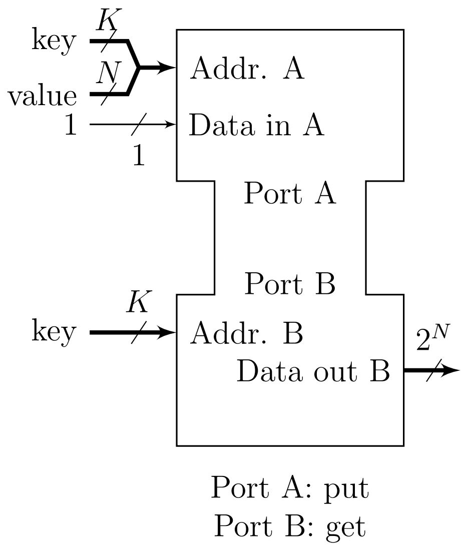
5.3 Content-Addressable Memory

The main advantage of using FPGAs for data processing is their intrinsic parallelism. Among others, this enables us to escape from the *von Neumann bottleneck* (also called the *memory wall*) that classical computing architectures struggle with. In the common von Neumann model, memory is physically separated from the processing CPU. Data is acquired from memory by sending the location of a piece of data, its *address*, to the RAM chip, then receiving the data back. In FPGAs, flip-flop registers and block RAM are distributed over the chip and tightly wired to the programmable logic. In addition, lookup tables can be re-programmed at runtime and thus be used as additional distributed memory. As such, the on-chip storage resources of the FPGA can be accessed in a truly parallel fashion.

A particular use of this potential is the implementation of *content-addressable memory* (CAM). Other than traditional memory, content-addressable memory can be accessed by data values, rather than by explicit memory addresses. Typically, CAMs are used to resolve a given data item to the address it has been stored at. More generally, the functionality can implement an arbitrary key-value store with constant (typically single-cycle) lookup time.

CAM Example. Consider a 10×32 CAM that can store 10-bit keys in 32 different locations. This CAM can be implemented using a single dual-ported 36 kbit BRAM block as shown in Figure 5.5(a). The CAM provides two interfaces: (a) a *put* interface that to store a 10-bit key at one out 32 address locations (value), and (b) a *get* interface to look up *all* address locations for a given key. Port A is configured for the *put* operation while port B is used for the *get*. Port A uses bit addressing The address is determined by concatenation of the key and the value bits. More precisely, the bit address used for port A is $\mathrm{Addr_A} = 32 \times \mathrm{key} + \mathrm{value}$. Assume the key 256 is stored at location one, then address used for port A is $\mathrm{Addr_A} = 32 \times 256 + 1 = 8,193$. Figure 5.5(b) shows the address space through port A after setting key and address location.

For reading, port B is addressed by the key, i.e., each 10-bit address then refers to 32 data bits. Note that both ports refer to the same data. The mapping is chosen such that each bit-address used in port A maps to a bit-position in port B. A read on port B returns *all* locations where the key may be stored (in a bitmap encoding). Figure 5.5(c) depicts the memory space seen through port B. It shows that at address 256 the bit at location 1 is set. Note that due to the one-



(a) BRAM configuration for CAM

addr	bit
0	0
⋮	⋮
8,192	0
8,193	**1**
8,194	0
⋮	⋮
32,767	0

(b) Port A

addr	31	...	2	1	0
0	0	...	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮
255	0	...	0	0	0
256	0	...	0	**1**	0
257	0	...	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮
1,023	0	...	0	0	0

(c) Port B

Figure 5.5: Example CAM with storage capacity for ten 32-bit values. Memory content as seen through the two ports after storing a data item (key = 256 and value = 1).

hot encoding a key can be stored at multiple locations. The data word read on port B can optionally be converted into a binary encoding through an additional combinatorial circuit, e.g., if multiple bits are set, it returns the binary address of the lowest bit set in the data word. □

By combining multiple BRAM blocks the key width as well number of locations, i.e., the depth of the CAM can be increased. We refer to the work of Guccione et al. [GLD00] or the documentation provided by Xilinx [Xil99] for details on FPGA-based CAM implementations. In Section 7.3.7, we use content-addressable memory to implement lookups during 'group by' execution. The access pattern in this context, frequent lookups with rare updates, suggests the use of a CAM implementation that is based on lookup tables. It excels with very high lookup speeds (a fraction of a clock cycle), but has a 16-cycle latency for updates. As an alternative, a block RAM-based implementation would require a full cycle for lookups and two cycles for updates.

5.4 Network Interface

FPGA-based accelerators need to be integrated into a full system solution. Input data needs to transferred FPGA and results read back. One way to achieved this

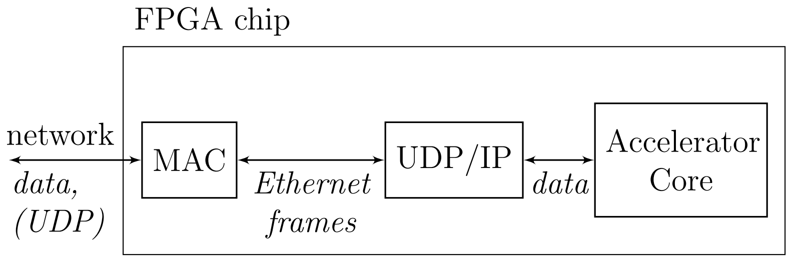


Figure 5.6: Network attachment of accelerator core. The core is connected to a gigabit Ethernet network through a UDP/IP core that is implemented on the FPGA fabric.

is by using a network attachment, i.e., the accelerator on the FPGA is connected a network. The architecture is shown in Figure 5.6. Data is sent to the FPGA board in form of UDP datagrams. A *medium access controller* (MAC) is provided as a hard-IP core on the FPGA. This controller provides direct access to the underlying Ethernet frame. We designed a UDP/IP engine that implements necessary the protocol stack of IP and UDP in order to extract the payload data for received datagrams and that generates the Ethernet frames for the outgoing data. The gigabit network the MAC and the protocol engine are clocked at 125 MHz. Data is transmitted over a 8-bit communication channel, resulting in the necessary on chip bandwidth of 1 Gb. Essentially, this attachment provides direct access to the wire. As we will show later this has important consequences for low-latency processing.

5.5 FPGA Co-Processor

An additional attachment we considered in this work is the use of the FPGA as a co-processor to a conventional CPU. Here we studied the use of the embedded PowerPC core and the direct connection to the FPGA fabric. Additional possible co-processor attachments in PC-based systems are through the PCI-Express Bus [Bit09] or a connection over the HyperTransport Bus. Building a PCI-Express attachment is a very daunting task as a complex protocol needs to be implemented not only in hardware but also as part of the driver in the host system.

5.5.1 Bus Attachment

Similar to traditional microcomputer systems the embedded CPU on the FPGA also requires memory and peripheral components for I/O. The components and the CPU are connected to different on-chip buses. The Xilinx platform uses two different types of bus systems: the *processor local bus* (PLB) and the *on-chip peripheral bus* (OBP). The former is designed to be used to connect comparatively fast components such as CPU, memory, and network chips whereas the latter is used for slow peripheral components, such as terminal UART.

The custom core is treated as any other component of the microcomputer system. It is also connected to bus system. An architecture used later on the ML510 board is shown in Figure 5.7. All components including the CPU are connected to the PLB bus. As small amount of Block RAM is used to store the program that is executed by the CPU. The data to be processed is stored in the larger off-chip DDR2 memory bank. A UART is used to connect to a terminal console. The connection between the external components and the PLB is made through controllers, i.e., a *muli-port memory controller* (MPMC) for the memory and a UART controller for the serial port. The acceleration core is attached over a bus interface called *IP interface* (IPIF), which is provided by Xilinx for all different bus systems. Hence, the developer of the application core does not need worry about the details of the bus protocol. The IPIF essentially maps the accelerator core into an unused memory space such that it been accessed by the CPU.

One processing model we use in later chapters corresponds to the bus attachment that is shown in Figure 5.7. First, input data is loaded into the accelerator core in step (1). Depending on the implementation of the accelerator core, the data can be processed online as is streamed in or it needs to be stored in a small cache-like memory structure in the accelerator. The data is then processed in step (2). The result data is written back to memory in step (3). Signaling to the CPU is either interrupt-based or implemented through polling. The transfers (1) and (3) do not necessarily involve the CPU as the accelerator core can be configured to initiate the transfers. The IPIF provides sophisticated bus operations such as *direct memory access* (DMA) between the accelerator core and the memory.

5.5.2 Attachment to CPU Execution Pipeline

Instead of connecting an accelerator core over the PLB and mapping the core into the main memory seen by the CPU we can implement the accelerator core as an *Auxiliary Processor Unit* (APU). The APU is directly connected to the execution pipeline of the PowerPC 440 CPU. It was designed by IBM as a PowerPC extension to connect a floating-point unit (FPU) to the embedded core. For example, the FPUs of the IBM Blue Gene/L supercomputer are connected to the PowerPC 440 core through the APU interface. The APU does not have access to memory bus but the design benefits from a short communication path to the embedded CPU.

The APU processor is accessed through a set of additional machine instructions. These additional instructions include load/store, floating-point and as well as user-

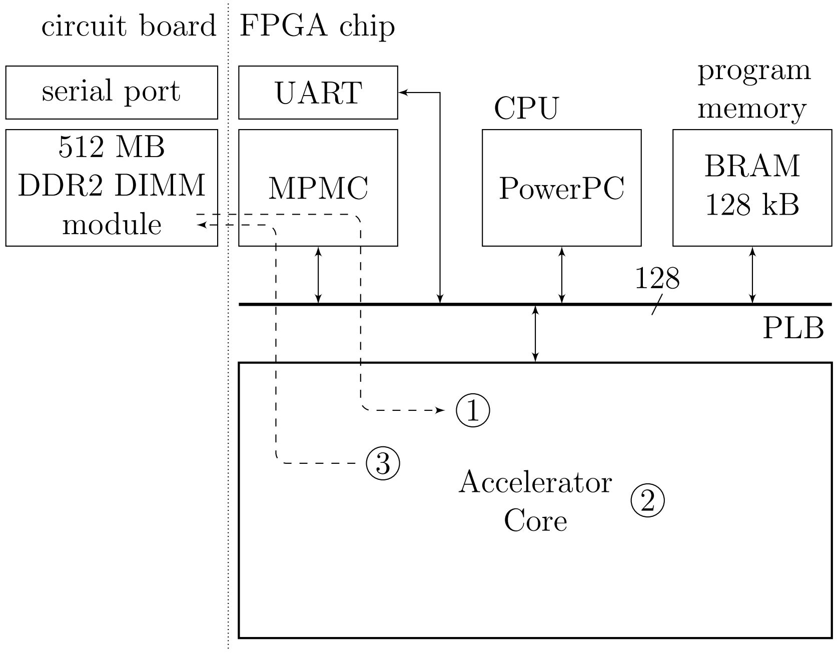


Figure 5.7: Architecture of the on-chip system that uses a bus attachment of the accelerator core.

defined instructions. The APU processor can contain its own register file. The load/store instructions can then be used to transfer data between memory and APU register file. Since the APU is directly connected to the memory unit of the CPU the APU can also benefit from the data cache, which makes sharing data between CPU and APU very efficient. The user-defined instructions can be used to pass data from the CPU register to the APU. Figure 5.8 shows the architecture. The APU consists of a controller implemented in hard silicon and co-located with the CPU core and the custom-defined *fabric co-processor module* (FCM) implemented using FPGA components. The APU connects to the CPU at different points of this simplified 5-stage RISC pipeline. The APU controller first decodes APU-specific instructions (ID stage). Supported instructions are user-defined instructions such as `udi0fcm Rt,Ra,Rb` and FCM load/store instructions. The necessary operands are provided to the FCM during the execution phase (EX stage), e.g., values for the two register operands Ra and Rb or the computed memory address for loads and stores. For user-defined instructions, the FCM computes the result and returns it to the CPU where it is into the target register Rt in the CPU's register file. For load instructions the CPU provides the data from memory up to 16 bytes in parallel. For stores, the data returned by the FCM is written back to the memory. The connection between the APU controller and the FCM is implemented through a well-defined interface that contains the necessary

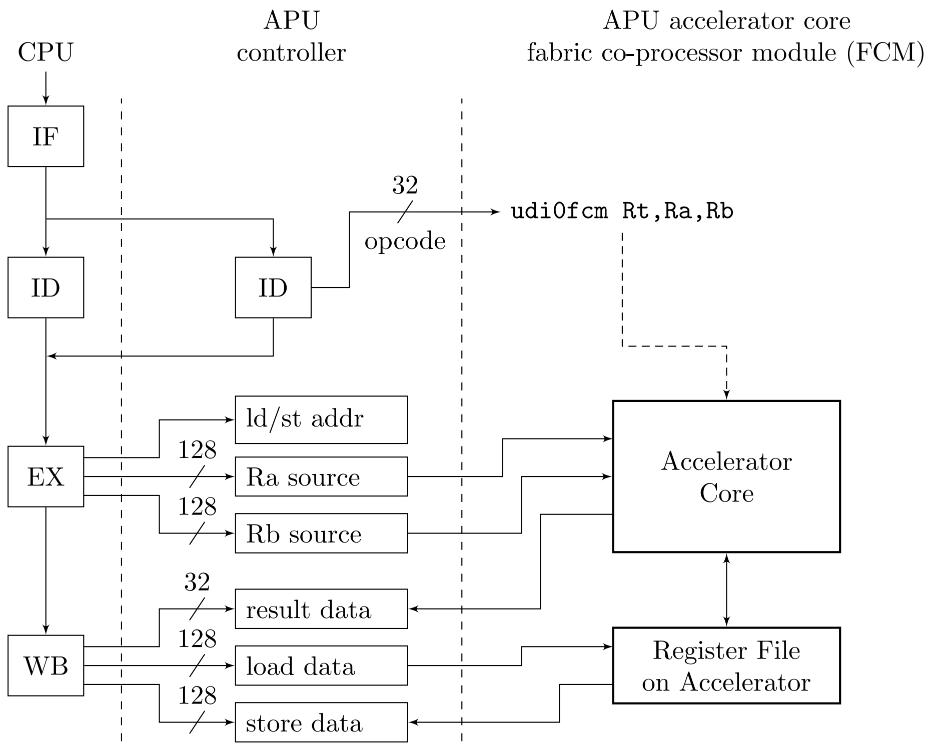


Figure 5.8: Accelerator core connected to *Auxiliary Processor Unit*

signals that allow the CPU-side *APU controller* to control the FCM and exchange data.

5.6 Summary

This chapter provided an introduction into the FPGA technology. An outline of the design flow for was given. Designing an FPGA-based system is very different from building traditional software systems. First of all, circuits have to be mapped onto primitives provided by the FPGAs such as lookup tables and flip-flops, then they are placed on the chip. Finally, on-chip signals paths are routed. Although, this is typically handled by design tools the developer nevertheless has to be aware of this as design choices significantly affect the success of the tools. The chapter also introduced architecture features that are specific to FPGA. For example, content addressable memory (CAMs) can be used as hash tables with fixed guarantees. Furthermore, FPGA can be used in many different configurations of system architectures. In the chapter the use of an FPGA in the network path as well as a co-processor to an conventional CPU is described. The system architectures will turn out to have a major impact on the end-to-end performance of the hardware solution. The next chapter discusses the trade-offs of the attachment methods of FPGA to conventional systems. The following chapter also evaluates FPGAs as a computing platform, i.e., how they can be used to perform computing tasks.

6 Sorting Networks on FPGAs

In this chapter, we first study the design trade-offs encountered when using FPGAs for data processing tasks. In particular, we look at sorting networks that are well suited for an implementation in hardware. Second, we provide a set of guidelines for how to make design choices such as:

(1) FPGAs have relatively low clock frequencies. Naïve designs will exhibit a large latency and low throughput. We show how this can be avoided by a careful design using *synchronous circuits* and circuits that use *combinational logic*. While the former increase throughput the latter reduce latency.

(2) Building combinational circuits is difficult to design than synchronous circuits. This has led to a preference for synchronous circuits in studies of FPGA usage [GS08]. Using the example of *sorting networks*, we illustrate systematic design guidelines to create combinational circuits that solve database problems.

(3) FPGAs provide inherent *parallelism* whose only limitation is the amount of *chip space* to accommodate parallel functionality. We show how this can be managed and present how the chip space consumption of different implementation can be estimated.

(4) FPGAs can be very useful as database co-processors attached to an engine running on conventional CPUs. This *integration* is not trivial and opens up several questions on how an FPGA can fit into the complete architecture. In two use cases, we demonstrate an embedded heterogeneous multi-core setup.

In the first use case we connect the custom logic over a bus to an embedded CPU. The second uses a tighter integration to the CPU, implemented though a direct connection to the CPU's execution pipeline. For both approaches we study the trade-offs in FPGA integration design.

(5) FPGAs are attractive as co-processors because of the potential for tailored design and parallelism. We show that FPGAs are also very interesting in regard to *power consumption* as they consume significantly less power, yet provide at a performance comparable to the one of conventional CPUs. This makes FPGAs good candidates for multi-core systems as cores where certain data processing tasks can be offloaded.

To illustrate the trade-offs in system integration we present two applications scenarios that are based on sorting networks. The first is the implementation of a *median operator*. In the second use case we evaluate a hardware/software co-design on a FPGA. A 8-element sorting co-processor is implemented in the FPGA logic and combined with a merge sort algorithm running on the embedded CPU. Through an extension of the CPU's instruction set we show how the FPGA accelerator can be used in heterogeneous setup together with existing CPU code. Our experiments show that FPGAs can clearly be a useful component in a modern data processing system, especially in the context of multi-core architectures.

Outline. We start our work by setting the context with related work (Section 6.1). After introducing the necessary background on sorting networks in Section 6.2, we show how to implement sorting networks on an FPGA (Section 6.3). We evaluate several implementations of different sorting networks in Section 6.4. While this allows an in-depth analysis of FPGA-specific implementation aspects it does not provide any insight of how the FPGA behaves in a complete system. We make up for that in Sections 6.6 and 6.7 where we illustrate two complete use cases. In Section 6.6 we illustrate the implementation of a median operator using FPGA hardware. The second use case (Section 6.7) consists of a sorting co-processor that is directly connected to the execution pipeline of the embedded PowerPC CPU.

6.1 Related Work

Sorting is an important problem in computer science and has been extensively studied. We reference relevant work on modern hardware below.

GPUTeraSort [GGKM06] parallelizes a sorting problem over multiple hardware shading units on the GPU. Within each unit, it achieves parallelization by using

SIMD operations on the GPU processors. The AA-Sort [IMKN07], CellSort [GBY07], and MergeSort [CNL$^+$08] algorithms are very similar in nature, but target the SIMD instruction sets of the PowerPC 970MP, Cell, and Intel Core 2 Quad processors, respectively.

FPGAs are being successfully applied in signal processing, and we draw on some of that work in Sections 6.6. The particular operator that we use in Section 6.6 is a median over a sliding window. The implementation of a median with FPGAs has already been studied by Wendt et al. [WCG86], but only on smaller values than the 32 bit integers considered in this paper. Our median implementation is similar to the sorting network proposed by Oflazer [Ofl83].

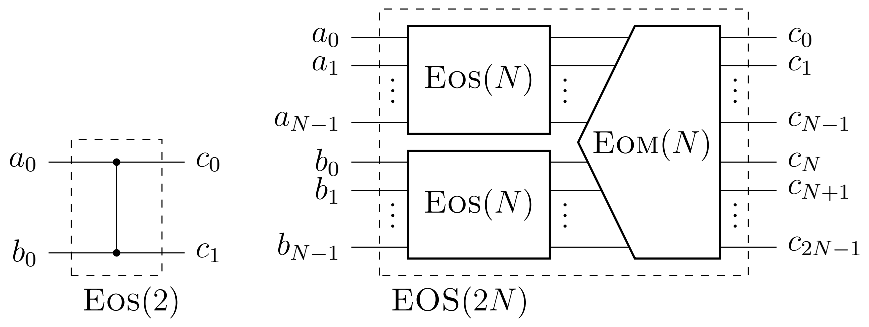
Claessen et al. [CSS03] present a FPGA-based sorter as a use case for their Lava language [BCSS99]. Lava is a hardware description language embedded in Haskell that provides programming level abstractions for circuit composition including layout. Regular structures such as sorting network can be expressed in Lava and then automatically placed on the chip.

6.2 Sorting Networks

Some of the most efficient traditional approaches to sorting are also the best options in the context of FPGAs. *Sorting networks* are attractive in both scenarios, because they *(i)* do not require *control flow* instructions such as branches and *(ii)* are straightforward to *parallelize* (because of their simple data flow pattern). Sorting networks are suitable for relatively short sequences whose length is known a priori. Sorting networks have been extensively studied in literature. For a detailed treatment see [Knu98, Bat68, CLRS01]. On modern CPUs, sorting networks suggest the use of vector primitives, which has been demonstrated in [GBY07, GGKM06, IMKN07].

The circuits of the sorting networks are composed of horizontal wire segments and vertical *comparators*. We represent the comparator elements using the widely known Knuth notation $\updownarrow$. The unsorted elements are applied at the left, one element per wire (Figure 6.6). The sorted output then appears on the right side. Wire segments connect the different compare-and-swap stages. Each wire segment can transfer an m-bit number. The comparator elements perform a two-element sort, such that the smaller of the two input values leaves the element on the right side through the upper output wire and the larger value through the lower wire.

In the following, we describe two systematic methods to build sorting networks. The first method is based on *Even-odd Merging networks*, the second on *Bitonic Merging networks*. Both were proposed by K. E. Batcher [Bat68].



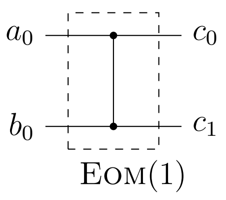
(a) Basis (b) Recursion

Figure 6.1: Recursive definition of even-odd sorter Eos($2N$)

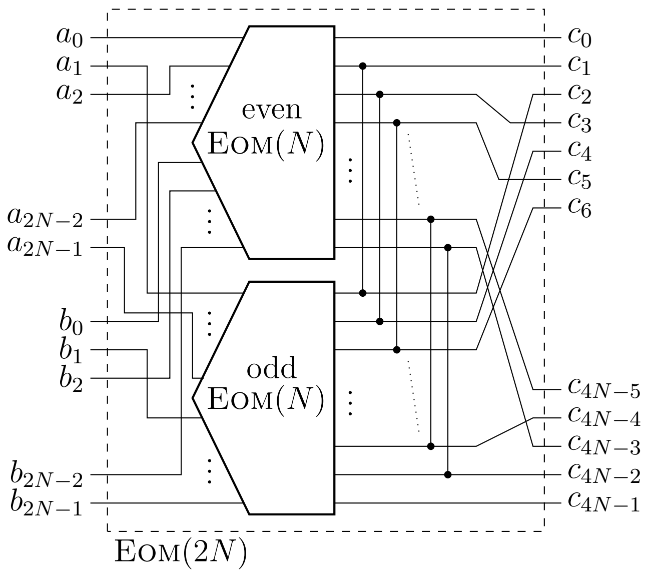
6.2.1 Even-odd Merging Networks

Even-odd merging networks are built following a recursive definition that is assumed to be efficient when number of elements $N = 2^p$ is a power of two [Knu98]. We use the exponent p to describe the size of a network. At the heart of the networks are even-odd merging elements that combine two sorted subsequences $a_0 \leq \cdots \leq a_{N-1}$ and $b_0 \leq \cdots \leq b_{N-1}$ into a single sorted sequence $c_0 \leq \cdots \leq c_{N-1} \leq c_N \leq \cdots \leq c_{2N-1}$. Using these merging elements a sorting network can be built recursively as shown in Figure 6.1. The input sequence of size $2N$ is split into two sequences of size N. Each of these sequences is sorted by an even-odd sorter Eos(N). The sorted outputs are then merged using an even-odd merger Eom(N) of size N. The recursive definition of Eos(N) is depicted in Figure 6.1(b). Figure 6.1(a) shows the basis of the recursion where a single comparator is used to sort the two elements.

The even-odd merging elements Eom(N) that combine the two sorted sequences of length N are defined in a similar recursive form. The basis of the recursion Eom(1) is a single comparator as shown in Figure 6.2(a). The recursion step illustrated in Figure 6.2(b) works as follows: Given are two sorted input sequences $a_0, a_1, \ldots, a_{2N-1}$ and $b_0, b_1, \ldots, b_{2N-1}$ for an even-odd merger Eom($2N$). The N even-indexed elements $a_0, a_2, \ldots, a_{2k}, \ldots, a_{2N-2}$ are mapped to the a-inputs of the "even" merger. The N odd-indexed elements $a_1, a_3, \ldots, a_{2k+1}, \ldots, a_{2N-1}$ are mapped to the a-inputs of the "odd" merger. The b inputs are routed similarly. As it can be easily shown the inputs of the even and odd mergers are sorted, hence, each produces a sorted sequence at the output. The two sequences are then combined by an array of $2N - 1$ comparators as shown in Figure 6.2(b). By unrolling the recursion of Eos(N) and Eom(N) a sorting network consisting of comparators is created. An example of an even-odd merging network that is able to sort eight inputs is shown in Figure 6.6(a).



(a) Basis



(b) Recursion

Figure 6.2: Recursive definition of Even-odd merger EOM(N) for $2 \times N$ elements

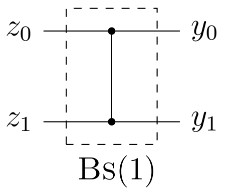
6.2.2 Bitonic Merging Networks

In CellSort [GBY07] and GPUTeraSort [GGKM06] sorting is based on *Bitonic Merging networks*. A bitonic sequence can be regarded as a partially sorted list that consists of two sorted monotonic subsequences, one ascending the other descending. For example, $1, 4, 6, 5, 2$ is a bitonic sequence whereas $1, 4, 6, 5, 7$ is not.

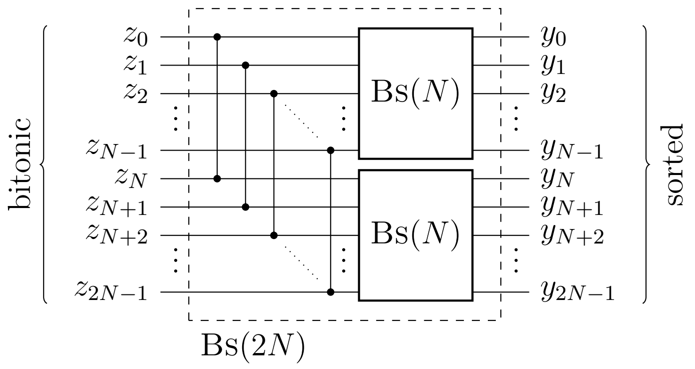
A bitonic sequence can be transformed into a sorted monotonic sequence using a *Bitonic Sorter* [Bat68]. Figure 6.3 shows the recursive definition of a bitonic sorter BS($2N$) that transforms the $2N$ bitonic input sequence $z_0, z_1, \ldots, z_{2N-1}$ into a sorted output sequence.

The reason for introducing bitonic sequences in the first place is that they can be easily generated from two sorted sequences $a_0, \ldots, a_{N-1}$ and $b_0, \ldots, b_{N-1}$. It can be shown that concatenating $a_0, \ldots, a_{N-1}$ and the sequence $b_{N-1}, \ldots, b_0$, i.e., the a sequence with the reversed b sequence, yields a bitonic sequence. This bitonic sequence can then be sorted by a bitonic sorter BS($2N$). This process generates a network that merges two sorted input sequences of length N. The resulting *Bitonic Merging* network is shown in Figure 6.4(a). Using the fact that reversing a bitonic circuit is also bitonic the circuit can be redrawn without wire-crossings in Figure 6.4(b).

Following the divide-and-conquer approach bitonic merger in Figure 6.4 can be recursively applied producing a complete sorting network. Such a network is

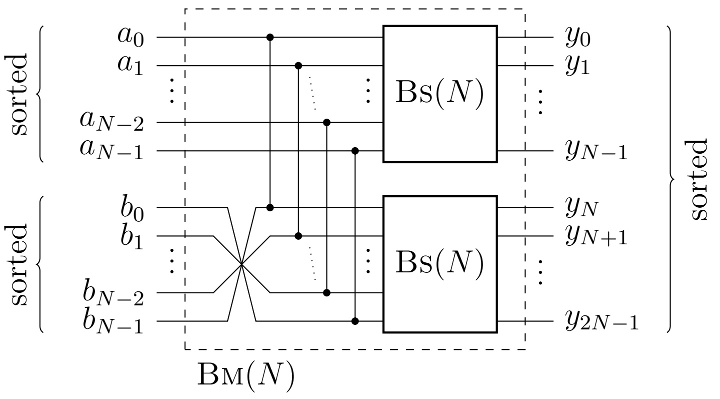


(a) Basis

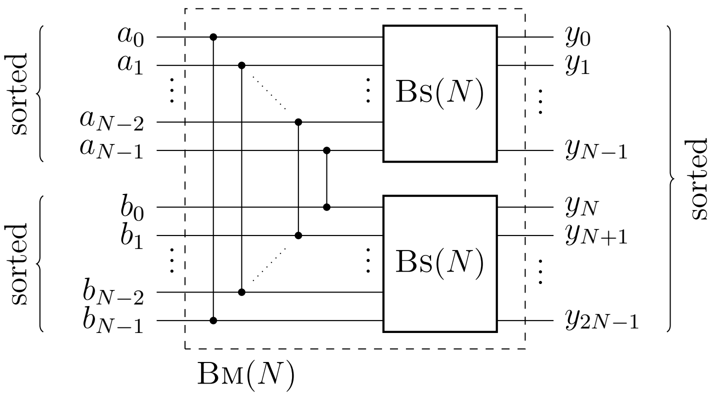


(b) Recursion

Figure 6.3: Bitonic Sorters produce a sorted output from a bitonic input sequence

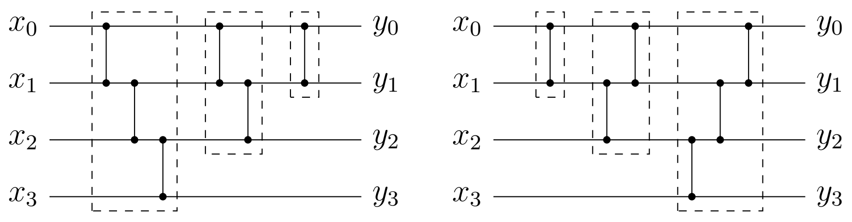


(a) Merge network using two Bitonic Sorters and flipping the b sequence.



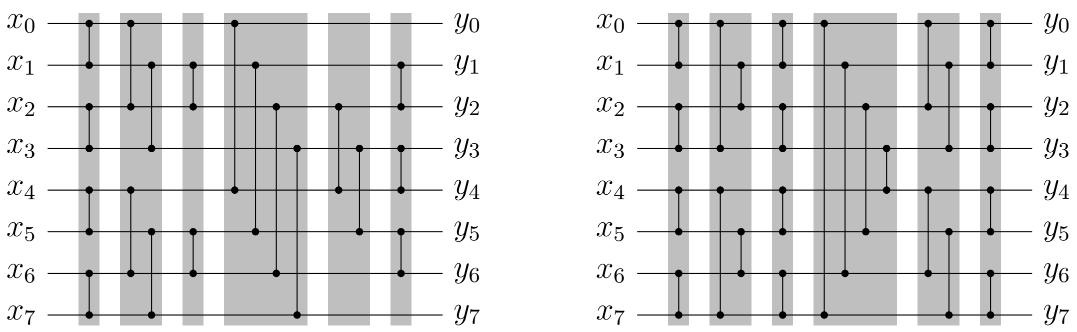
(b) Equivalent circuit without wire-crossings.

Figure 6.4: Sorting network as recursive definition of a merging network based on Bitonic Sorters



(a) Bubble Sort (b) Insertion Sort

Figure 6.5: Sorting networks based on Insertion Sort and Bubble Sort are equivalent



(a) Even-odd Merging Network (b) Bitonic Merging Network

Figure 6.6: Sorting networks for 8 elements

shown in Figure 6.6(b) for size $N = 8$.

6.2.3 Bubble and Insertion Sort Networks

Sorting networks can also be generated from traditional sorting algorithms. Figure 6.5 shows two networks that are generated from Bubble and Insertion Sort. When comparing the two circuits diagrams 6.5(a) and 6.5(b) it can be seen that the resulting networks are structurally equivalent. Like their algorithmic counterparts these sorting networks are inefficient. Networks generated by this approach require many comparator elements.

6.2.4 Sorting Networks Comparison

In general, the efficiency of a sorting network can be measured in the number of comparators required and the number of stages, i.e., steps, through the sorting network. Table 6.1 shows the resulting number of comparators $C(N)$ of a sorting network of size N and $S(N)$ the number of stages.

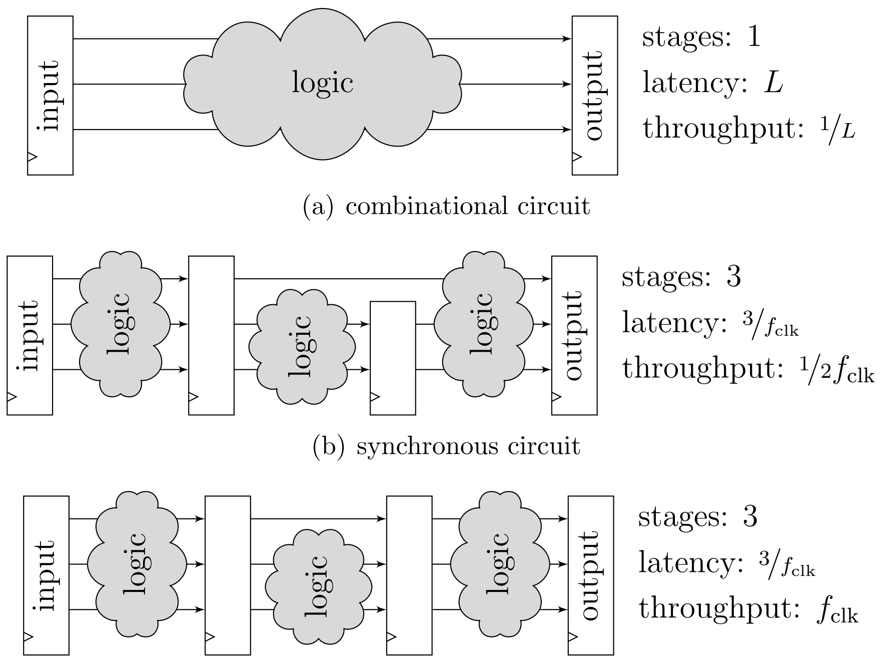
	bubble/insertion	even-odd merge	bitonic merge
exact	$C(N) = \frac{N(N-1)}{2}$	$C(2^p) = (p^2 - p + 4)2^{p-2} - 1$	$C(2^p) = (p^2 + p)2^{p-2}$
	$S(N) = 2N - 3$	$S(2^p) = \frac{p(p+1)}{2}$	$S(2^p) = \frac{p(p+1)}{2}$
asymp-	$C(N) = O(N^2)$	$C(N) = O\left(N \log^2(N)\right)$	$C(N) = O\left(N \log^2(N)\right)$
totic	$S(N) = O(N)$	$S(N) = O\left(\log^2(N)\right)$	$S(N) = O\left(\log^2(N)\right)$
$N = 8$	$C(8) = 28$	$C(8) = 19$	$C(8) = 24$
	$S(8) = 13$	$S(8) = 6$	$S(8) = 6$

Table 6.1: Comparator count $C(N)$ and depth $S(N)$ of different sorting networks of size N. For sizes $N = 2^p$ the exponent p is given.

Bitonic Merge and Even-odd Merge sorters have the same depth and the same asymptotic complexity. However, asymptotic behavior is of little interest here, as we are dealing with relatively small array sizes. An interesting result was published in [AKS83] that proposes a sorting network with a better asymptotic complexity $C(N) = O\big(N \ln(N)\big)$ and depth $S(N) = O\big(\ln(N)\big)$. However, the constant dropped in the O-notation is too big and thus renders it unsuitable for practical sorting networks [Knu98].

Despite requiring more comparators bitonic merge sorters are frequently used because they have two important properties: (1) all signal paths have the same length and (2) the number of concurrent compares for each stage is constant. For example, in the Bitonic Merging network in Figure 6.6(b) every wire undergoes six compares. In contrast, consider the uppermost wire of the even-odd merging sorter in Figure 6.6(a). The path $x_0 \rightsquigarrow y_0$ passes only through three comparator stages, whereas $x_2 \rightsquigarrow y_2$ passes through all 6 stages. This has the disadvantage that different signal lengths must be considered, for example, if the data is clocked through the network, i.e., one stage every clock, additional registers may be necessary of buffering intermediate values.

In a bitonic merging network, $N/2$ compares are present in each stage. For even-odd mergers the number is not constant. In Figure 6.6(b) for example, stage 1 has 4 concurrent compares, whereas stage 3 has only 2. A constant number of comparisons is useful to efficiently implement the sorting network in a sequential form using a fixed number of comparators M given by the architecture, e.g., the SIMD vector length. A single stage can be executed in $N/(2M)$ steps. If a stage is not using all operators, some comparators remain idle. This fact is exploited in [GBY07, GGKM06].



(a) combinational circuit

(b) synchronous circuit

(c) fully pipelined circuit

Figure 6.7: Implementation approaches for digital circuits

6.3 Implementing Sorting Networks

The sorting networks shown in Section 6.2 can be implemented in several ways using different technologies. In this context we study two implementations in FPGA hardware and a CPU-based implementation. For the hardware variant we differentiate between three types of circuits: *combinational*, *synchronous*, and *pipelined* implementations. Before diving into the implementation details we first discuss the key properties of these circuit types.

6.3.1 Combinational vs. Synchronous Pipelined Circuits

Purely combinational circuits are operated without a clock signal. They consist of *combinatorial logic* only. Figure 6.7(a) illustrates the combinatorial logic as a cloud between a pair of registers. Input signals are applied from a register on the left. The signals travel through the combinatorial logic that comprises the sorting network. The comparators are implemented primarily using FPGA lookup tables (combinatorial logic). The signals travel through the sorting stages without following a synchronization signal such as a clock. At some predetermined time instants the signals at the output are read and stored in another register. The key characteristics of combinational circuits is the absence of registers.

In implementations of sorting networks that are using combinational logic only one single N-set can reside in the network at any given time. [1] The next N-set can only be applied after the output signals for the previous set are stored away in a register following the sorting network. Hence, both the latency and the issue interval are determined by the length of the combinatorial signal path between the input and the output of the sorting network. More precisely, they are given by the longest delay path L from any input to any output. The throughput of the design that uses combinational only is $1/L$ N-sets per second. The path delay is difficult to estimate in practice, as it involves not only the delay caused by the logic gates themselves but also the signal delay in the routing fabric. The maximum delay path L directly depends on the number stages $S(N)$ but also on the number of comparators $C(N)$ as they contribute to the routing pressure and further increase latency. In Section 6.4, we measure the propagation delay for different network sizes. For an even-odd merging network with $N = 8$ and $m = 32$ we measure $L = 18$ ns for the FX130T–2 chip. The throughput of the combinational circuit is $1/L$ N-sets per second. For the example circuit this translates into 56 M 8-sets/sec, which corresponds to a processing rate of 1.66 GB/s.

By introducing registers in the combinatorial circuit the length of the signal path can be broken up. The computation of the circuit is thereby divided into stages that are separated by registers (see Figure 6.7(b)). The resulting circuit is called *synchronous* since a common clock is used to move the data through the network from stage to stage at specific instants. Clearly, both types perform the same computation, hence, the combinatorial logic is conceptually identical. The crucial difference are the registers. They require additional space but have an interesting advantage.

A natural way is to place the register after each stage in the sorting network. Since the path delays between registers are smaller than the delay in the entire network consisting of purely combinational logic it can be clocked faster. The highest possible clock frequency is now determined by a shorter maximal path. The overall latency of the synchronous circuit is $S(N)f_{\mathrm{clk}}$ where f_{clk} is the frequency of the clock that drives the registers. The registers can be inserted arbitrarily in the combinatorial signal paths, not necessarily at the end of each sorting stage, allowing to trade-off the latency $S(N)f_{\mathrm{clk}}$ with the operational clock speed f_{clk}. In VLSI design this technique is known as *register balancing*. In this work, we assume

[1] A technique known as *wave pipelining* [BCKL98] can be used to send several items through a combinatorial network. The signals travel through the network as multiple waves. In wave pipelining the temporal spacing between the signals is less than the longest path in the circuit. Although wave pipelining can significantly reduce latency, the circuit design is very complex. Signal paths must be carefully routed while considering the uncertainty in the path delay (e.g., temperature and data-dependent variations). Traditional FPGA design tool do not provide support for wave pipelining designs.

that registers are added after each comparator.

Note that by introducing stage registers alone the circuit does not necessarily become fully pipelined. In a fully pipelined circuit a new input set can be applied every clock cycle resulting in a throughput of f_{clk} N-sets per second. However, just adding a register at the output of a comparator does not necessarily make it fully pipelined. Additional registers are required to buffer the value on wires that are not processed between stages. This is illustrated in Figure 6.7. The first wire in 6.7(b) is not buffered by the second register stage. It seems unnecessary as this signal is not involved in the combinatorial logic of the second stage. While this saves a flip-flop, now special care needs to be taken for timing the signals. In order to have the signals line up correctly at the output registers, the inputs have to be applied during two consecutive cycles. When buffering every signal as shown in 6.7(c) the circuit becomes fully pipelined, i.e., the all signal path reaching the output register have the same length and a new input can be applied at every clock cycle. This is particularly relevant for even-odd sorting networks as we will see later. The network shown in Figure 6.9 can be clocked at $f_{\mathrm{clk}} = 267\,\mathrm{MHz}$ on our chip. Being fully-pipelined, this directly translates into a data processing rate of 7.9 GB/s.

6.3.2 Implementing Comparators on FPGAs

The sorting network circuits shown in Section 6.2 can be directly translated into digital circuits. The essential component is the implementation of the comparator $\updownarrow$ in FPGA logic. The sorting network can then be built by instantiating the required comparators and wiring them up accordingly.

In the following, we look at how the comparator can be defined in a high-level hardware description language VHDL. Then we study how the FPGA tool chain translate this description and maps it to the FPGA primitives shown in Figures 5.1 and 5.2. This allows us to analyze the resource utilization on the chip.

Purely Combinational Comparators. The complexity of the comparator is given by the width of its inputs. For this analysis we consider fixed-length m-bit integer values. The results we provide in the thesis are based on $m = 32$ bit integers. In general, however, any m and any comparison function can be used, e.g., double precision floats etc. In hardware, the choice of the data type only affects the implementation of the comparator not the sorting network itself. This is different from sorting network realizations on GPUs and CPUs where different types are provided in different configurations, e.g., a compare-and-swap instruction is only provided for integers but not for floating-point values.

We specify the behavior of the comparator element in the VHDL hardware description language as follows (where the first `<=` indicates a signal assignment

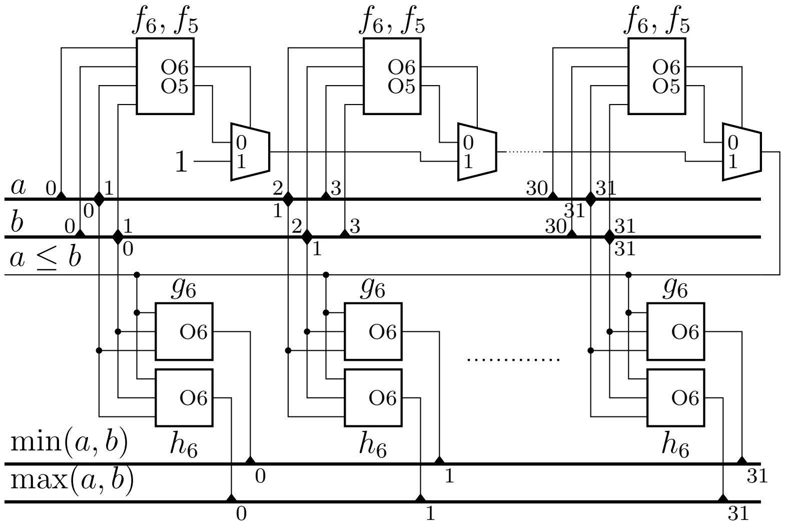


Figure 6.8: FPGA implementation of an *purely combinational* 32-bit comparator requiring 80 LUTs (16 for evaluating $a \leq b$ and 2×32 to select the minimum/maximum values).

and the second a less-or-equal operator):

```
entity comparator is
  port (
    a   : in  std_logic_vector(31 downto 0);
    b   : in  std_logic_vector(31 downto 0);
    min : out std_logic_vector(31 downto 0);
    max : out std_logic_vector(31 downto 0));
end comparator;
architecture behavioral of comparator is
  min <= a when a <= b else b;
  max <= b when a <= b else a;
end behavioral;
```

The two conditional signal assignments are *concurrent assignments*, i.e., they describe the functional relationship between the inputs and the outputs and can be thought as being executed "in parallel". The component `comparator` is instantiated once for each comparator element in the sorting network. The vendor-specific FPGA synthesis tools will then compile the VHDL code, map it to device-primitives, place the primitives on the 2D grid of the FPGA and finally compute an efficient routing of the signals between the sites on the chip.

Figure 6.8 shows the circuit for our Virtex-5 FPGA generated by the Xilinx ISE 11.3 tool chain. The 32 bits of the two inputs a and b are compared first (upper half of the circuit), yielding a Boolean output signal for the outcome of the predicate $a \leq b$. The signal drives 2×32 LUTs configured as multiplexers that connect the proper input lines to the output lines for $\min(a, b)$ and $\max(a, b)$ (lower half of the circuit).

For the comparisons $a \leq b$, the LUTs are configured to compare two bits of a and b each. As shown earlier in Figure 5.2 the LUTs on the Virtex-5 chip can have up to two outputs and can be connected to up to 5 common inputs. The two outputs are then connected through the fast carry-multiplexers . This results in a carry chain where the multiplexer selects the lower input if O6 is high. Otherwise, the carry-multiplexer selects the output O5 of the LUT. Therefore, the two Boolean functions f_5 and f_6 implemented by the LUT are

$$\begin{aligned}
f_6(a_i, a_{i+1}, b_i, b_{i+1}) &= \left(\bar{a}_i\bar{b}_i \vee a_i b_i\right)\left(\bar{a}_{i+1}\bar{b}_{i+1} \vee a_{i+1}b_{i+1}\right) \\
f_5(a_i, a_{i+1}, b_i, b_{i+1}) &= \bar{a}_i b_{i+1} b_i \vee \bar{a}_{i+1}\bar{a}_i b_i \vee \bar{a}_{i+1} b_{i+1} \ .
\end{aligned}$$

Here, a_i and b_i refers to the i-th of the two integers a and b bit in little-endian order. f_6 compares the two bit positions (a_{i+1}, a_i) and (b_{i+1}, b_i). If they are not equal f_5 evaluates the predicate $(a_{i+1}2^{i+1} + a_i 2^i) < (b_{i+1}2^{i+1} + b_i 2^i)$.

The lower array of LUT pairs implement a multiplexers that select the right bits for the min- and the max-outputs of the comparator element using the predicate $a \leq b$. Let c the Boolean value of the comparison. Then, the LUT g_6 for the minimum-output is

$$g_6(a_i, b_i, c) = a_i c \vee b_i \bar{c}$$

and for the maximum-output

$$h_6(a_i, b_i, c) = b_i c \vee a_i \bar{c} \ .$$

Resource Usage. From Figure 6.8 it can be seen that a comparator that performs a compare-and-swap operation of two m-bit numbers can be implemented using $\lceil 5m/2 \rceil$ LUTs and $\lceil m/2 \rceil$ carry-multiplexers. Usually, chip utilization is measured in the number of occupied slices. The number of slices used for a design consisting of a given number of LUTs depends on the packaging strategy followed by the placer of the FPGA tool chain. In a optimal packaging with maximum density where all four LUTs in a slice are used (see Figure 5.2) in total $\lceil 5m/8 \rceil$ FPGA slices are used for each comparator. Thus, for $m = 32$ at least 20 slices are required for each comparator. This results in an upper bound of 1,024 comparators that be placed on our Virtex-5 FX130T chip. Note that in practice, however, the placer does not use this maximum packing strategy. In general, not every slice is fully occupied, i.e., all its LUTs are in use. Sometimes is is more efficient to co-locate a LUT with the input output block (IOBs) to the chip pins in order to reduce routing distances and, hence, latency. The slice usage can be even higher as the tool may

	$a > b$	min/max	total
32-bit integer	16 LUTs	64 LUTs	80 LUTs
single precision float	44 LUTs	64 LUTs	108 LUTs
double precision float	82 LUTs	128 LUTs	210 LUTs

Table 6.2: Number of LUTs required for different comparator types. The numbers are subdivided into the logic evaluating the predicate and the logic that selects the min/max values based on the predicate value.

be forced to use LUTs as plain "route-through" elements when it runs short on direct connection wires in the interconnect fabric.

Latency of a Single Comparator. The FPGA implementation in Figure 6.8 is particularly time efficient. All lookup tables are wired in a way such that all table lookups happen in parallel. Outputs are combined using the fast carry logic implemented in silicon for this purpose. Ignoring routing delays for the moment the latency of the circuit, i.e., the time until output signals "min" and "max" of the comparator are valid after applying the inputs, is given by sum of two LUTs (one of the comparison chain and one multiplexer LUT) and the propagation delay of the chain of $\lceil 5m/2 \rceil$ carry-multiplexer. From the Virtex-5 data sheet [Xil09a] the logic delay (excluding routing in the network) is 0.89 ns for $m = 32$.

Comparators for Floating Point Numbers. When sorting floating point numbers the compare-and-swap elements of the sorting network have to be replaced. The logic used to evaluate the predicate $a \leq b$ for two floating point numbers is significantly different from two integer values. Xilinx provides an IP core that implements the $a \leq b$ comparison for floating-point numbers. It supports the basic IEEE-754 single- and double-precision format, however, without denormalized numbers (treated as zeros). Table 6.2 shows the number of lookup tables used for a single compare-and-swap element for different data types. The numbers are split into the logic used to evaluated to the predicate $a \leq b$ and the multiplexer logic to select the min/max value. Since single-precision is also 32 bits wide the multiplexer logic has the same complexity as the integers used in this paper. The single-precision comparison $a \leq b$ requires 108 Virtex-5 LUTs in total compared to the $\lceil m/2 \rceil = 16$ LUTs for integers. For double-precision 210 LUTs are needed.

6.3.3 Combinational Sorting Networks

The sorting network is implemented by instantiating comparators and wiring them accordingly. As pointed out earlier, there are no explicit stages and register that buffer intermediate results. Instead, the comparator circuits (LUTs and carry-multiplexer) form a large network of *combinatorial logic.*

We can provide a lower-bound for the chip area required for the entire sorting network. As pointed out in the previous section, comparator elements require $^{5m}/_{2}$ LUTs each, where m is the data width in bits. The total number of lookup tables thus is

$$\#\text{LUTs} = \frac{5}{2} C(N) m \ .$$

Using the $C(N)$ from Table 6.1 we compute the number of lookup tables for even-odd merging and bitonic merging networks:

$$\begin{aligned} \#\text{LUTs}_{\text{even-odd}} &= 5m(p^2 - p + 4)2^{p-3} - \frac{5m}{2} \\ \#\text{LUTs}_{\text{bitonic}} &= 5m(p^2 + p)2^{p-3} \ . \end{aligned}$$

The total area consumption measured in number of occupied slices depends on the packaging. We can provide a lower bound based on the following simplifying assumption that multiplexer LUTs (see Figure 6.8) are not placed in the same slice as the logic used to evaluate the $a \leq b$ predicates (no combination of "unrelated" logic). The area utilization can be estimated as

$$\begin{aligned} \#\text{slices}_{\text{even-odd}} &= 5m \left[(p^2 - p + 4)2^{p-5} - {}^1/_8\right] \\ \#\text{slices}_{\text{bitonic}} &= 5m(p^2 + p)2^{p-5} \ . \end{aligned}$$

Ignoring additional LUTs used as "route throughs" the chip area of the 8-element even-odd merging network (for $m = 32$) shown in Figure 6.6(a) containing 19 comparators requires 380 slices, i.e., 1.86 % of the Virtex-5 FX130T chip. The corresponding network based on bitonic mergers (Figure 6.6(b)) requires 24 comparators resulting in 480 FPGA slices, or equivalently, 2.34 % of the chip.

6.3.4 Synchronous Implementation on FPGA

In a synchronous design an external clock signal moves the data from stage to stage through the sorting network. To this extent, the comparator outputs are connected to banks of flip-flips, called *stage registers.* The register store the input during the rising edge of the clock signal. The output of a stage register is then fed to the next comparator stage as shown in Figure 6.9.

Latency. The latency is determined by the clock frequency f and the depth $S(N)$ of the sorting network. In fact, the time between applying the data at the input and reading the sorted data at the output is given by $^{S(N)}/_{f_{\text{clk}}}$. For example, the even-odd merging network shown in Figure 6.9 on our Virtex FX130T–2 FPGA can be operated at $f_{\text{clk}} = 267\,\text{MHz}$. Hence, the overall latency for the 6-stage network is $^6/_{267\ \text{MHz}} = 22.5$ ns.

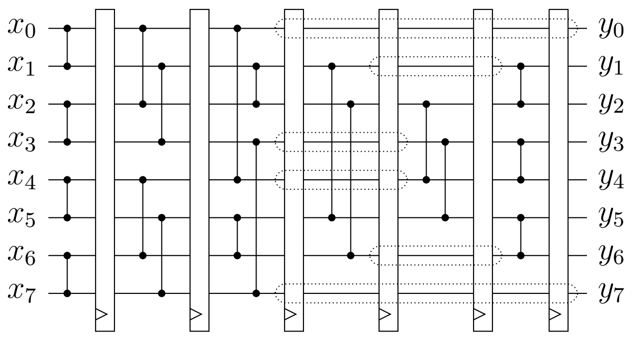


Figure 6.9: Pipelined synchronous even-odd merge sorting networks using six $8 \times 32 = 256$ bit pipeline registers. Dotted rectangles indicate register stages that can be combined into a *shift register*.

Pipelining. Synchronous sorting networks can further be implemented in a fully-pipelined way. This allows keeping an N-item set "in-flight" at every stage of the sorting network. Because the outputs of the comparator are buffered in a register after every stage, a complete new N-set can be inserted at the input every cycle.

As it can be seen in Figure 6.9, in even-odd merging networks not all wires are processed by a comparator in every cycle. For example, in the third stage, the wires x_0, x_3, x_4, and x_7 are not processed by a comparator. In order to obtain a pipeline these wires still need to buffered by a register as shown in the Figure 6.9. This increases the number of occupied slices.

Resource Usage. The synchronous implementation differs from the purely combinational network by the stage registers. In a non-fully pipelined implementation the registers can be easily accommodated in the comparator logic. The outputs of the lookup tables g and h of the combinational comparator implementation (see Figure 6.8) can simply be connected to the corresponding flip-flops (see Figure 5.2 for the LUT–flip-flop configuration in an FPGA slice). Hence, no additional slices are used for comparators and the total resource consumption of a sorting network identical to the combination implementation, i.e., for a sorting network consisting of $C(N)$ comparators on N m-bit inputs $C(N)\lceil 5m/8 \rceil$ FPGA slices. Again, this is a lower-bound that is only reached if all slices are fully-occupied, i.e., all 4 LUTs/flip-flops of a slice are used, and no additional lookup tables are used as "route-throughs".

For the fully-pipelined implementation we can provide a similar lower-bound for the chip area required for fully-pipelined implementation. Now, a complete Nm-bit register is needed for each stage. Hence, the total number of LUTs and flip-flops (FFs) required is

$$\begin{aligned}\#\text{LUTs} &= \frac{5}{2}C(N)m \\ \#\text{FFs} &= S(N)Nm \ .\end{aligned}$$

It can be easily verified that the resource usage in even-odd merging and bitonic merging networks is given by

$$\begin{aligned}\#\text{LUTs}_{\text{even-odd}} &= 5m(p^2 - p + 4)2^{p-3} - \frac{5m}{2} \\ \#\text{LUTs}_{\text{bitonic}} &= 5m(p^2 + p)2^{p-3} \ .\end{aligned}$$

The number of stages $S(N)$ is the same for both network types, therefore, also the number of registers:

$$\#\text{FFs} = 4m(p^2 + p)2^{p-3} \ .$$

For the lower bound on the slice count we are using the assumption that the register following a comparator is always placed in the same slice, i.e., the output of the multiplexer-LUT is directly routed to the flip-flop register that is co-located with that LUT. Furthermore, we assume that flip-flops of stage registers without a comparator (e.g., shown inside dotted rectangles in Figure 6.9) are not placed in the same slice as the logic used to evaluate the $a \leq b$ predicates (no combination of "unrelated" logic). The area utilization then is:

$$\begin{aligned}\#\text{slices}_{\text{even-odd}} &= m\left[(5p^2 + 3p + 4)2^{p-5} - 1/8\right] \\ \#\text{slices}_{\text{bitonic}} &= 5m(p^2 + p)2^{p-5} \ .\end{aligned}$$

Note that under these placement assumptions the slice usage for bitonic merging networks is identical to the implementation that uses only combinational logic. This is due to the fact that in bitonic networks there is a comparator for each wire in every stage, hence, all flip-flops registers can be co-located with a lookup table belonging to a comparator such that no additional slices are required.

6.3.5 Sorting Networks on CPUs

Sorting networks for CPUs have been extensively studied in literature, in particular for exploiting data parallelism on modern SIMD processors [IMKN07, CNL+08, FAN07, GBY07]. In this section, we show how sorting networks can be directly implemented on general-purpose CPUs. We show the implementations for two different hardware architectures: Intel x86-64 and PowerPC. We use these implementations later to compare the FPGA design against.

Neither of the two architectures provides built-in comparator functionality in its instruction set. We therefore emulate the functionality using conditional moves (x86-64) or the carry flag (PowerPC). The following two sequences of assembly code implement the comparator operation for PowerPC and x86-64 processors:

$$[r8, r9] \leftarrow [\min(r8, r9), \max(r8, r9)] \ .$$

PowerPC Assembly	x86-64 Assembly
`subfc r10,r8,r9`	`movl  %r8d,%r10d`
`subfe r9,r9,r9`	`cmpl  %r9d,%r8d`
`andc  r11,r10,r9`	`cmova %r9d,%r8d`
`and   r10,r10,r9`	`cmova %r10d,%r9d`
`add   r9,r8,r11`	
`add   r8,r8,r5`	

Neither piece of code makes use of branching instructions. The same property has important consequences also in code for traditional CPUs. Branch instructions incur a significant cost due to flushing of instruction pipelines (note that sorting algorithms based on branching have an inherently high branch misprediction rate). This is why the use of a sorting network is a good choice also for CPU-based implementations.

Related Implementations using SIMD. Chhugani et al. [CNL+08] describe an SIMD implementation for sorting single precision floating point numbers using Intel SSE instructions. Similar work is done by Inoue [IMKN07] with the PowerPC AltiVec instruction set for both integer and single precision floating point data. In both cases, data parallelism in SIMD is used to sort multiple elements in a SIMD vector register in one step. For example, in the approach described by Chhugani et al. four 32-bit single precision floating point numbers are compared and swapped. Below, we briefly outline how they compare two vectors $\mathtt{A} = (a_3, a_2, a_1, a_0)^T$ and $\mathtt{B} = (b_3, b_2, b_1, b_0)^T$.

Assume that we want to compare a_0 to a_1, a_2 to a_3, b_0 to b_1 and b_2 to b_3. This pattern occurs, for example, in the first stage of 8-element bitonic merging network shown in Figure 6.6(b). The Intel SSE architecture provides two instructions that determine the *element-wise* minimum and maximum of two vectors. In order to perform the desired comparisons the elements in the two vectors have to be shuffled into the correct position, which is done by additional shuffle instructions. The required shuffle operations are illustrated in Figure 6.10. The operations can be directly implemented in C using SSE-intrinsics as follows:

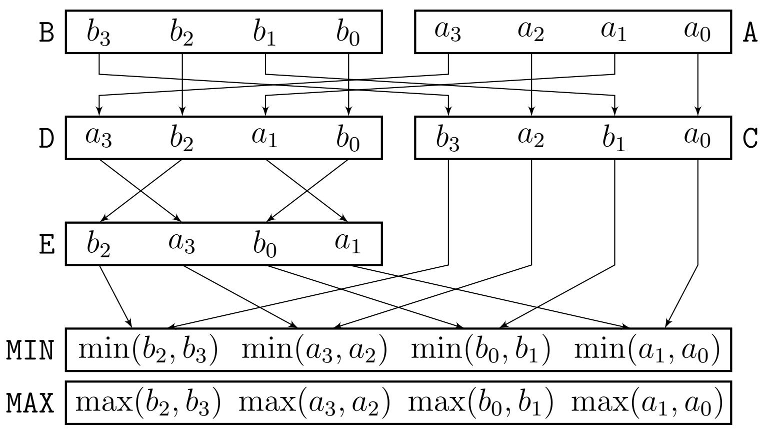

B
b_3 b_2 b_1 b_0
a_3 a_2 a_1 a_0
A
D
a_3 b_2 a_1 b_0
b_3 a_2 b_1 a_0
C
E
b_2 a_3 b_0 a_1
MIN
$\min(b_2, b_3)$ $\min(a_3, a_2)$ $\min(b_0, b_1)$ $\min(a_1, a_0)$
MAX
$\max(b_2, b_3)$ $\max(a_3, a_2)$ $\max(b_0, b_1)$ $\max(a_1, a_0)$

Figure 6.10: Vector compare implemented using SSE instructions

```
__m128 A, B, C, D, E, MIN, MAX;
C = _mm_blend_ps(A,B,0xA);
D = _mm_blend_ps(B,A,0xA);
E = (__m128)_mm_shuffle_epi32((__m128)D,0xB1);
MIN = _mm_min_ps(C,E);
MAX = _mm_min_ps(C,E);
```

Chhugani et al. [CNL$^+$08] split sorting into different stages to account for the fixed-length SIMD registers. First, an in-register sort phase sorts 16 elements in 4 SIMD registers. Then a 2×4 bitonic merging network BM(4) is used to merge two resulting sorted lists. The fixed vector length of SSE makes Bitonic networks a good choice as they have a constant number of comparators $N/2$ in each stage. Even-odd sorters would require additional buffering, which adds to the cost for shuffling elements in vectors. This shuffling overhead increases for larger networks and limit along with the fixed number of SIMD registers (16 on x86-64) available the scalability of this approach. In FPGAs implementations this additional shuffling translates into an increased signal routing complexity, which also limits scalability.

6.4 Evaluation: Sorting Circuits on FPGAs

In this section, we provide a detailed evaluation of the sorting network implementations on our Virtex-5 FPGA (FX130T). Before turning to the application use cases in Sections 6.6 and 6.7 we analyze both the purely combinational and synchronous implementations of the *even-odd merging* and *bitonic merging* networks without the side-effects caused by the attachment of FPGA (e.g., bus and memory performance).

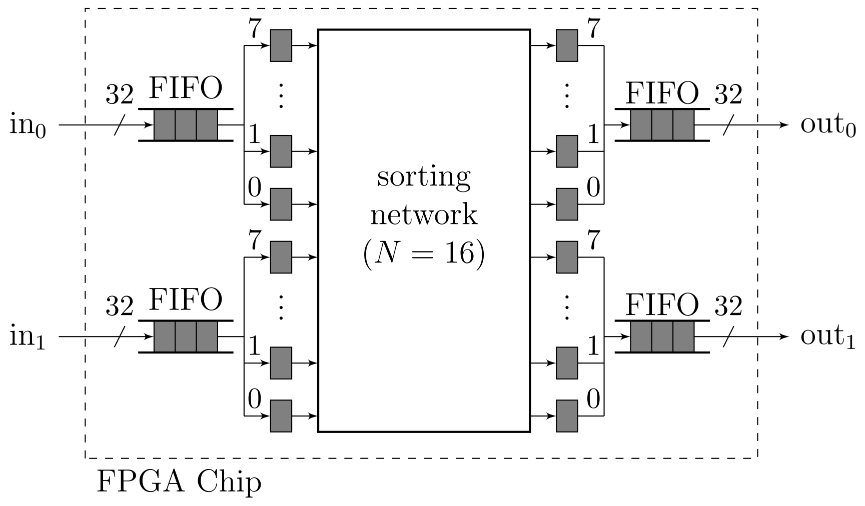


Figure 6.11: Sort chip architecture used to evaluate implementations of sorting networks. Example shown for $N = 16$.

To this extent, we implement the sorting network as a dedicated sort chip. Data to be sorted is applied at I/O pins of the FPGA. Similarly, the sorted output can be read from an other set of I/O pins. The sort chip approach can be regarded as being artificial because an integrated circuit in custom silicon only implementing sorting in practice is of limited use. Nevertheless, it provides an environment to evaluate the sorting networks. When designing the chip we consider two important aspects. First, the implementation must be fully functional, that is, no simplifications are allowed that might lead the FPGA tool chain to shortcut parts of the design. For example, all input and outputs of the sorting networks must be connected to an I/O pin of the chip, otherwise, sub-circuits driving unconnected signals might be pruned in the optimization stage of the synthesis.

Second, for evaluating the raw performance of the sorting circuit, routing pressure when connecting to I/O blocks must not dominate the overall speed. Although it is in principle possible to connect all inputs and outputs for an $N = 8$ element sorting network to the FPGA pins, it leads to longer routing distances because a large chip area needs to be covered since the I/O are uniformly distributed over the chip. For larger networks $N > 8$ more than the 840 I/O pins available on the Virtex-5 FX130T FPGA are required to interface the sorting network.

Hence, in order to minimize the impact of routing I/O signals we significantly reduced the width of the chip interface and use an architecture as shown in Figure 6.11. The key are FIFO buffers (BRAM blocks) placed at the input and output of the sorting network. The FIFO buffers that have different widths at the read and write interfaces. Xilinx provides FIFO IP cores that can have a width ratio between inputs and outputs of up to 1:8 or 8:1. For example, for $N = 8$ we use an input-side FIFO with an write with of 32 bit. This allows us to write one 32-bit word per clock cycle. Using a width ratio 1:8 we can read 8 consecutive elements from this FIFO into the sorting network. Similarly for the output-side FIFO in a 8:1 configuration, we can write all 8 32-bit outputs of the sorting network into the FIFO. The output FIFO is the connected to the output pins of the chip through a 32-bit wide interface, such that we can read the sorted output one element per clock cycle. For network sizes $N > 8$ we use multiple FIFO lanes with ratio 1:8 and 8:1. Figure 6.11 shows two FIFO lanes for $N = 16$.

An additional advantage of using FIFOs is that they can be clocked at a different rate than the sorting network. This isolates the timing analysis of the sorting network from that of the chip interface. Although it is impossible to clock the FIFOs eight times higher than the sorting network, we nevertheless can try to maximize the clock of the sorting network in order to determine the raw speed of the sorting network alone.

We evaluate resource consumption the network types and the synchronous and purely combinational implementations as follows. The resources used by the implementations (number of lookup tables, flip-flop registers and slices) are shown for the sorting network alone excluding logic for handling clock and the FIFOs. We estimate the resource consumption of the sorting network by using the number of the complete circuit. To this extent we replace the sorting network by a "pass-through" and compare it with the full implementation including the sorting network. Since the input/output logic in both case is the same, the difference is due to the actual sorting network. In the following, we only report the difference numbers.

6.4.1 Synchronous Implementations

Figure 6.12 shows the number of flip-flops (registers) used in the synchronous, fully-pipelined implementation of the even-odd and bitonic sorting network. The dotted line shows the prediction of the cost model introduced in Section 6.3.4. The model predicts the same value for both network types. It can be seen in Figure 6.12 that the model accurately predicts the flip-flop number for the bitonic sorting network. However, for even-odd sorting networks the model overestimates the register usage. This can be explained by the specific structure of even-odd networks that is not considered by the simple model. In even-odd networks not every wire has a comparator in every stage. This has an important consequence in a fully-pipelined implementation shown in Figure 6.9. Several stages without comparators represent shift registers (shown as dotted rectangles in Figure 6.9). Instead of using flip-flop registers the shift registers can be implemented more efficiently on Xilinx FPGAs using LUTs configured as such called *shift register*

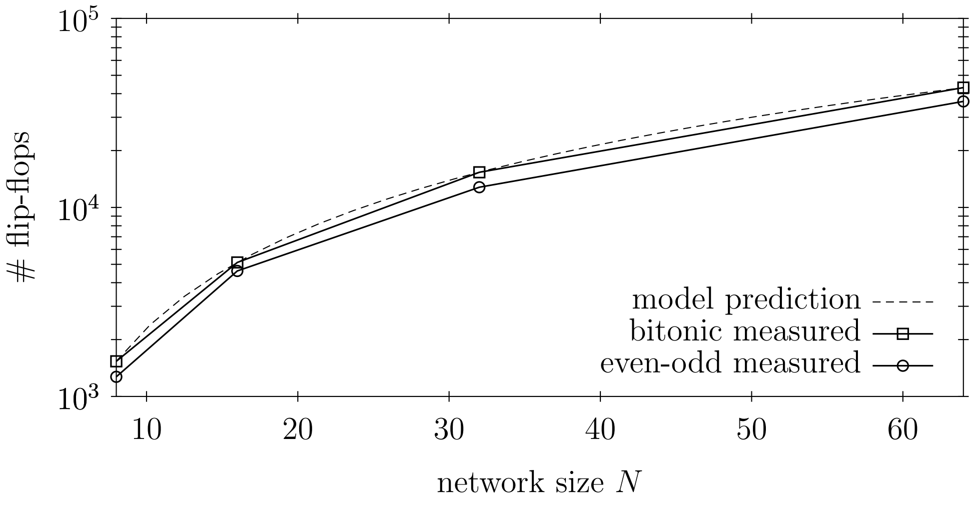


Figure 6.12: Flip-flop usage of synchronous fully-pipelined implementations. While the model accurately predicts the resource requirements for bitonic merging networks it overestimates the flip-flop usage for even-odd merging networks.

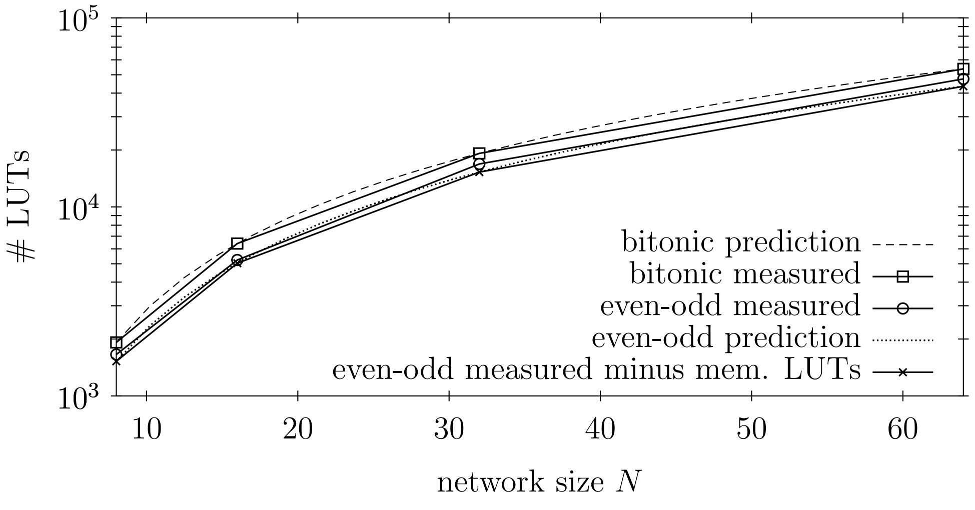


Figure 6.13: LUT usage of synchronous fully-pipelined implementations. The model accurately predicts the resource consumption of bitonic merging networks. When LUTs used as shift register lookup table in even-odd merging networks are subtracted the model predictions are also correct for even-odd networks.

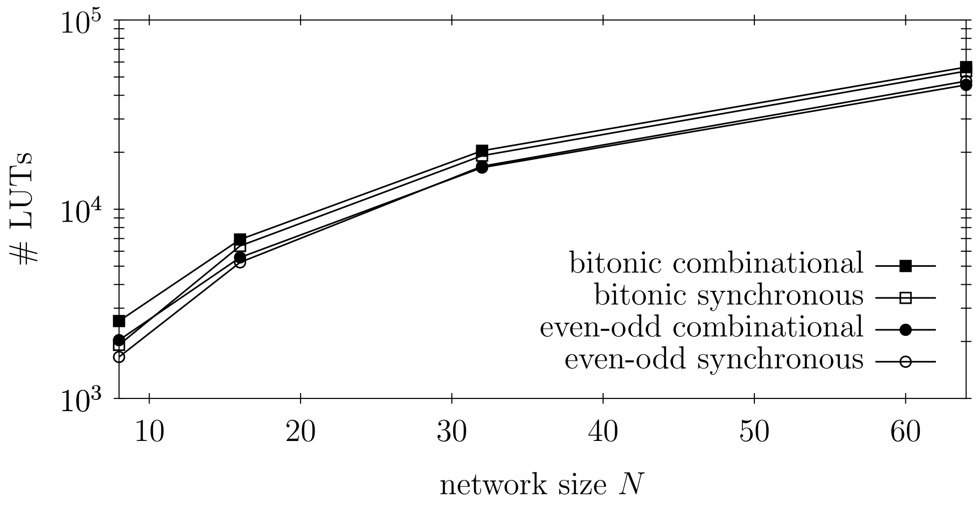


Figure 6.14: LUT usage in purely combinational and synchronous fully pipelined implementations. Purely combinational circuits result in a higher LUT usage.

lookup tables (SRL). Hence, the actual number of flip-flop registers is reduced for even-odd sorting networks.

Replacing flip-flops by LUTs increases number of LUTs, such that the LUT resource model will underestimate the LUT usage for even-odd networks. This can be seen in Figure 6.13 that shows the LUT utilization. The figure also shows the predicted values for both network types (from Section 6.3.4). Whereas the model prediction is correct for bitonic networks it underestimates the LUT usage for even-odd networks. However, when subtracting the number of LUTs configured as SRL from the total number the model for even-odd networks is accurate too.

When comparing the synchronous implementation of the two network architectures we observe that even-odd networks require less chip space, both, in number of flip-flop registers and lookup tables.

6.4.2 Implementation based on Combinational Logic

Combinational implementations do not contain any flip-flops in the sorting network. By analyzing the routed FPGA design we could verify that FPGA design tools furthermore did not introduce any flip-flops, for example, registers for pipelining or latency balancing. The lookup table utilization is shown in Figure 6.14. The figure also shows the effective number of LUTs used in the synchronous designs for comparison. It can be seen that for bitonic merging networks the combinational circuit requires more LUTs than the synchronous design. It turns out that this also holds for even-odd networks once the additional LUTs used in synchronous

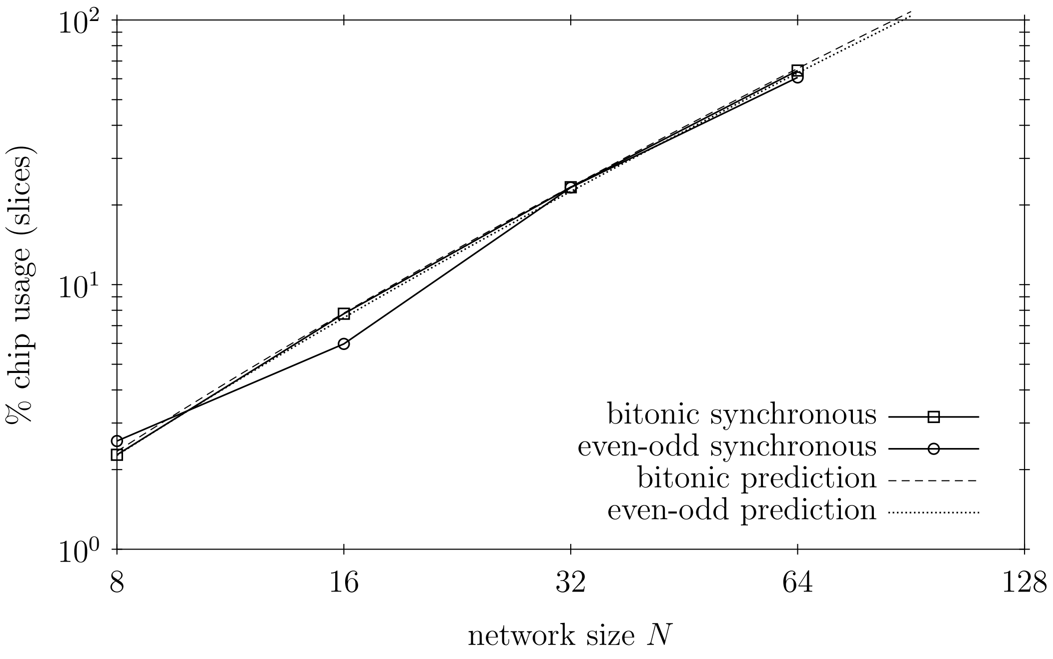


Figure 6.15: Chip usage (measured in slices) of fully-pipelined synchronous implementations

implementations for shift registers are subtracted.

It is not quite clear why the combinational circuits versions require more lookup tables. We believe it is an artifact introduced by the Xilinx design tools. An analysis of the routed design showed that LUTs were not fully used, e.g., not all four inputs in the comparators for evaluating the predicates $a > b$. The difference to synchronous circuits is that in the combinational case each output bit of the sorting network can be expressed as a huge Boolean function of all inputs bits, e.g., for $N = 8$ sized network, there are 256 Boolean functions with 256 Boolean inputs each. During synthesis, the tools try to minimize these functions and later on map the resulting expressions back to FPGA LUTs. We believe that this process is based on heuristics and has limitations in performance.

6.4.3 Chip Usage

Figure 6.15 and 6.16 show the overall chip utilization in % of FPGA slices for the synchronous and purely combinational implementations respectively. Both plots are down in double-logarithmic scale. The resulting straight line corresponds to a power function. We are able to place designs up to $N = 64$ elements onto the Virtex-5 FX130T FPGA chip. When synthesizing networks for $N = 128$ the tools

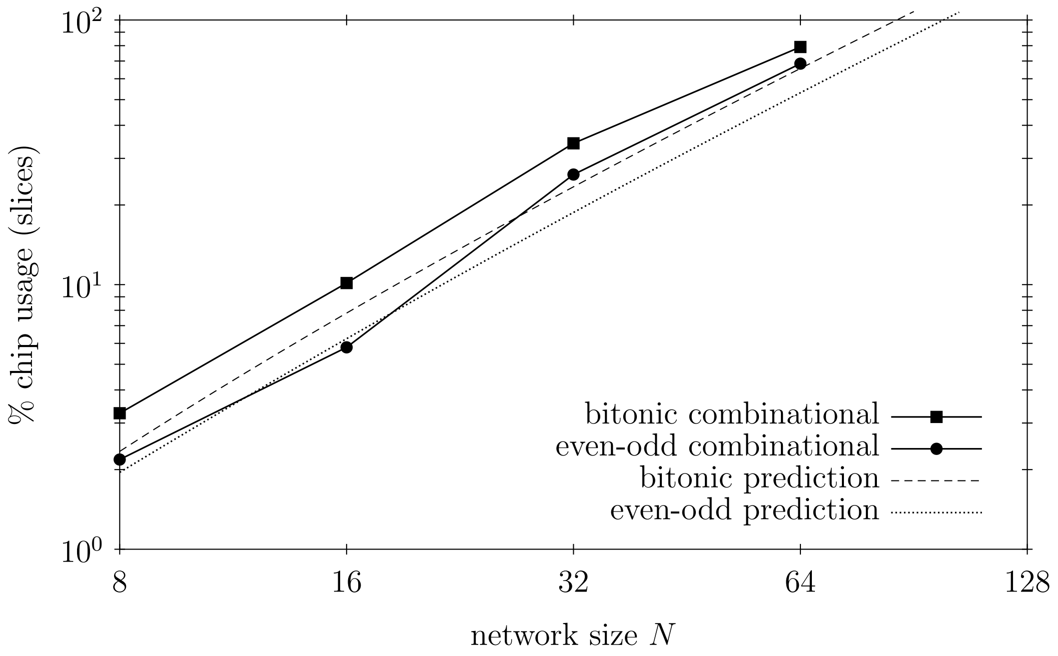


Figure 6.16: Chip usage (measured in slices) using purely combinational logic

will abort due to overmapping over both registers and lookup tables.

In general, the slice utilization highly depends on the timing constraints, i.e., the clock frequency for synchronous networks and the maximum signal delay through the combinational network. The chip utilization values we report here are obtained at the highest performance constraints that can be met by the FPGA tools. For the fully-pipelined implementations in Figure 6.15 we can observe that the slice usage roughly corresponds to the model prediction. The outlier for even-odd networks at $N = 16$ seems to related again to heuristics in the tools as it only occurs at tight timing constraints.

The chip utilization for purely combinational circuits (Figure 6.16) significantly deviates from the model predictions. In general, the model underestimates the utilization, in particular for larger network sizes. An analysis of the synthesized design showed that many slices are not fully occupied, i.e., not all LUTs or flip-flops are used. We observe this behavior when the place-and-route stage optimizes for speed instead of space. This is the case here as we chose tight timing constraints while there are still enough resources (area) available on the chips such that the tools are not forced to combine unrelated logic into the same slices.

In conclusion, we can observe that the resource models introduced earlier work best for flip-flop and LUT usage for synchronous circuits. They are less accurate for combinational circuits, in particular for slice usage.

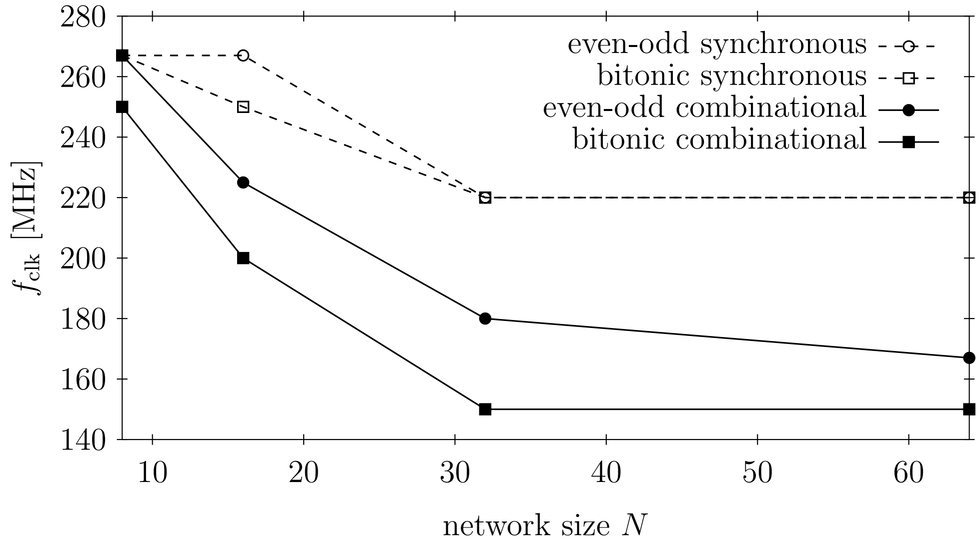


Figure 6.17: Maximum clock frequencies the sorting network implementations can be operated

6.4.4 Circuit Performance

For FPGAs, as for any hardware design, performance is given by the timing behavior. The timing behavior is specified during synthesis through one or more time constraints. The tools then try to find a design during the place-and-route phase that meets these constraints. If this phase completes successfully the system is able to operate that at this timing. The phase fails if one or more constraints are not met. In this case, the system does not operate correctly at this timing. The designer is then left to operate the circuit as close to the desired parameters the tool was able to synthesize a design, otherwise, the timing constraints have to be relaxed and the entire process repeated until a successful design is found.

For synchronous designs we only set the desired clock frequency of the sorting network. We gradually decrease the clock frequency until the place-and-route phase completes successfully. We plot the clock frequency in Figure 6.17. Combinational networks have two timing constraints. The first is the core clock, which is needed to drive the scheduler that reads the input data from the FIFOs, applies the data to sorting network, waits until the output of the sorting work is valid, and then stores the output in the out-FIFO. The second parameter is the latency in the combinational sorting network that corresponds to the longest delay path in the sorting network. We round this path delay down to the next closest number of clock cycles. Clearly, the two timing parameters are correlated. We perform a search in this 2-dimensional space as follows. First, we maximize the clock fre-

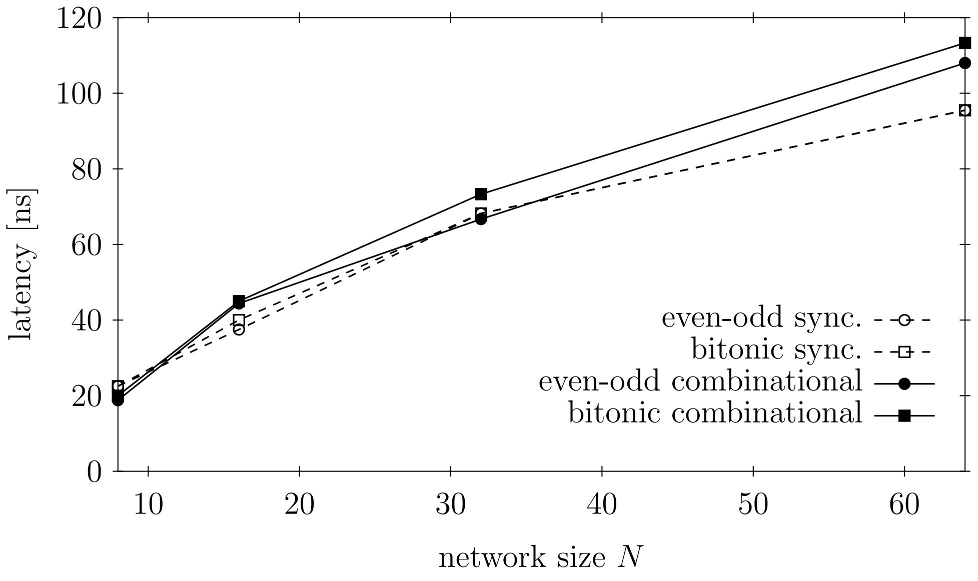


Figure 6.18: Data latency in the different sorting network implementations

quency f_{clk} and set delay constraint for the sorting network to $\lfloor S(N)/f_{\text{clk}} \rfloor$. $S(N)$ is the number of swap stages in sorting network. Once we found the maximum f_{clk} we gradually reduce the path delay constraint until no valid design can be found.

As it can be seen in Figure 6.17 the clock frequency decreases as the network size increases. This behavior corresponds to algorithms in traditional computing where execution times increase (or throughput decreases) with the problem size. If the clock frequency would not increase the throughput would increase as N grows. It can also be seen from the figure that synchronous circuits can be clocked significantly higher. There is no significant difference between the two network types. For combinational-only circuits the higher complexity of bitonic networks result in a lower clock speed.

Figure 6.18 shows the latency of the sorting network. It is measured as the time between applying the inputs at the sorting network and reading the sorted data the output of the sorting network. For the fully-pipelined implementations the latency is equal to $S(N)/f_{\text{clk}}$. For combinational implementations, we directly determine the latency L. It can be seen that for networks $N > 8$ synchronous circuits have lower latency, even though the additional register stages in the sorting network inherently uses the signal propagation through the network. The reason why combinational networks do have a higher latency is due to the lower overall clock speed that feeds and extracts data to and from the network. The large combinatorial circuits have a significant negative impact on the clock frequency as shown in Figure 6.17 such that latency gains by omitting the stage register cannot compensate the loss in the overall clock frequency.

Throughput is related to latency L. Fully-pipelined implementations can process an N-set every clock cycles while combinational implementations can process a tuple every $\lceil L f_{\text{clk}} \rceil$ cycle. Here, we can observe the significant gains of fully-pipelined designs. For example, both synchronous networks can process 64 elements at 220 MHz. This corresponds to a throughput of 14.08×10^9 elements/sec. Since the elements are 32-bit in size, the resulting throughput is 52.45 GiB/sec. In contrast, the fastest corresponding combinational network (even-odd as shown Figure 6.18) has a latency of 113.3 ns 150 MHz, which results in a throughput of 8.8×10^6 elements/sec or equivalently 2.1 GiB/sec. The high throughput numbers are very promising for the FPGA technology. However, so far we only analyzed the isolated performance the sorting network. The overall system performance depends on the integration, that it the attachment of the FPGA to the rest of the system. We analyze this performance through two different use cases in this chapter.

6.5 Manual Placement

So far the circuit were synthesized from a behavioral specification of the comparator element (see VHDL code in Section 6.3.2 and Figure 6.8) and a structural description of the wiring in the sorting network. The synthesized circuit is then mapped into FPGA elements and placed onto the chip by the FPGA design tools. The performance evaluation in the previous section showed that sorting networks built as combinatorial circuits have a lower performance than synchronous designs. This section discusses whether it is possible to increase the performance of combinatorial circuits through manual placement.

Sorting networks exhibit a regular structure, which allows generating circuits with explicit, in the sequel called "manual", placement of components on the FPGA chip. The section illustrates the following two important points: (1) manual circuit design and placement can result in circuits with better performance characteristics than those generated by the automatic design tools, (2) manual placement is a complex and nontrivial engineering task. In most cases the increased engineering effort outweighs the performance gains. Manual placement is typically used in practice when the design tools fail to meet timing requirements.

For manual placement, 2D layout tools such as Xilinx *PlanAhead* are used. The manual placement results in additional constraints that are considered in the later stages of the design flows (see Figure 5.3 on page 155). Alternatively, location constraints can be specified in VHDL code directly using VHDL attributes.

6.5.1 Relative Location Constraints

Components are placed relative to a coordinate system. The coordinate system is defined on the 2D FPGA array. Each coordinate addresses a single slice (see Figure 5.2, page 155). The Y-coordinate specifies the column, X-coordinate addresses the row in the array. FPGA components such as RAMs, LUTs, flip-flops, carry-chains, etc. can be addressed relative to this coordinate system. Since circuits are typically composed in a hierarchical structure the placement of the components and subcomponents is *relative* rather than *absolute*. The location of a VHDL component is set using an *RLOC* relative location constraint.

Example: The following VHDL code segment instantiates four Virtex-5 6-LUTs and places them all into the same slice at relative location X0Y0.

```
attribute rloc : string;
...
generate_luts : for i in 0 to 3 generate
   attribute rloc of comp : label is "X0Y0";
begin
   comp : LUT6_2
   generic map (
      -- implemented 6-to-2 bit logic function
      INIT => X"0000900900004F04")
   port map (
      O6  => eq(i),
      O5  => le(i),
      I0 => in_a(2*i),
      I1 => in_b(2*i),
      I2 => in_a(2*i+1),
      I3 => in_b(2*i+1),
      I4 => '0',
      I5 => '1');
end generate generate_luts;
```

Since all four LUTs are mapped to the same slice they are distributed among the A, B, C and D LUTs of the slice. □

6.5.2 Placement of Sorting Network Elements

The components of the sorting network are mapped onto chip slices as shown in Figure 6.19 for a eight element bitonic sorting network. The X/Y position is obtained from the recursive definition of the network in Figure 6.3 and Figure 6.4.

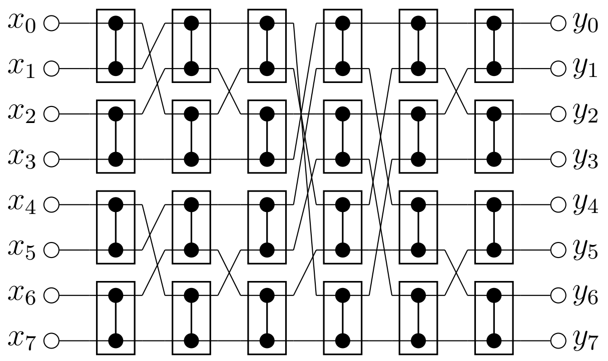
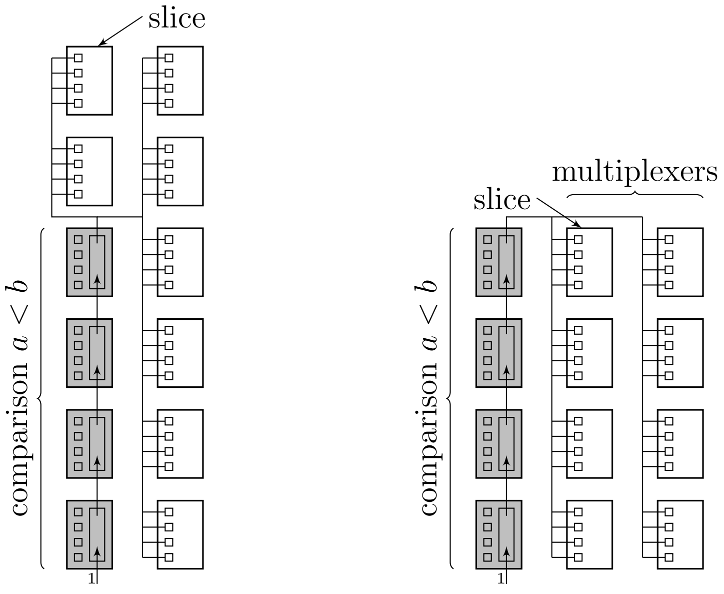


Figure 6.19: 8-element bitonic sorting network with explicit wiring between comparator elements

Even-odd sorters are placed in a similar manner. As shown earlier, networks based on even-odd sorters require less elements non-constant number of comparator elements in each column. This leads to “holes” in the placement.

For each comparator an RLOC constraint is specified. The spacing between the comparators has to be chosen such that there is no overlap between the units, i.e., the spacing is given by the dimension of the implementation of a comparator. The implementation of a 32-bit is was already depicted in Figure 6.8. A comparator element consists of a circuit for the predicate $a < b$ and a set of multiplexers. The analysis in Section 6.3.2 showed that 4 slices (16 LUTs and 4 carry elements) are used for the predicate and 16 slices (64 LUTs) for the output multiplexers (see Table 6.2). The output multiplexers can be implemented more efficiently for combinatorial sorters. Due to the missing flip-flops in the combinatorial sorters both outputs (O5 and O6) of the lookup tables can be used. This reduces the number of LUTs configured as output multiplexer by half. Only 8 slices (32 LUTs) are required for the multiplexers. In total, thus, 12 slices are needed for each comparator element. Two possible layouts of the resulting comparators are shown in Figure 6.20.

Figure 6.20(a) uses 2×6 slices whereas the layout in Figure 6.20(b) uses 3×4 slices. Given the routing structure of the FPGA interconnect the two different layouts result in different signal latencies. An observation obtained by placing sorting networks of different sizes and comparator layouts on the Virtex-5 FX130T FPGA showed that 2×6 layout in Figure 6.20(a) leads to 10–20 % shorter signal paths. In the following evaluation, therefore, the 2×6 layout is evaluated.



(a) 2×6 slices layout (b) 3 × 4 slices layout

Figure 6.20: Manual layouts of 32-bit combinatorial comparators for Virtex-5 FPGAs. Shaded slices contain comparison logic, unshaded slices contain output multiplexers.

6.5.3 Performance of Manual Placement

We evaluate the performance of even-odd and bitonic networks of different sizes on a Virtex-5 FX130T chip. Using a structural specification of the sorting network and its comparators the LUTs and carry units are explicitly placed on the chip. Figure 6.21 depicts the FPGA floor plans of two 32-element combinational bitonic sorting network designs. On the left, Figure 6.21(a) shows the design obtained through manual placement whereas Figure 6.21(b) shows the design obtained by applying the Xilinx synthesizer on a behavioral description of the comparator (as in Section 6.4.4) and automatic (unconstrained) placement. In both cases, the maximum delay path through the network is constrained at the lowest possible value PAR (place and route) can find a feasible solution. FPGA resources used for auxiliary logic, e.g., input and output FIFOs and tuple scheduling are shaded in gray, slices used for sorting network logic are colored black (Figure 6.21). Since the sorting network in both designs use approximately the same number of slices the size of black areas are roughly equal. However, the design on chip obtained by automatic placement ends up much more sparse.

Manual placement of components allows denser packing. Denser packing, however, does not necessarily lead to shorter signal paths. If the connectivity complex-

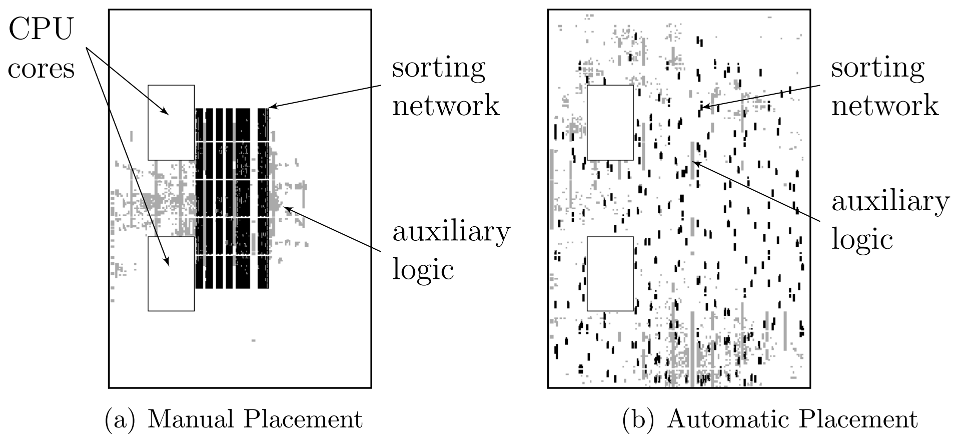


Figure 6.21: Floor plan of 32-element bitonic sorting networks on a Virtex-5 FX130T FPGA

ity exceeds the available routing capacity of the interconnect a larger chip area has to be used for routing. This in turn increases the length of the signal paths and thus defeats the purpose of a denser packing. Figure 6.22 compares the resulting latencies of the combinatorial sorting networks. The dashed lines show the latencies for manually placed bitonic and even-odd sorting networks. For comparison Figure 6.22 contains the latencies obtained by automatic placement and using a behavioral specification of the comparators (shown earlier in Figure 6.18). Due to a memory bug in Xilinx MAP tool in the Xilinx ISE 10.1, 11.4, 11.5, 12.1 and 12.2 tool suites, 64-element sorters could not be mapped. Hence, Figure 6.22 only shows latencies for 8, 16, and 32-element sorters. Compared to the automatically synthesized and placed circuits a 15–40 % latency improvement can be obtained for the manually placed circuits. Manual placement and mapping to FPGA resources allows to outperform the vendor design tools. We can conclude that the suboptimal performance behavior observed from automatically placed and mapped circuits in the previous section is solely a result of the current Xilinx tools and not due to the circuit structure or chip properties.

Explicit placement of sorting networks on FPGAs was also studied by Claessen et al. [CSS03]. They express the structure of a sorting network in their Lava hardware description language. The Lava code is an extension to Haskell and is compiled into a program that generates VHDL code and EDIF netlists for the corresponding sorting network. They implement various sorters for $N = 32$ elements with 16-bit word width on a Virtex-II FPGA. In contrast to work presented in this section, Claessen et al. used fully pipelined designs. In this dissertation, manual placement of purely combinatorial sorters is studied in order to verify that the poor performance behavior of the generated circuits is due to the current Xilinx design

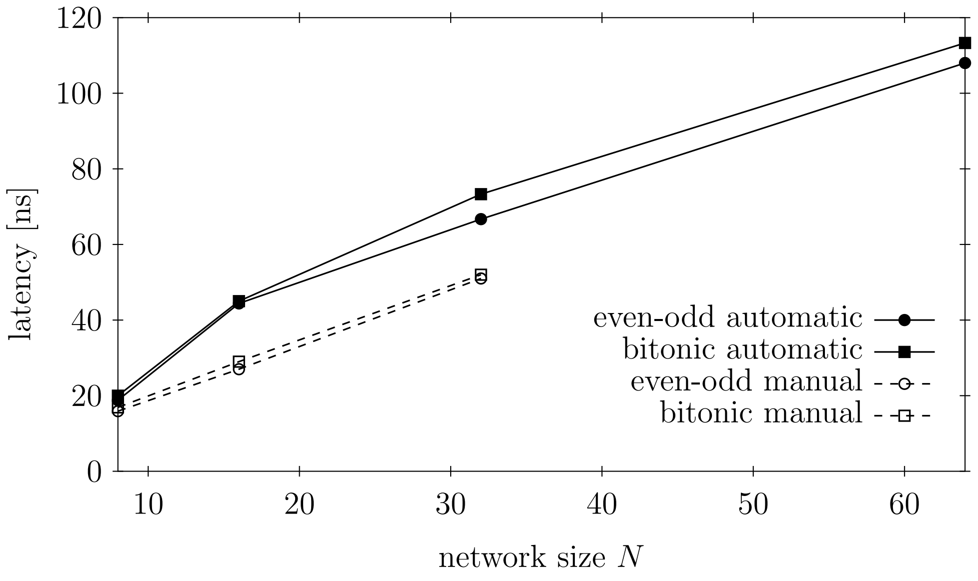


Figure 6.22: Data latency in the combinatorial sorting network implementations. Optimal manual placement leads to shorter delay paths in the network.

tools. The work by Claessen et al. does not compare the circuit performance of the manual placement with the placement obtained from the tools, however, a comparison is given for bitonic and even-odd networks. Claessen et al. can clock their even-odd sorter at 147 MHz and the bitonic sorting network at 127 MHz. This corresponds to a 15 % higher throughput for even-odd sorters. At the same time they report a 14 % latency improvement for even-odd sorters. In the results presented in this section for manual placement of combinatorial sorters no such difference can be observed. However, for automatic placement (Section 6.4.4) we can report a similar 9 % lower latency for even-odd sorters.

Manual placement is able to outperform vendor tools by 15–40 % for combinatorial sorting networks. Sorting networks exhibit a regular structure, which allows a systematic placement. However, manual placement as well as the technology mapping introduces an additional dependency as designs are chip-dependent and in contrast to high-level specifications cannot be easily ported to chips of different vendors or even to a different chip family within the same vendor. Furthermore, placement significantly increases the engineering effort. In general, designs generated by tools are good enough for most applications. However, the tools can be easily outperformed by hand-crafted designs [AFSS00]. For generating the query circuits presented in the next chapter we opted for automatic placement. First of all, the circuits do not exhibit a regular structure such as sorting networks. Second, in order to maximize throughput fully pipelined designs are used. As

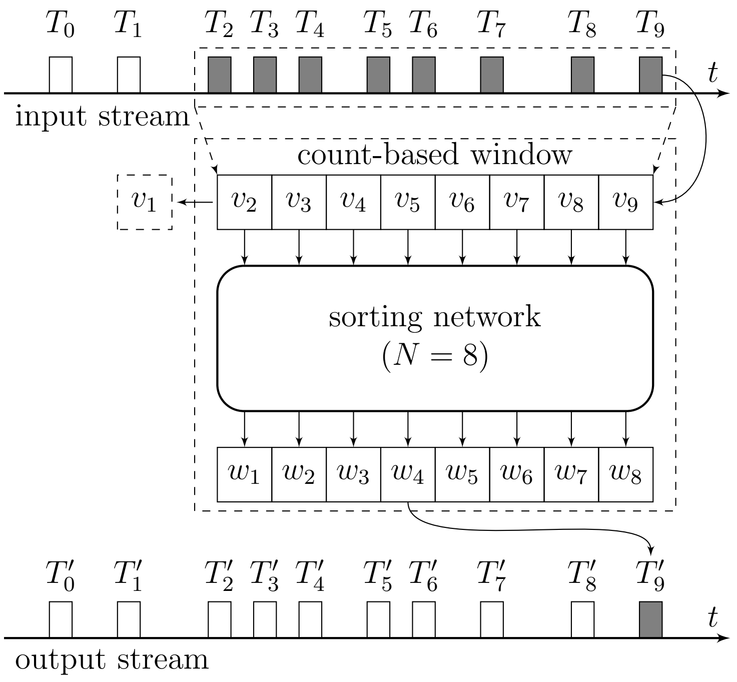

T_0 T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8 T_9
t
input stream
count-based window
v_1
v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9
sorting network
$(N = 8)$
w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8
T'_0 T'_1 T'_2 T'_3 T'_4 T'_5 T'_6 T'_7 T'_8 T'_9
t
output stream

Figure 6.23: Median aggregate over a count-based sliding window (window size 8)

shown in Figure 6.17 the tools are able to generate reasonable designs if they are synchronous and fully pipelined.

6.6 Use Case: A Streaming Median Operator

As a first use case for the sorting network circuits we choose a *median* operator over a count-based *sliding window* implemented on the aforementioned Xilinx board. This is an operator commonly used to, for instance, eliminate noise in sensor readings [RSS75] and in data analysis tasks [Tuk77]. For illustration purposes and to simplify the figures and the discussion, we assume a window size of 8 tuples. For an input stream S, the operator can then be described in CQL [ABW06] as

```
SELECT median(v)
  FROM S [ Rows 8 ] .
```
(Q_1)

The semantics of this query is illustrated in Figure 6.23. Attribute values v_i in input stream S are used to construct a new output tuple T'_i for every arriving input tuple T_i. A conventional (CPU-based) implementation would probably use a ring buffer to keep the last eight input values (we assume unsigned integer numbers), then, for each input tuple T_i,

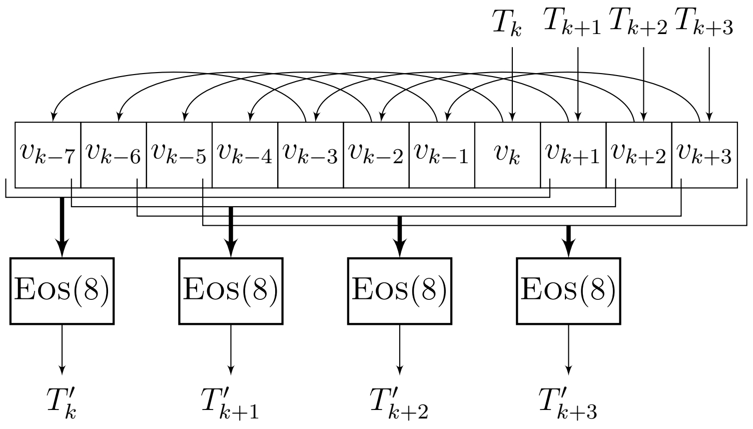


Figure 6.24: Implementation of median operator able to process four items per clock tick

(1) *sort* the window elements $v_{i-7}, \ldots, v_i$ to obtain an ordered list of values $w_1 \leq \cdots \leq w_8$ and

(2) determine the *mean value* from the ordered list. For the even-sized window we return w_4, corresponding to the *lower median.* Similarly, w_5 corresponds to the *upper median.*

The ideas presented here in the context of the median operator are immediately applicable to a wide range of other common operators. Operators such as selection, projection, and simple arithmetic operations (max, min, sum, etc.) can be implemented as a combination of logical gates and simple circuits similar to the ones presented here. We described one strategy to obtain such circuits in [MTA09b].

6.6.1 An FPGA Median Operator

We take advantage of the inherent hardware parallelism when implementing the operator. The goal is to maximize throughput by choosing a design that is able to process several tuples per clock cycle. The design of the median operator is illustrated in Figure 6.24. The operator accepts four consecutive tuples $T_k, T_{k+1}, T_{k+2}, T_{k+3}$ in every clock cycle. The tuple's values $v_k, \ldots, v_{k+3}$ are then inserted into the sliding window, which is implemented as a shift register. The shift register stores 11 32-bit elements. Since four new elements are inserted every cycle the elements have to move by four positions to the left. The 11-element shift register contains four overlapping sliding windows of length eight that are separated by one element. The elements of the four windows are then connected to four instances of

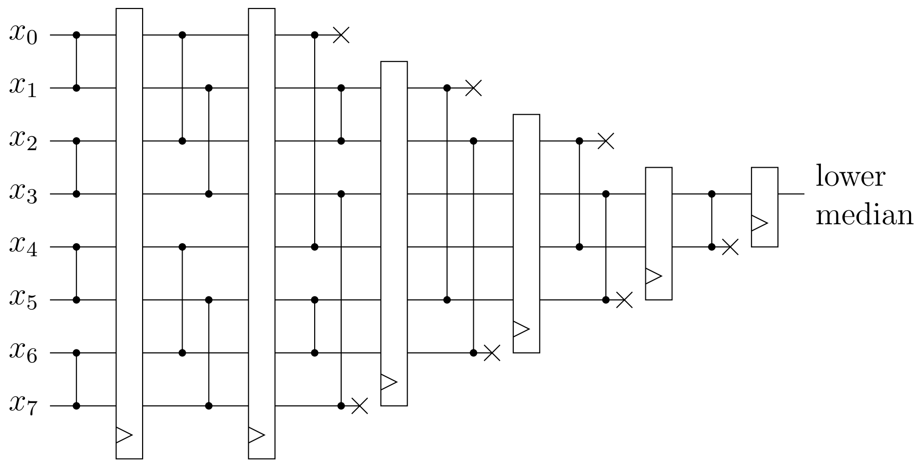


Figure 6.25: When computing the median the complexity of the fully-pipelined synchronous even-odd merge sorting network can be reduced from 19 comparators and 48 32-bit registers to 10 comparators, 7 half-comparators and 29 32-bit registers.

synchronous, fully-pipelined *even-odd merging sorting networks* Eos(8) (see Figure 6.9). The lower median for each window finally appears at the fourth output of the corresponding sorting network. In summary, the result tuples $T'_k, \ldots, T'_{k+3}$ are computed as follows from the windows:

$$\begin{aligned}
T'_k &\leftarrow \text{Eos}\big([v_k, v_{k-1}, \ldots, v_{v-7}]\big)_4 \\
T'_{k+1} &\leftarrow \text{Eos}\big([v_{k+1}, v_k, \ldots, v_{v-6}]\big)_4 \\
T'_{k+2} &\leftarrow \text{Eos}\big([v_{k+2}, v_{k+1}, \ldots, v_{v-5}]\big)_4 \\
T'_{k+3} &\leftarrow \text{Eos}\big([v_{k+3}, v_{k+2}, \ldots, v_{v-4}]\big)_4 \ .
\end{aligned}$$

Since we are only interested in the computation of a median, a fully sorted data sequence is more than required. Consider the even-odd sorting network shown in Figure 6.9. The lower median value appears at output y_3. Therefore, the upper and lower comparators of the last stage that sort y_1, y_2 and y_5, y_6 as well as the preceding register stages are not needed and can be omitted. The FPGA synthesis tool is able to detect unconnected signals and automatically prunes the corresponding part of the circuit. The pruned network is shown in Figure 6.25. Besides the reduction by 2 comparators and 19 32-bit registers the circuit complexity can further be reduced. Note that in Figure 6.25 7 comparators have one output unconnected. This means that the one the two 32-bit multiplexer that select the min/max value can be saved, which reduces the complexity by 32 LUTs.

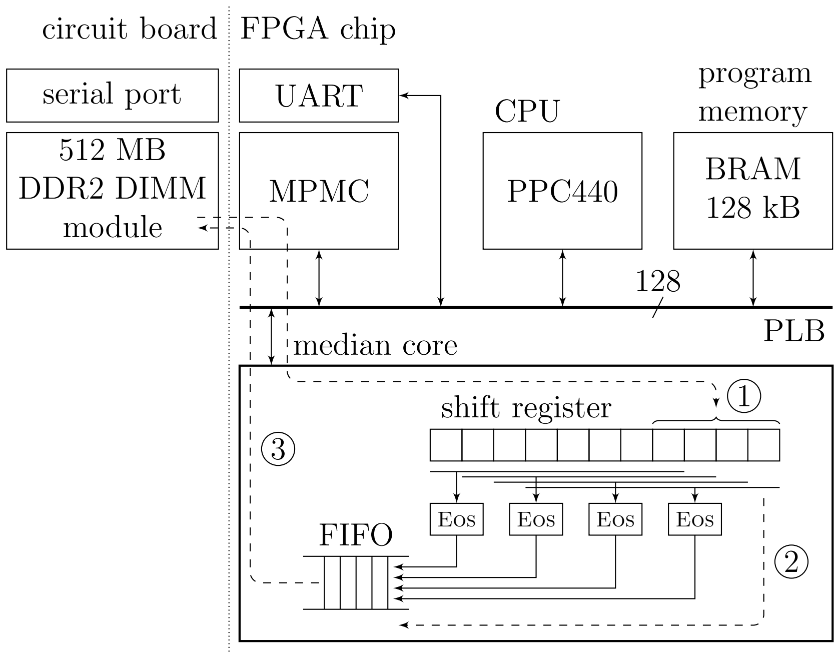


Figure 6.26: Architecture of the on-chip system: PowerPC core, 3 aggregation cores, BRAM for program, and interface to external DDR2 RAM

6.6.2 System Design

So far we have looked at our FPGA-based database operator as an isolated component. However, FPGAs are likely to be used to complement regular CPUs in variety of configurations. For instance, to offload certain processing stages of a query plan or filter an incoming stream before feeding it into the CPU for further processing.

In conventional databases, the linking of operators among themselves and to other parts of the system is a well understood problem. In FPGAs, these connections can have a critical impact on the effectiveness of FPGA co-processing. In addition, there are many more options to be considered in terms of the resources available at the FPGA such as using the built-in PowerPC CPUs and soft IP-cores implementing communication buses or controller components for various purposes. In this section, we illustrate the trade-offs in this part of the design and show how hardware connectivity of the elements differs from connectivity in software.

6.6.3 System Overview

Using the ML510 Virtex-5-based development board described in Section 5.2.3, we have implemented the embedded system shown in Figure 6.26. The system primarily consists of FPGA on-chip components. We use additional external (off-chip) memory to store larger data sets. The memory is provided by a 512 MB DDR2 DIMM module that is directly connected to the FPGA pins. The DIMM module operates at a bus clock of 200 MHz, which corresponds to DDR2-400 with a peak transfer rate of 3200 MB/sec.

On the FPGA we use one of the built-in PowerPC 440 cores, which we clock at the highest specified frequency of 400 MHz. The on-chip components are connected over a 128-bit wide *processor local bus* (PLB). We use 128 kB on-chip memory (block RAM) to store the code executed by the PowerPC (including code for our measurements). External memory used for the data sets is connected to the PLB through a *multi-port memory controller* (MPMC). It implements the DDR2 protocol. To control our experiments we interact with the system through a serial terminal. To this extent, we instantiate a soft IP-core for the serial UART connection link (RS-232).

Our streaming median operator participates in the system inside a dedicated processing core (Figure 6.26). As described in Section 6.6.1 the core contains the 11-element shift register and four sorting network instances. Additional logic is required to connect the core to the PLB. A parameterizable *IP interface* (IPIF, provided by Xilinx as a soft IP-core) provides the glue logic to connect the user component to the bus. In particular, it implements the bus protocol and handles bus arbitration and DMA transfers.

In order maximize performance while minimizing the CPU load we use DMA transfers initiated by the median core to access the memory. The DMA controller is implemented in the IPIF logic of our core. For our experiments we generate random data corresponding to the input stream in the external memory from a program running on the PowerPC core. Next, the CPU sets up the DMA transfers to read the input data from memory and to write back the median results. The two DMA transfers are setup by specifying the start addresses of the input data and result data as well as the number of items to process. Note that the output data size is equal to the input data size. The CPU communicates these parameters to median core by writing them into three memory-mapped registers of the median core. The logic for the registers is also implemented in the IPIF. We implemented the core such that the processing is started implicitly after writing the size register. The processing consists of three phases shown in Figure 6.26.

(1) A read transfer moves the data from the external memory into the median core. For maximum efficiency the full 128-bit width of the bus is used. In other words 16 bytes are sent over the bus per clock cycle. The PLB operates at 100 MHz resulting in a peak bus bandwidth of 1,600 MB/sec. During for each clock cycle 4 32-bit elements are received by the aggregation. This is the

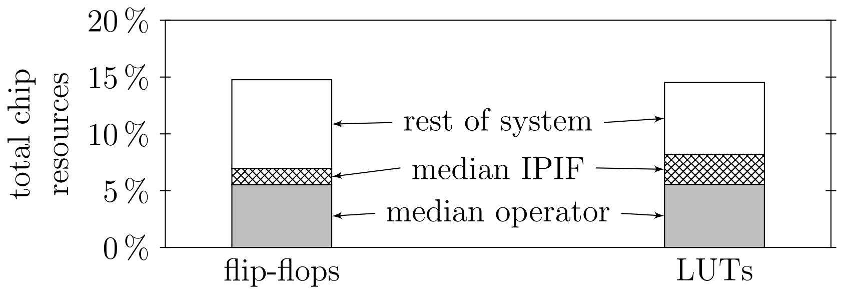


Figure 6.27: Distribution of FPGA resources in a system with median core

reason why we designed the aggregation core in Figure 6.24 such that it can process four items at once.

(2) The sorting network is fully pipelined hence we can process the input data immediately as it arrives. The median computation is performed in a pipelined manner for four elements in parallel. The PLB and IPIF only permit one active DMA transfer at any given time, hence, we need store the result data for the later write-back DMA transfer to external memory. We implement this on-chip buffer as a FIFO memory. The size of the memory is equal to maximum transfer size of 4,080 bytes supported by the controller. Our logic splits larger data sizes in multiple read/write DMA transfers without CPU involvement.

(3) After completing a data chunk the result data in the FIFO buffer is written back to memory by a write DMA transfer. After a 4,080-byte chunk is complete the next data chunk is read into the core (phase 1). After the last chunk is processed the median core signals completion to the CPU by rising an interrupt.

6.6.4 Evaluation

Resource Usage. The entire system occupies 28 % of the FPGA slices available on Virtex-5 FX130T chip. This includes not only the median core but also all additional soft IP-cores that are implemented using FPGA logic, for example, the memory controller, processor bus, and UART core. This figure does not include used components that are available in discrete silicon (hard IP-cores), such as the PowerPC core and block RAM. The design tools report 225 kB of block RAM memory used, which corresponds to 17 % of the available on-chip memory.

We further analyzed the resources used by the median core itself. In particular, how much is spent for implementing the interface to the PLB and the DMA logic. As for the evaluation in Section 6.4 we replace the operator implementation by a simple “route-through” logic, synthesize the design and compute the difference in flip-flops and LUTs to the complete design in order to estimate the chip resources used by the median operator alone. Figure 6.27 shows the distribution of flip-flop registers and lookup tables. The median core consisting of four 8-element even-odd sorting networks the sliding window, and control logic that schedules inserting data into the sorting network and extracting the results occupies about 40 % of all flip-flops and LUTs used by design. Also shown in Figure 6.27 is the space required for IP-core interface (IPIF) implementing the PLB interface and DMA transfer. 10 % of flip-flops and 18 % of the lookup tables are spent for the IPIF. Approximately half of the overall chip resources were used for logic unrelated to the median operator (memory controller, PLB, UART, etc.).

The implementation of median core is dominated by LUT usage. The median core uses 8.2 % of the LUTs available on the Virtex-5 FX130T FPGA while rest of the system occupies 6.3 % of chip’s LUTs. Hence, from a space perspective, it can be estimated that 11 instances of the median core can be placed on the chip. However, the number of components that can be connected to the PLB for DMA operations is limited to eight. Since the DMA bus functionality is also used by the PowerPC core this leads to at most 7 median cores that can be instantiated under the current system design.

Performance. The overall performance of the system is determined by the clock frequency as well as the latency and bandwidth limitations of the bus and memory interface. We operate the median core at the core system clock of 100 MHz. This clock is determined by other system components. In particular, the DDR2 memory controller is very sensitive to timing errors. Although, the sorting network operates at significantly lower clock speed compared to the evaluation in Section 6.4 (100 MHz vs 267 MHz) the design of the median operator still allows us to process a full bus width of data every cycle. Performance limitations is not due to the aggregation core but to the other system components.

The hardware implementation of the median operators requires 1.302 seconds for processing the 256 MB data set. While processing this data set 2 $\times$ 256 MB are transferred, once from external memory into the sorting network and once from the on-chip memory holding the result data back to the external memory. This leads to an effective end-to-end throughput of 393 MB/sec. Putting this figure in contrast to the peak bandwidth of the DDR2 memory (3,200 MB/sec) and the PLB (1,600 MB/sec) there is an significant loss. The memory is accessed sequentially and we paid special care to avoid contention on the processor bus. For example, we made sure that the PLB is not occupied by other components (e.g., the CPU) during while the median core is processing data. We believe that the observed reduction in bandwidth is due to the arbitration overhead of the bus.

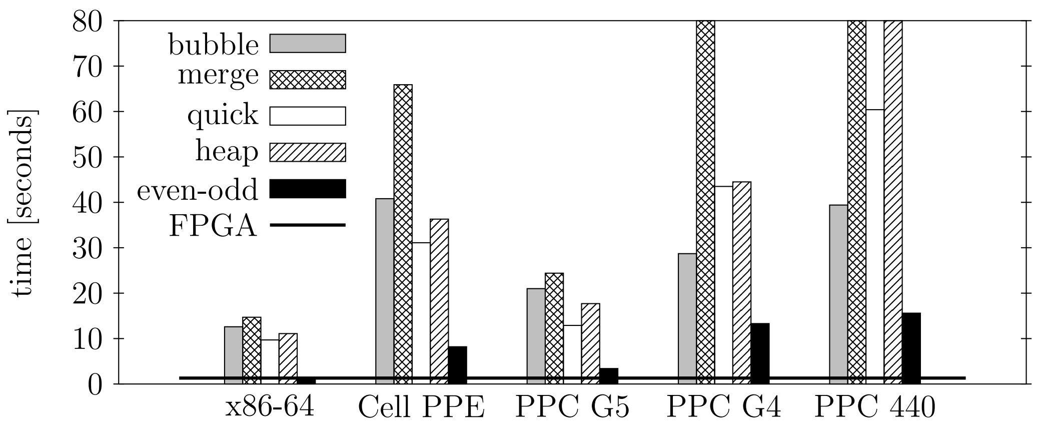


Figure 6.28: Execution time for computing the stream median of a 256 MB data set on different CPUs using different sorting algorithms and on the FPGA

FPGA Performance in Perspective. FPGAs can be used as co-processors of data processing engines running on conventional CPUs. This, of course, presumes that using the FPGA to run queries or parts of queries does not result in a net performance loss. In other words, the FPGA must not be slower than the CPU. Achieving this is not trivial because of the much slower clock rates on the FPGA.

Here we study the performance of the FPGA compared to that of CPUs. To ensure that the choice of a software sorting algorithm is not a factor in the comparison, we have implemented eight different sorting algorithms in software and optimized them for performance. Seven are traditional textbook algorithms: quick sort, merge sort, heap sort, gnome sort, insertion sort, selection sort, and bubble sort. Building accelerators with FPGAs is a complex and nontrivial processing. In order to perform a fair comparison we deliberately spent a considerable effort to also optimize the CPU-based implementations. The eighth implementation is based on the even-odd merge sorting network shown in Figure 6.25 using CPU registers. We implemented the sorting network using the assembly code variant shown in Section 6.3.5. As for the hardware implementation we applied the same optimization of the sorting that are possible for computing the lower median, i.e., removing comparator stages and optimizing the assembly code for "half-comparators". This process has lead to a hand-written, fully optimized and branch-free implementation of median computation in assembly language. The PowerPC implementation consists of 88 instructions. For Intel x86-64 we end up with 61 instructions.

We ran the different algorithms on several hardware platforms. We used an off-the-shelf Intel x86-64 CPU (2.66 GHz Intel Core2 quad-core Q6700) and the following PowerPC CPUs: a 1 GHz G4 (MCP7457) and a 2.5 GHz G5 Quad (970MP), the PowerPC element (PPE not SPEs) of the Cell, and the embedded 440 core of our FPGA. All implementations are single-threaded. For illustration purposes, we limit our discussion to the most relevant subset of algorithms.

Figure 6.28 shows the wall-clock time observed when processing 256 MB (as 32-bit tuples) through the median sliding window operator shown in Figure 6.23. The horizontal line indicates the execution time of the FPGA implementation. Timings for the merge and heap sort algorithms on the embedded PowerPC core did not fit into scale (162 s and 92 s, respectively). All our software implementations were clearly CPU-bound. It is also worth noting that given the small window, the constant factors and implementation overheads of each algorithm predominate and, thus, the results do not match the known asymptotic complexity of each algorithm. The best CPU result is obtained for the hand-written even-odd merging implementation on the Intel Core2 Q6700. Processing 256 MB requires 1.314 sec.

The performance observed indicates that the implementation of the operator on the FPGA is able to slightly outperform a modern conventional CPU. Processing 256 MB requires 1.314 sec on the Intel Core2, compared to 1.302 s on the FPGA. This is a bit discouraging result given the large effort spent for the FPGA implementation. As we already pointed out Section 6.4, the sorting network itself is very fast. Here, the comparatively low full-system performance is due to the currently available system components and bus design. Building FPGA-based accelerators is, not unlike to GPUs, difficult as the culprit is the same; getting data to and from the device. We can show that if directly combined with I/O FPGA can lead to significant performance improvements over traditional information systems. In the next chapter and in [MTA09b] we build a 1 gigabit network interface (UDP/IP) for the FPGA and combined it with a data processing engine on the same chip. This allows us to process data at wire speed. In [ME09] we apply the same technique to object deserialization. Nevertheless, being not worse than CPUs the FPGA is a viable option for offloading data processing out of the CPU, which then can be devoted to other purposes. When power consumption and parallel processing are factored in, FPGAs look even more interesting as co-processors for data management.

Power Consumption. While the slow clock rate of our FPGA (100 MHz) reduces performance, there is another side to this coin. The *power consumption* of a logic circuit depends linearly on the frequency at which it operates (U and f denote voltage and frequency, respectively):

$$P \propto U^2 \times f \ .$$

Therefore, we can expect our 100 MHz circuit to consume significantly less energy than a 3.2 GHz x86-64 CPU. It is difficult to reliably measure the power consumption of an isolated chip. Instead, we chose to list some approximate figures in

Intel Core 2 Q6700:	
Thermal Design Power (CPU only)	95 W
Extended HALT Power (CPU only)	24 W
Measured total power (230 V)	102 W
Xilinx ML510 development board:	
Calculated power estimate (FPGA only)	10.0 W
Measured total power (230 V)	40.8 W

Table 6.3: Power consumption of an Intel Q6700-based desktop system and the Xilinx ML510 FPGA board. Measured values are under load when running the median computation.

Table 6.3. Intel specifies the power consumption of our Intel Q6700 to be between 24 and 95 W (the former figure corresponds to the "Extended HALT Powerdown State") [Q6707]. For the FPGA, a power analyzer provided by Xilinx reports an estimated consumption of 10.0 W. A large fraction of this power (5.3 W) is used to drive the outputs of the 234 pins that connect the FPGA chip. CPUs with such large pin counts have the same problem. Additional power is also spent for the PowerPC (0.8 W) and the different clock signals (0.5 W).

More meaningful from a practical point of view is the overall power requirement of a complete system under load. Therefore, we took both our systems, unplugged all peripherals not required to run the median operator and measured the power consumption of both systems at the 230 V wall socket. As shown in Table 6.3, the FPGA has a 2.5-fold advantage (40.8 W over 102 W) compared to the CPU-based solution here. The power values are significantly higher for the Virtex-5 board that the 8.3 W wall power what we reported in [MTA09a] where we used a smaller board with a Virtex-II Pro FPGA. The higher power consumption is only partially due to increased power requirements of Virtex-5 FPGA. The new ML510 board also contains additional and faster components, which even when inactive their quiescent currents lead to a higher overall power consumption. Additionally, the 250 W ATX power supply we use for the ML510 board is a switching power supply, which is known to have a low efficiency when operated significantly below the nominal power (16 % in our case). A power-aware redesign of the board and the use of a matching power supply can reduce the power consumption much below 40 W.

As energy costs and environmental concerns continue to grow, the consumption of electrical power (the "carbon footprint" of a system) is becoming an increasingly decisive factor in the system design. Though the accuracy of each individual number in Table 6.3 is not high, our numbers clearly show that adding a few FPGAs can be more power-efficient than simply adding CPUs in the context of many-core architectures.

Modern CPUs have sophisticated power management such as dynamic frequency and voltage scaling that allow to reduce idle power. FPGAs offer power management even beyond that, and many techniques from traditional chip design can directly be used in an FPGA context. For example, using *clock gating* parts of the circuit can be completely disabled, including clock lines. This significantly reduces the idle power consumption of the FPGA chip.

6.7 Use Case: A Sorting Co-Processor

In the second use case we directly integrate the sorting network into a system. We built an 8-element even-odd merging network and connect it to the PowerPC core. Instead of connecting it over the *processor local bus* (PLB) and mapping the core into the main memory seen by the CPU the sorting core is implemented as an *Auxiliary Processor Unit* (APU). The APU interfaces was described in Section 5.5.2 (page 161). This use case illustrates another approach to integrate an FPGA accelerator into a heterogeneous system.

6.7.1 Heterogeneous Merge Sort

The hardware solutions described before have the disadvantage that they can only operate on a fixed-length data set. As a workaround that allows variable sized input make used of the CPU to merge chunks of data that is sorted in hardware. The sorting algorithm of choice here is *merge sort*. In this use case we implement an 8-element even-odd merging network as a sort core in the APU. The FPGA accelerator will sort consecutive blocks of 8 elements in-place. The sorted blocks are then merged on the CPU as shown in Figure 6.29. This corresponds to a merge sort where the lowest $L = 3$ (leaf) levels are performed by the APU. For sorting N elements in total $\lceil \log_2(N) \rceil$ are required. Hence, for $N = 2^p$ elements, $p - L$ merge levels in software are needed.

6.7.2 Sorting Core connected to the CPU Execution Pipeline

We access our sorting core through load/store instructions. The FCM contains two 16-byte registers `s0` and `s1` that are able to store 4 elements each. The CPU code for sorting 8-elements is shown below:

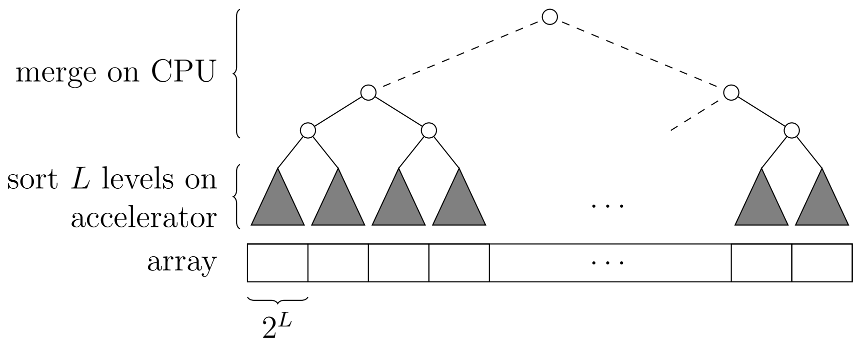


Figure 6.29: Heterogeneous merge sort where the lowest L levels are performed on the FPGA accelerator

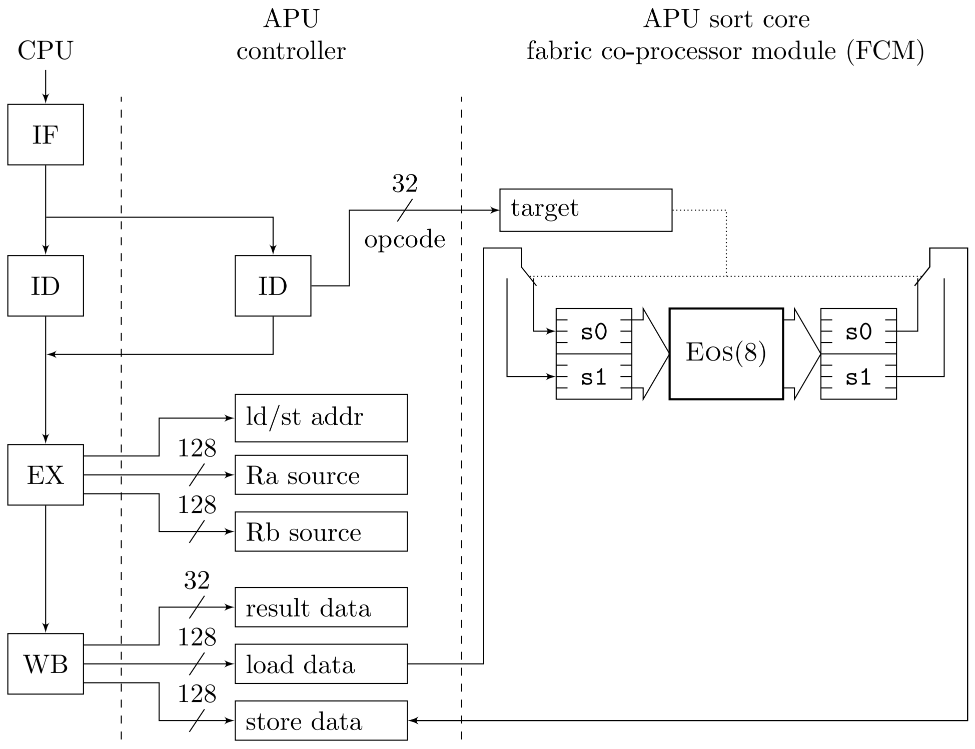


Figure 6.30: Sort core as *Auxiliary Processor Unit*

APU Assembly	
	`r8` ← address of input array
	`r9` ← address of output array
	`r10` ← 16 stride in bytes
`ldfcmux s0,r8,r10`	`s0` ← mem[`r8` + `r10`],
	`r8` ← `r8` + `r10`
`ldfcmx  s1,r8,r10`	`s1` ← mem[`r8` + `r10`]
...	*6 additional writes to* `s1`
`stfcmux s0,r9,r10`	mem[`r9` + `r10`] ← `s0`,
	`r9` ← `r9` + `r10`
`stfcmx  s1,r9,r10`	mem[`r9` + `r10`] ← `s1`

For sorting 8 values on the APU, we first load the input data from memory into `s0` and `s1`. The data loaded by using to instruction. `s0` corresponds to the first 4 elements, while the last 4 elements are written to `s1`. The `ldfcmux` instruction also updates the first source register operand whereas `ldfcmx` does not. We designed the FCM such that after writing `s1` the content [`s0`,`s1`] is fed into the sorting network (Eos(8)). The sorting network is implemented following a fully-pipelined synchronous design. In order to simplify instruction scheduling we clock the sorting network based on writing `s1`, i.e., after writing 6× to register `s1` the sorted output appears at the output of the sorting network. The sorted output is written back to memory using a FCM store instruction. Note that in fact `s0` and `s1` each refer to two different registers when loading or storing (see Figure 6.30). We can hide this 6-instruction latency by using *software pipelining* in the assembly program.

6.7.3 Evaluation

We evaluate the APU implementation and compare it to a CPU-only version of the merge sort algorithm running on the PowerPC 440 core. Figure 6.31 shows the speedup of the hardware acceleration for sorting arrays containing 256–16M elements.

The speedup decreases asymptotically as the size of the data set increases. The reason is that ratio between the work done by the CPU to work done in the accelerator decreases as the data set increases as the following simple analysis shows. Let $T(n)$ be the time to sort n elements. The recursive merge sort leads to recurrence equation $T(n) = 2T(n/2) + n$. By considering only $n = 2^p$, i.e., power of twos, we obtain the following recursive definition:

$$T(2^p) = 2T(2^{p-1}) + 2^p \quad \text{with} \quad T(2^1) = 2 \ .$$

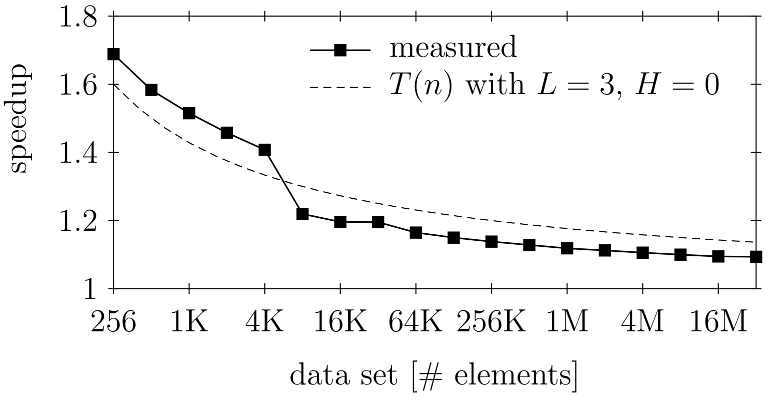


Figure 6.31: Speedup of APU sorting core over traditional on chip sort

The solution to this recurrence equation is $T(2^p) = p2^p$. This corresponds to the execution time for the CPU-only implementation. Now, with the hardware acceleration the lowest L levels are performed in the APU, say spending a processing time H for each 2^L-element set. This changes the initial condition of the recurrence equation to $T(2^L) = H$. As it can easily be verified, the solution for the accelerated implementation is $T'(2^p) = H2^{p-L} + (p - L)2^p$. Hence, the speedup is

$$\text{speedup} = \frac{T(2^p)}{T'(2^p)} = \frac{p}{H2^{-L} + p - L} \xrightarrow{H=0} \frac{p}{p - L}$$

Figure 6.31 also shows the predicted speedup for $H = 0$, i.e., the operation performed by the accelerator requires no time. Clearly, it follows that for large datasets the speedup decreases to 1. The sharp decrease can see between 4K and 8K is the effect of the 32 kB data cache on the core. Using the aggregation the cache pressure can be reduced as no temporary data needs to be kept, hence, as long as the data fits into the cache the aggregation core can provide higher speedups. Although, the absolute values of the speedup obtained is not particularly high, this use case illustrates how a tightly coupled accelerator can be implemented in a heterogeneous system. Also, it is another attempt to release the power of sorting network in hardware we observed in Section 6.4.

6.8 Summary

In this chapter, we have assessed the potential of FPGAs as a computing platform, in particular, to accelerate sorting. We presented different approaches to implement sorting networks on FPGAs and discussed the on-chip resource utilization. Despite the complexity involved with designs at the hardware level the flip-flop and LUT utilization of a circuit can be estimated beforehand, in particular, for synchronous fully-pipelined implementations. We also showed how FPGAs can be used as a co-processor for data intensive operations in the context of multi-core systems. We have illustrated the type of data processing operations where FPGAs have performance advantages (through parallelism, pipelining and low latency) and discussed different ways to embed the FPGA into a larger system so that the performance advantages are maximized. Our evaluation shows that implementations of sorting networks on FPGAs do lead a high performance (throughput and latency) on their own. However, the two use cases put these high performance numbers into perspective. It is challenging to maintain this performance once the hardware implementation of the algorithm is integrated into a full system. The design of a hardware accelerator and the system integration is nontrivial and may not immediately lead to a performance advantage over a conventional software-based solution on modern high-end CPU. The use of an FPGA as a co-processor requires a carefully designed data path in order to reduce the cost for exchanging data between CPU and co-processor. In both use cases presented in this chapter the performance was limited by the communication bandwidth either over the memory or the APU interface, respectively. In the next chapter, we are studying a different attachment. As we will see, we can significantly improve matters for the FPGA when inserting it into the data path, e.g., to perform processing on the data path from and to the network.

Next to raw performance, our experiments also show that FPGAs bring additional advantages in terms of power consumption. These properties make FPGAs very interesting candidates for acting as additional cores in the heterogeneous many-core architectures that are likely to become pervasive.

The analysis in this chapter is a first but important step to incorporate the capabilities of FPGAs into data processing engines in an efficient manner. The higher design costs of FPGA-based solutions may still amortize, for example, if a higher throughput (using multiple parallel processing elements as shown in Section 6.6) can be obtained in a FPGA-based stream processing system for a large fraction of queries. If performance is of paramount importance and design costs can be neglected more efficient FPGA designs can be obtained by manual placement of the circuits on the chip. From the observation made in this chapter we set up the scene for the next chapter where we describe how to queries can be automatically translated into hardware circuits. We will consider synchronous and pipelined designs and use automatic placement provided by the vendor tools.

7 Query-to-Hardware Compiler

In this chapter, we present *Glacier*, a library of components and a basic compiler for continuous queries implemented on top of an FPGA. The ultimate goal of this line of work is to develop a hybrid data stream processing engine where an optimizer distributes query workloads across a set of CPUs (general-purpose or specialized) and FPGA chips. In this chapter describe on how conventional streaming operators can be mapped to circuits on an FPGA; how they can be combined into queries over data streams; the proper system setup for the FPGA to operate in combination with an external CPU; and the actual performance that can be reached with the resulting system. The chapter has the following outline.

1. We describe *Glacier*, a component library and compiler for FPGA-based data stream processing. Besides classical streaming operators, *Glacier* includes specialized building blocks needed in the FPGA context. With the operators and specialized building blocks, we show how *Glacier* can be used to produce FPGA circuits that implement a wide variety of streaming queries.

2. Since FPGAs behave very differently than software, we provide an in-depth analysis of the *complexity and performance* of the resulting circuits. We discuss *latency* and *issue rates* as the relevant metrics that need to be considered by an optimizing plan generator.

3. Finally, we evaluate the *end-to-end performance* of an FPGA inserted between the network and the CPU, running a query compiled with *Glacier*. Our results show that the FPGA can process streams at a rate beyond one million tuples per

second, far more than the CPU could. These results demonstrate the potential of FPGAs as co-processors in engines running on many-core architectures.

7.1 Streams in Software

7.1.1 Motivating Application

Our running example is based on a collaboration with a Swiss bank. Their financial trading application receives data from a set of streams with up-to-date market information from different stock exchanges. The information is distributed via UDP messages and in small packages in order to reduce latency. The main challenge is the data rate at which messages arrive. By the end of next year OPRA, the Option Price Authority that collects and distributes real-time data from different stock exchanges, predicts the message rate to approach 3 million messages per second [OPR09].

Traditional techniques such as load shedding [TcZ$^+$03] cannot be applied in trading applications because of potential financial loss. This is particularly true in peak situations, which typically indicate a turbulent market situation. At the same time, latency is critical and measured in units of microseconds.

To abstract from the real application in this discussion, we assume an input stream that contains a reduced set of information about each trade handled by Eurex (the actual streams are implemented as a compressed representation of the feature-rich FIX protocol [FIX09]). Expressed in the syntax of StreamBase [Str], the schema of our artificial ticker stream would read:

```
CREATE INPUT STREAM Trades (
  Seqnr  int,         -- sequence number
  Symbol string(4),   -- valor symbol
  Price  int,         -- stock price
  Volume int)         -- trade volume
```

To keep matters simple, we look at queries that process a single data stream only. To facilitate the allocation of resources on the FPGA, we restrict ourselves to queries with a predictable space requirement. We do allow aggregation queries and windowing; in fact, we particularly look at such functionality in the second half of Section 7.3.

These restrictions can be lifted with techniques that are no different to those applied in software-based systems. The necessary FPGA circuitry, however, would introduce additional complexity and only distract from the FPGA-inherent considerations that are the main focus of this work.

7.1.2 Example Queries

Our first set of example queries is designed to illustrate a hardware-based implementation for the most basic operators in stream processing. Queries Q_1 and Q_2 use simple projections as well as selections and compound predicates:

```
SELECT Price, Volume
  FROM Trades
 WHERE Symbol = "UBSN"                          (Q1)
  INTO UBSTrades

SELECT Price, Volume
  FROM Trades
 WHERE Symbol = "UBSN" AND Volume > 100000      (Q2)
  INTO LargeUBSTrades
```

Financial analytics often depend on statistical information from the data stream. Using sliding-window and grouping functionality, Query Q_3 counts the number of trades of UBS shares over the last 10 minutes (600 seconds) and returns the aggregate every minute. In Query Q_4, we assume the presence of an aggregation function `wsum` that computes the weighted sum over the prices seen in the last four trades of UBS stocks (similar functionality is used, e.g., to implement finite-impulse response filters). Finally, Query Q_5 determines the average trade prices for each stock symbol over the last ten-minutes window:

```
SELECT count () AS Number
  FROM Trades [SIZE 600 ADVANCE 60 TIME]
 WHERE Symbol = "UBSN"                          (Q3)
  INTO NumUBSTrades

SELECT wsum (Price, [.5, .25, .125, .125]) AS Wprice
  FROM (SELECT * FROM Trades
          WHERE Symbol = "UBSN")                (Q4)
       [SIZE 4 ADVANCE 1 TUPLES]
  INTO WeightedUBSTrades

SELECT Symbol, avg (Price) AS AvgPrice
  FROM Trades [SIZE 600 ADVANCE 60 TIME]
 GROUP BY Symbol                                (Q5)
  INTO PriceAverages
```

We use these five queries in the following to demonstrate various features as well as the compositionality of the *Glacier* compiler.

Table 7.1: Supported streaming algebra (a, b, c: field names; q, q_i: sub-plans; x: parameterized sub-plan input)

$\pi_{a_1,\ldots,a_n}(q)$	projection
$\sigma_a(q)$	select tuples where field a contains `true`
$\circledast_{a:(b_1,b_2)}(q)$	arithmetic/Boolean operation $a = b_1 \star b_2$
$q_1 \cup q_2$	union
$agg_{b:a}(q)$	aggregate agg using input field a, $agg \in \{$`avg`, `count`, `max`, `min`, `sum`$\}$
$q_1 \ \mathsf{grp}_{x\|c}\ q_2(x)$	group output of q_1 by field c, then invoke q_2 with x substituted by the group
$q_1 \boxplus^{t}_{x\|k,l}\ q_2(x)$	sliding window with size k, advance by l; apply q_2 with x substituted on each wind.; $t \in \{\text{time}, \text{tuple}\}$: time-, or tuple-based
$q_1 \infty q_2$	concatenation; position-based field join
$q_1 \bowtie_p q_2$	window join; join predicate p

7.1.3 Algebraic Plans

Input to our compiler is a query representation in an algebra for streaming queries. Our compiler currently supports the algebra dialect listed in Table 7.1, whereby operators may be composed in an arbitrary fashion. Our algebra uses an "assembly-style" representation that breaks down selection, arithmetics, and predicate evaluation into separate algebra operators. In the context of the current work, the notation turns out to have nice correspondences to the data flow in a hardware circuit and helps detecting opportunities for parallel evaluation.

Figure 7.1 illustrates how our streaming algebra can be used to express the semantics of Queries Q_1 through Q_5. Observe how in Figure 7.1(a), e.g., operator $\ominus$ makes the comparison of each input tuple with the requested stock symbol "UBSN" explicit. Its output, the new column a, is used afterwards to filter out non-qualifying tuples (operator σ_a).

The *concatenate operator* ∞ represents what could be called a "join by position". Tuples from both input streams are combined into a wide result tuple in the order in which they arrive. The operator is necessary, for instance, to evaluate and return different aggregation functions over the same input stream.

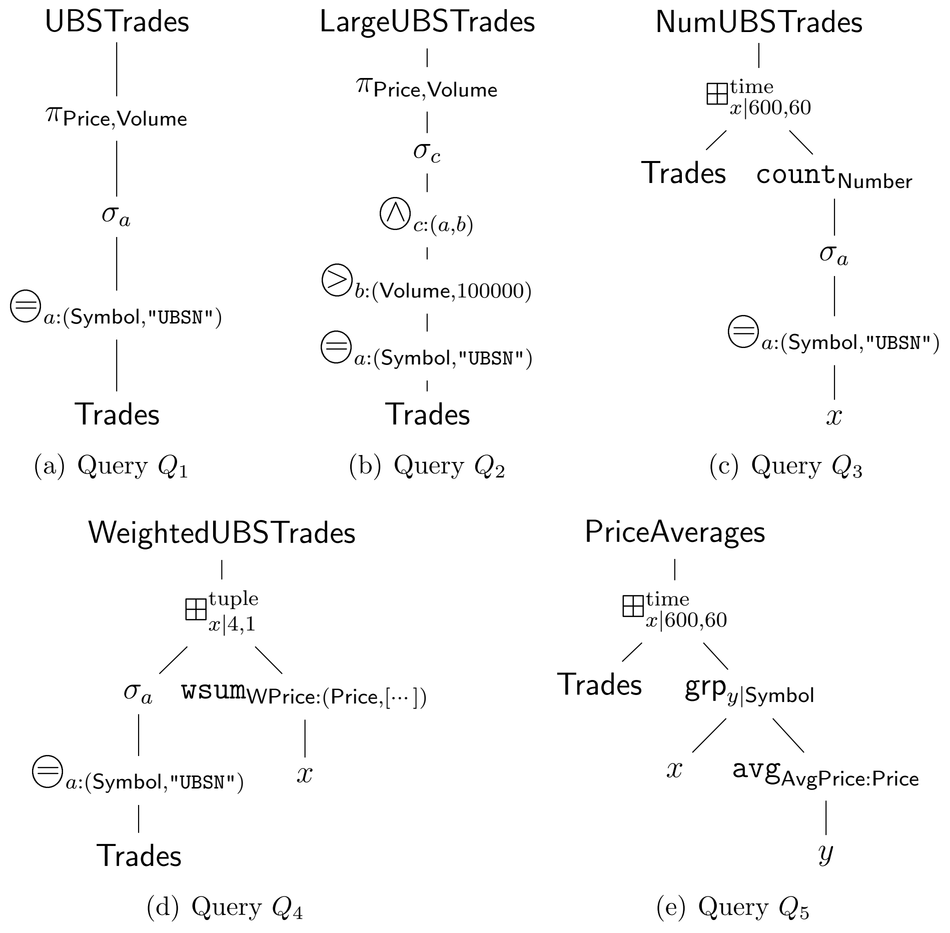

UBSTrades
$\pi_{\text{Price,Volume}}$
σ_a
$\ominus_{a:(\text{Symbol},\texttt{"UBSN"})}$
Trades
(a) Query Q_1
LargeUBSTrades
$\pi_{\text{Price,Volume}}$
σ_c
$\owedge_{c:(a,b)}$
$\oslash_{b:(\text{Volume},100000)}$
$\ominus_{a:(\text{Symbol},\texttt{"UBSN"})}$
Trades
(b) Query Q_2
NumUBSTrades
$\boxplus^{\text{time}}_{x|600,60}$
Trades
$\texttt{count}_{\text{Number}}$
σ_a
$\ominus_{a:(\text{Symbol},\texttt{"UBSN"})}$
x
(c) Query Q_3
WeightedUBSTrades
$\boxplus^{\text{tuple}}_{x|4,1}$
σ_a
$\texttt{wsum}_{\text{WPrice:(Price,}[\cdots])}$
$\ominus_{a:(\text{Symbol},\texttt{"UBSN"})}$
x
Trades
(d) Query Q_4
PriceAverages
$\boxplus^{\text{time}}_{x|600,60}$
Trades
$\text{grp}_{y|\text{Symbol}}$
x
$\texttt{avg}_{\text{AvgPrice:Price}}$
y
(e) Query Q_5

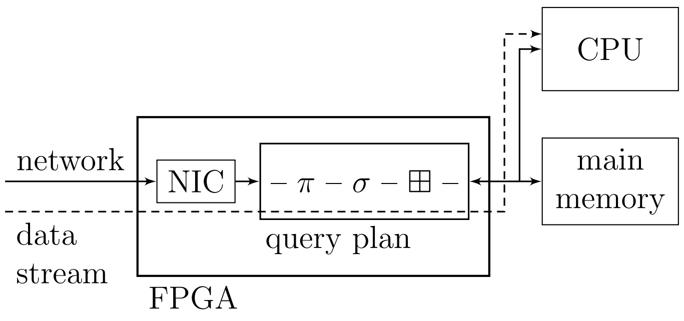
Figure 7.1: Algebraic query plans for the five example queries Q_1 to Q_5

7.2 FPGAs for Stream Processing

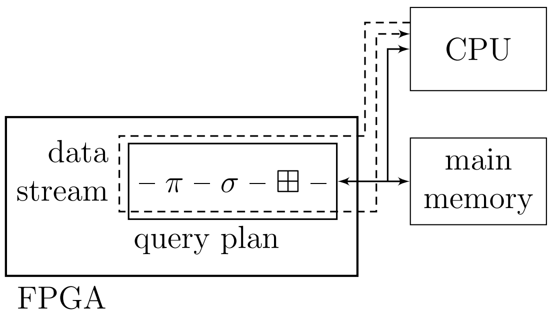
7.2.1 System Setup

FPGAs can mimic arbitrary logic functionality by mere reconfiguration. In contrast to existing special-purpose hardware (such as graphics or floating-point processors), this makes the role of an FPGA inside the overall system not predetermined. By implementing the respective bus protocols, e.g., FPGAs can be connected to memory or peripheral buses, communicate with external devices, or any combination thereof.

Figure 7.2 shows the two possible configurations of how the query circuits can interact with a host CPU that runs a traditional stream processing engine. Figure 7.2(a) the FPGA is directly connected to the physical network interface,



(a) Stream engine between network interface and CPU.



(b) Stream engine as a coprocessor to the CPU.

Figure 7.2: System architectures of FPGA integration with the host CPU

with parts of the network controller implemented inside the FPGA fabric. After reception, data from the network is directly fed into the hardware implementation of a database query plan. The host CPU only becomes involved once result items have been produced for the user query. Using DMA, the *Glacier* circuit writes the result tuples from the FPGA into the system main memory, then informs the host CPU about the arrival of new data (e.g., by raising an interrupt).

Alternatively, the FPGA can also be used in a traditional co-processor setup, as illustrated in Figure 7.2(b). Here, the CPU hands over data to the FPGA either by writing directly into FPGA registers (so-called *slave registers*) or it prepares the input data into a shared RAM region, then sends a *work request* to the FPGA-based co-processor. We described both attachments in Sections 5.4 and 5.5.

The architecture in Figure 7.2(a) fits a pattern that is highly common in data stream applications. Oftentimes, rate-reducing filtering or aggregation stages precede more complex high-level processing (done on the CPU). Even simple filter stages, fully supported by the algebra dialect of *Glacier*, suffice to significantly reduce the load on the back-end CPU. In algorithmic trading, for instance, they discard about 90 % of all input data. Only the remaining 10 % of the data actually hits the CPU, which significantly increases the applied load that the system can

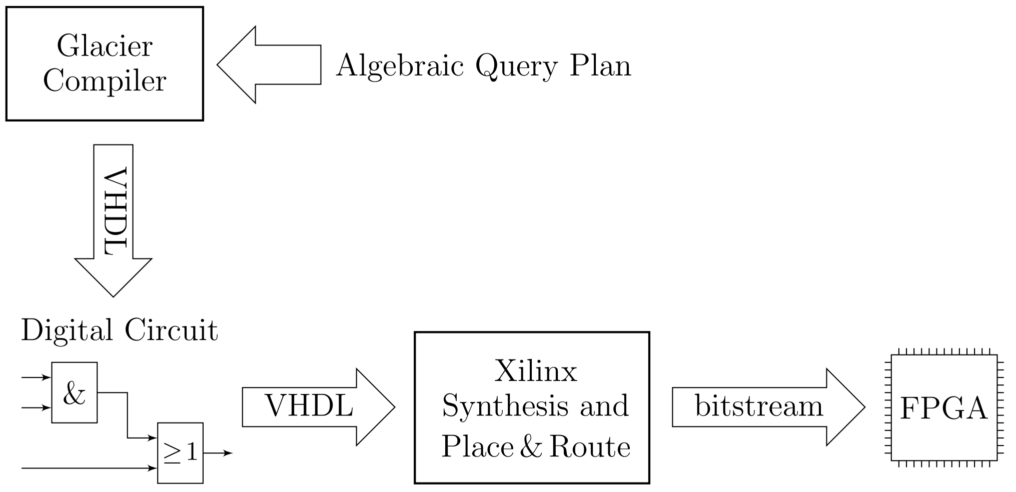


Figure 7.3: Compilation of abstract query plans into hardware circuits for the FPGA

sustain.

7.2.2 Query Compilation

Figure 7.3 illustrates the compilation process from algebraic plans to FPGA circuits. The input to the *Glacier compiler* are algebraic plans of the kind introduced in Section 7.1.3. The compiler applies compilation rules (Section 7.3) and optimization heuristics (Section 7.5), then emits the description of a logic circuit that implements the input plan.

The generated circuits are expressed in VHDL hardware description language. The VHDL code is fed to the Xilinx tool chain (see Section 5.2.2 on page 154), which creates the actual low-level, FPGA-specific representation of the circuit (configuration of the LUTs and the interconnect network). The output of the synthesizer is then used to program the FPGA. In Figure 7.3, the compilation of VHDL code into an FPGA configuration follows the usual design flow in traditional FPGAs design. Using the *Glacier* compiler, the creation of VHDL code can be fully automated.

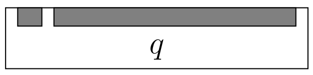
7.3 From Queries to Circuits

Using pre-built components from the *Glacier* library, each operator in Table 7.1 can be compiled into a hardware circuit in a systematic way. To ensure the full compositionality of the translation rules later in this section, every translated subplan adheres to the same well-defined wiring interface.

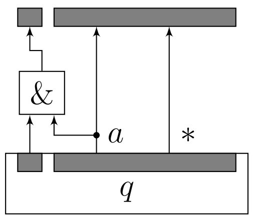
7.3.1 Wiring Interface

As in data streaming engines, our processing model is entirely push-based. Each n-bit-wide tuple is represented as a set of n parallel wires in the FPGA fabric. On a set of wires, a new tuple can be propagated in every cycle of the FPGA's system clock (i.e., 100 million tuples per second). An additional *data valid* line signals the presence of a tuple in a given clock cycle. Tuples are only considered to be part of the data stream if their *data valid* flag is set to true, i.e., if the *data valid* line carries an electrical "high" signal.

In the following, we use rectangles to represent logic components (with the exception of multiplexers, for which we use the common trapezoid notation). Our circuits are all clock-driven or *synchronized* and every operator in our library writes its output into a flip-flop register after processing. We indicate registers as gray-shaded boxes and make the *data valid* flag explicit as each operator's leftmost output. For instance, we depict the black-box view of a hardware implementation for a query q as shown on the right.



We use arrows to denote the wiring between hardware components. Wherever appropriate, we identify those lines from a tuple bus that correspond to a specific tuple field with a label at the respective input/output port. The label '$*$' stands for "all remaining fields". We do not represent the order of fields within a tuple. The hardware plan for the algebra expression $\sigma_a(q)$ can thus be illustrated as



.

In this circuit, the logical 'and' gate invalidates the output tuple whenever field a contains false.

Circuit Characteristics

The above circuit will compute its output in a single clock cycle and will be ready to consume a new input tuple at every tick of the clock. We say that its *latency* and *issue rate* are both 1. In general, circuits may need more than one cycle until the result of their computation can be picked up at the operator output—they have a latency that is larger than 1. Due to their semantics, circuits that implement grouping or windowing cannot produce output before they have seen the last tuple of the respective query window. For these operators, we define latency to be the number of clock cycles between the closing of the input window and the generation of the first output tuple.

We define the issue rate as the number of tuples that can be processed per clock cycle. The issue rate is always ≤ 1. For example, an operator that can accept a tuple every five cycles has an issue rate of 0.2.

Some operations can be *pipelined*. The corresponding circuits will be ready to consume new input already *before* the output of the preceding tuple has been fully computed. Their issue rate is higher than the reciprocal value of their latency. The terms latency and issue rate are also used in the system architecture domain [HP02]. Latency and issue rate are important parameters to determine the performance of a hardware circuit. Latency directly corresponds to the observable response time, whereas the issue rate determines throughput.

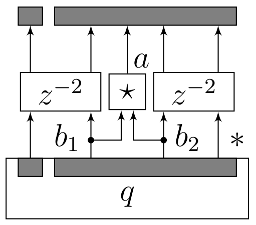
Synchronization

Both properties sometimes also need to be considered during query compilation. For instance, all compilation rules must ensure that a generated circuit will never try to push two tuples in successive cycles into an operator that has an issue rate less than one. We use two types of logic components to implement the synchronization between sub-circuits:

FIFO queues act as short-term buffers for streams with a varying data rate. They emit data at a predictable rate, typically the issue rate of an upstream sub-circuit. Note that, at runtime, the average input rate must not exceed what is achievable with the output rate.

In most practical cases, the depth of the FIFO can be kept very low. This not only implies a small resource footprint, but also means that the impact on the overall latency is typically small.

Delay operators z^{-n} can block data items for a fixed number of cycles n. This can be used, e.g., to properly synchronize the output of slow arithmetic operators with the remaining tuple flow (the circuit below implements $\circledast_{a:(b_1,b_2)}(q)$; assume that the latency of $\star$ is 2):



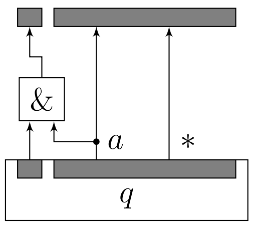
.

Equipped with notation, we are now ready to describe a complete set of compilation rules that covers the streaming algebra listed in Table 7.1. We start with rather simple operators. We defer the discussion of the window join operator to Section 7.7.2 due to its complexity and different semantics.

7.3.2 Selection and Projection

We saw earlier how our assembly-style selection operator σ_a can be cast into a hardware circuit. Compilation Rule Sel formalizes this translation in the notation we also use in the remainder of this work. We use the $\Mapsto$ symbol to indicate the "compiles to" relation and, as before, assume that a rectangle labeled q is the circuit that results from compiling q:

$\sigma_a(q) \quad \Mapsto$

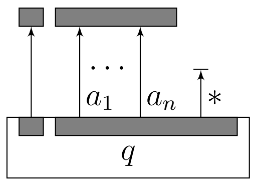


. (Sel)

Note that the resulting circuit leaves all tuples essentially intact, but invalidates discarded tuples by setting their *data valid* flag to false. This is very similar in nature to the "selection vectors" that MonetDB/X100 [HZdVB07] uses to avoid data copying. The logical 'and' gate $\boxed{\&}$ completes within a single cycle. Therefore, the latency and the issue rate of the circuit generated for σ_a are both 1.

Here, we use the projection operator $\pi_{a_1,\dots,a_n}$ to discard fields from the tuple flow. Support for field renaming (often expressed using the π operator) is a straightforward extension of what we present here. Discarding a field from the tuple flow simply means to not wire the respective output ports with any inputs further down the data path, as shown in Rule Proj:

$\pi_{a_1,\dots,a_n}(q) \quad \Mapsto$



. (Proj)

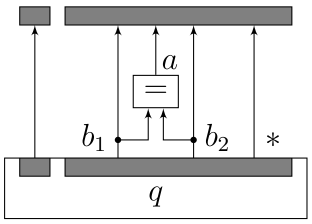
This implementation for $\pi_{a_1,\dots,a_n}$ has an interesting side effect. Our compiler emits the *description* of a hardware circuit that is passed into a synthesizer to generate the actual hardware configuration for the FPGA. The synthesizer optimizes out "dangling wires", effectively implementing *projection pushdown* for free. There is no actual work to do at runtime (though fields are propagated into a new set of registers). Latency and issue rate of this implementation for projection are both 1.

7.3.3 Arithmetics and Boolean Operations

As indicated in Table 7.1, we use the generic $\circledast_{a:(b_1,b_2)}$ operator to represent arithmetic computations, value comparisons or Boolean connectives in relational plans. The instance

$$\ominus_{a:(b_1,b_2)}(q) \ ,$$

e.g., will emit all fields in q, extended by a new field a that contains the outcome of $b_1 = b_2$. This semantics directly translates into an implementation in logic (we saw a similar circuit a moment ago):

$\ominus_{a:(b_1,b_2)}(q) \quad \Mapsto$ 

. (Equals)

Most simple arithmetic or Boolean operations will run within a single clock cycle. More complex tasks, such as multiplication/division, or floating-point arithmetics, may require additional latency. Sometimes, the actual circuit that implements $\boxed{\star}$ can be tuned within the trade-offs latency, issue rate, and chip space consumption. If the latency of $\boxed{\star}$ is greater than one, delay operators have to be introduced to synchronize the operator output with the remaining fields (as shown before in Section 7.3.1).

Example. With the rules we have seen so far, we can now translate our first example query into a hardware circuit. In Figure 7.4, we illustrated the circuit that results from applying our compilation rules to Query Q_1. The hardware circuit quite literally reflects the shape of the algebraic plan. Each of the operators can individually operate in a single cycle (i.e., have latency and issue rates of one). Since all plan operators are applied sequentially, latencies add up and the circuit in Figure 7.4 has an overall latency of three. By contrast, the issue rate of a pipelined execution plan is determined by its slowest sub-plan. Since all sub-plans have an issue rate of one, this is also the rate of the complete plan. □

7.3.4 Union

From a data flow point of view, the task of an algebraic union operator $\cup$ is to accumulate the output of several source streams into a single output stream. Since, in our case, all source streams operate truly in parallel, a hardware implementation for $\cup$ needs to ensure proper synchronization. We do so by buffering all input ports using FIFOs:

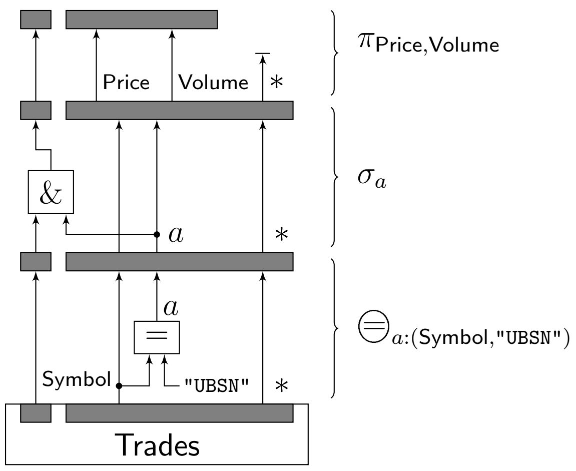
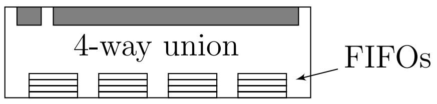
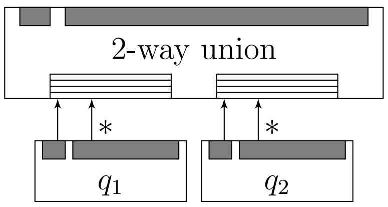

$\pi_{\text{Price,Volume}}$
Price
Volume
$*$
σ_a
&
a
$*$
$\ominus_{a:(\text{Symbol},\text{"UBSN"})}$
a
=
Symbol
"UBSN"
$*$
Trades

Figure 7.4: Compiled hardware execution plan for Query Q_1. Latency of this circuit is 3, issue rate 1.



A state machine inside the union component then forwards tuples from the input FIFOs in a round-robin fashion and emits them as the union result. Although every individual input may feed into the union component at an arbitrary tuple rate (i.e., issue rate 1), the average rate of all input streams together must not exceed more than one tuple per cycle, which is the maximum tuple rate that the union component can forward up-stream the data path. It is important that dequeuing of the four FIFOs shown above happens in a controlled and fair manner such that starvation can be avoided. For the implementation this means that the output register needs to be arbitrated among the individual FIFOs. Our implementation uses the well-known *round-robin token passing arbiter* [SMR02]. It guarantees fairness (no starvation) among the input FIFOs and in terms of latency, the resulting state machine inside the operator requires a single cycle to process. The FIFOs at the input, implemented using either flip-flop registers or block RAM (a resource trade-off), add another latency cycle. The overall circuit therefore has a minimum latency of 2. Depending on the input data distribution, however, the observed latency may be higher whenever tuples queue up in an input FIFO.

Strictly speaking, a binary union component is sufficient to implement the algebraic $\cup$ operator:

$q_1 \cup q_2 \quad \mapsto$  . (UNION)

As we will see in the following, however, the availability of a general, n-way union implementation eases the implementation of other functionality.

7.3.5 Windowing

The concept of windowing bridges the gap between stream processing and relational-style semantics. The operation $q_1 \; \boxplus_{x|k,l} \; q_2$ consumes the output of its left-hand sub-plan (q_1) and slices it into a set of *windows*. For each window, $\boxplus_{x|k,l}$ invokes a parameterized execution of the right-hand sub-plan $q_2(x)$, with each occurrence of x replaced by the current window. Sub-plan q_2 thus sees a finite input for every execution and may, e.g., use aggregation in a semantically sound manner.

Our compiler implements this semantics by wrapping q_2 into a template circuit (full compilation rule shown in Figure 7.5). We introduce an additional input signal eos ("end of stream") next to the *data valid*. It is asserted "high" when a window closes to notify the sub-plan that it has seen all elements of that window. The signal typically triggers the sub-plan to start generating output tuples.

A common use case are *sliding windows*, where input tuples belong to several windows at the same time. Here we can exploit the available *parallelism* on the FPGA chip. We replicate the hardware plan of q_2 as many times n as there may be windows open in parallel during query execution, plus 1. For time- and tuple-based windows, *e.g.*, we have that $n = \lceil k/l \rceil + 1$ (where k is the window size and l is the size of the slide). In Figure 7.5, we assume $n = 4$ (i.e., at most three windows open in parallel). To keep matters simple, we assume that k is a multiple of l; the extension to the general case is straightforward.

We use the *cyclic shift register* CSR1 (indicated as a dashed box in Figure 7.5) to keep track of window states. For every instance of the sub-plan q_2, this shift register carries the information whether the instance actively processes an open window. Figure 7.5 assumes that three windows are open in parallel, i.e., three bits are set in CSR1. Whenever the end of a window is reached, triggered by the "advance" signal adv the shift register rotates (to the right), such that the oldest open window is closed and a new one opened. The signal adv may be driven either by a clock (for time-based windows) or by a counter that implements tuple-based windows.

$$q_1 \boxplus_{x|k,l} q_2(x) \mapsto \qquad \text{(Wind)}$$

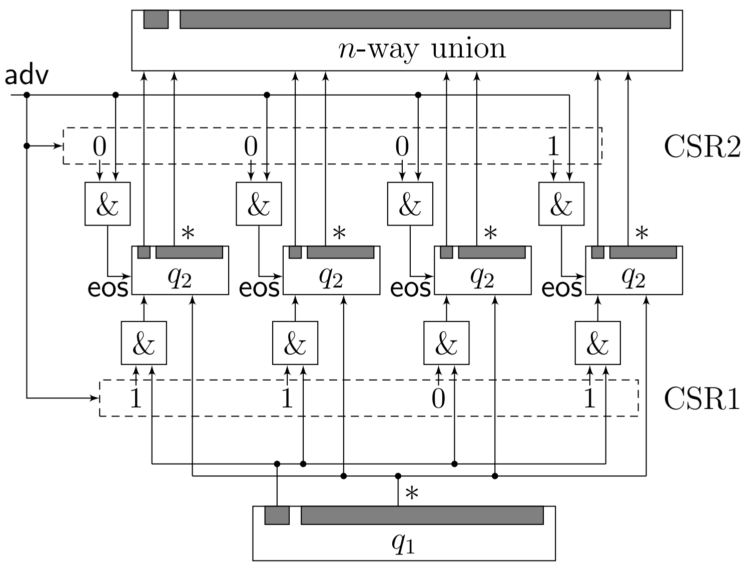


Figure 7.5: Compilation rule for windowing operator ⊞ (shown for an instance with at most three windows open in parallel)

Parallel to advancing CSR1, we send an eos signal to the sub-plan that processes the oldest open window. This sub-plan will then start producing output and feed it to the up-stream plan through a union operator. While doing so, the sub-plan will have the 0-bit in CSR1, i.e., it will not receive any new input while emitting tuples. To communicate the eos signal to the correct sub-plan, we use a second shift register CSR2, shifted in sync with CSR1. The single bit in CSR2 identifies the oldest open window.

Example. The hardware circuit that implements the sliding-window query Q_4 is shown in Figure 7.6. With the windowing clause `[SIZE 4 ADVANCE 1 TUPLES]`, at most four windows can be open together at any point in time. Hence, we instantiate five copies of the `wsum` sub-plan. The window type of this query is tuple-based. The *counter* component on the left counts incoming tuples and sends the adv signal as often as specified by the query's `ADVANCE` clause (in this particular case, `ADVANCE` = 1 and we could simplify our circuit by directly routing *data valid* to the adv line). □

Signal processing in the windowing part of the plan is implemented using combinational logic. It fits into a single clock cycle and is fully pipelineable. The latency of the overall circuit thus is the latency of the inner plan plus 2 (the latency of the n-way union operator). The issue rate is the one of the inner sub-circuit.

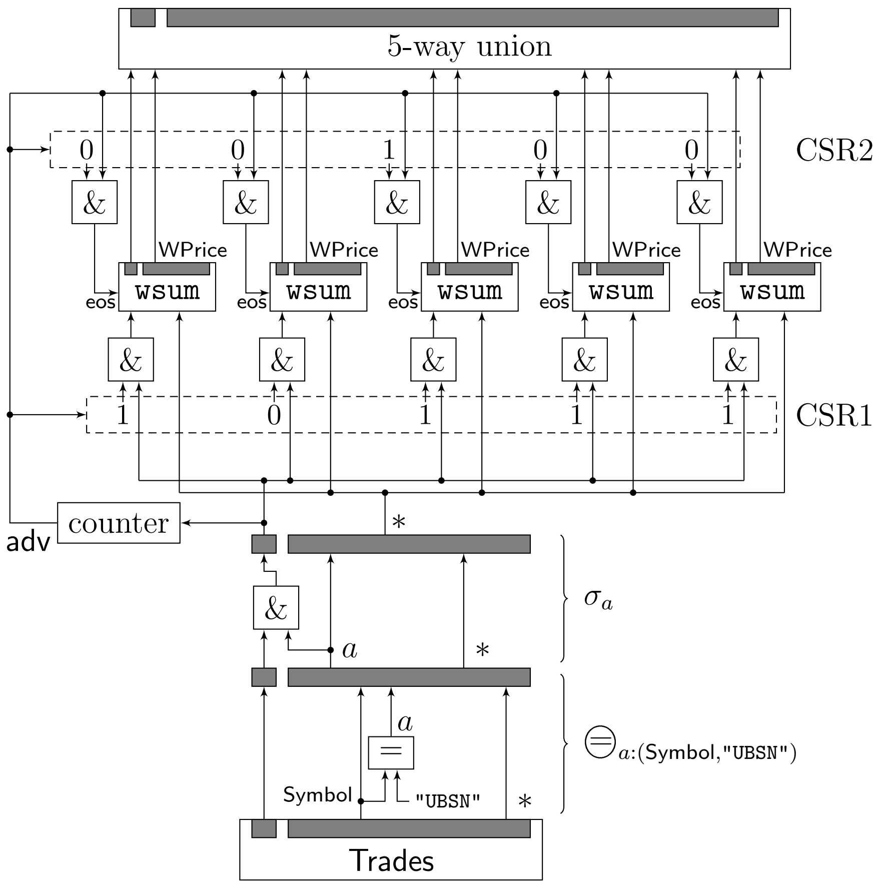


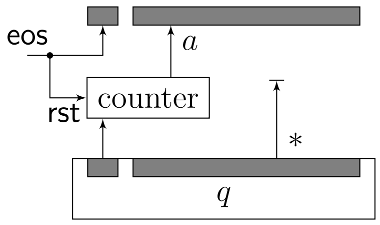
Figure 7.6: Hardware execution plan for Query Q_4

7.3.6 Aggregation

Other than the previous operators, aggregation functions (`count`, `min`, `max`, `avg`, etc.) assume a *finite* input data set. Typically, they are applied on windows. As seen in the previous section, windowing breaks a potentially infinite stream into finite sub-streams. In practice—and as implemented in the previous section—tuples are streamed into a set of open windows immediately after arrival, rather than batching them up until a window closes. The eos signal to notifies the aggregation circuit when a window closes or when the end of the current input stream has been reached (for example when a finite input from a persistent database table has been fully consumed).

Note that the window operator itself does not provide storage for data elements. The tuples are directly forwarded and therefore storage needs to be provided by the implementation of the aggregation function instead. This has the advantage that each aggregation function needs to provide storage just for the amount of state it requires, rather than maintaining the entire window. Following [GBLP96], we classify aggregation functions as follows:

Algebraic Aggregate Functions. We implement algebraic aggregate functions (i.e., ones that use a fixed amount of state) [GBLP96] in a straightforward fashion. To implement `count`, *e.g.*, we use a standard counter component and wire its trigger input to the *data valid* signal of the input stream. Once we reach the end of the current stream, we *(a)* emit the counter value to the upstream data path and *(b)* reset the counter to zero to prepare for the next input stream. In the translation rule for $\texttt{count}_a(q)$,

$\texttt{count}_a(q) \quad \Rrightarrow$ 

, (COUNT)

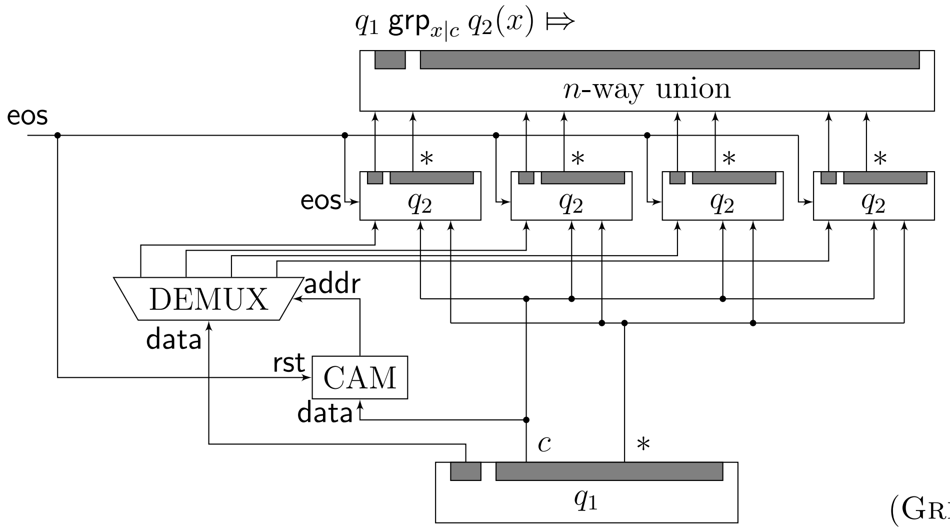
we forward the `eos` signal to the *data valid* output register to implement *(a)* and feed the same signal into the reset input of the counter to implement *(b)*. Note that $\texttt{count}_a$ constructs a new output field without reading any particular input value. The operator emits no other field but the aggregate (we handle grouping separately, see next). For the algebraic aggregates we consider, `count`, `sum`, `avg`, `min`, and `max`, the latency is one cycle. A tuple can be applied at the input every clock cycle (the issue rate is 1).

Holistic Aggregate Functions. For some aggregate functions, the state required is not within constant bounds. They need to batch (parts of) their input until the aggregate can be computed when the end of the stream is seen. The prototype example for such operators are the computation of medians or most frequent items. Our weighted sum operator `wsum` behaves similarly, but needs to remember only the last four input tuples. The use of *flip-flops* is a good choice to hold such small quantities of data. Here we can use them in a *shift register* mode, such that the operator buffer always contains the last four input values.

7.3.7 Grouping

Semantically, a grouping expression $q_1 \; \mathsf{grp}_{x|c} \; q_2$ evaluates the left-hand sub-plan q_1, then routes each tuple to one of a number of independent evaluations of the sub-plan $q_2(x)$. The grouping column c thereby determines the target sub-plan for every input tuple.

FPGA circuits provide excellent support for such functionality. In Section 5.3, we discussed *content-addressable memory* as an efficient mechanism to implement



(GrpBy)

Figure 7.7: Compilation rule to implement the *group by* operator grp

key-value stores. Here, we use that functionality to identify the matching group for an input tuple. Our CAM returns the index i of the sub-plan that matches the given input tuple. We feed this index into the address port of a *de-multiplexer*, which will then route the signal on the data input to the ith output line.

Once again, the *data valid* flag comes in handy here. Rather than explicitly "switching" the entire tuple to the proper sub-plan instance, we use the demultiplexer only to control the *data valid* flag. The actual payload is sent to all sub-plan instances in parallel.

Following our earlier assumptions, we preallocate a number of sub-plan instances, depending on the number of groups that are going to result at runtime. Typically, the sub-plan is a simple aggregate operation with low complexity. Overestimating the number of groups at compile time thus rarely causes a noticeable effect on the overall chip space consumption.

Grouping is typically used in combination with aggregation. Although grouping by itself does not chop an infinite stream into finite subsets, we explicitly indicate the necessary routing of eos signals to the sub-plan instances. In addition, we use the signal to clear the content-addressable memory after each group (rst input).

Our CAM implementation is based on lookup tables with very fast lookup performance. De-multiplexing can be processed using combinational logic, such that the entire routing circuit can typically be processed within a single clock cycle or two (high-capacity CAMs and high-fanout de-multiplexers may be more complex and require an additional wait cycle). As discussed in Section 5.3, LUT-based CAMs have a slow write performance, which we have to pay for whenever a group item is seen the first time. Since this makes the issue rate of the circuit data-dependent, we use a FIFO (not shown in the circuit) to buffer all input. The circuit thus has a variable latency. A hit or a miss can be determined with a latency of one cycle. If no entry is found in the CAM, additional 16 wait cycles are necessary to insert a new entry. Thus, the overall performance of a CAM is one cycle on a hit and 17 cycles for a miss. The latency at the output side is given by the latency of the sub-plan plus one (for the n-way union). The average issue rate is one if we assume that the FIFO is large enough (i.e., at least 16 times the number of groups) to buffer the incoming tuples during the wait cycles when writing to the CAM.

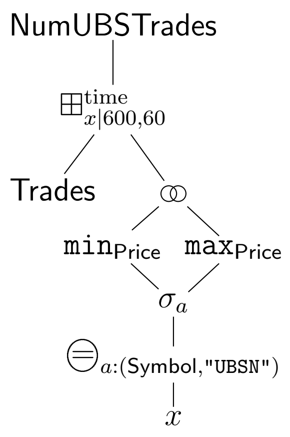
Example. Compiling Query Q_5 would yield a circuit like the one in Figure 7.7, wrapped into a windowing circuit (as in Figure 7.5). We omit the plan here because of its obvious complexity. In the actual application, the Trades stream contains market data of a subset of the stock indexes. With less than a hundred different stock symbols per stream, we can easily replicate the `avg` sub-circuit as demanded by Compilation Rule GrpBy. □

7.3.8 Concatenation Operator

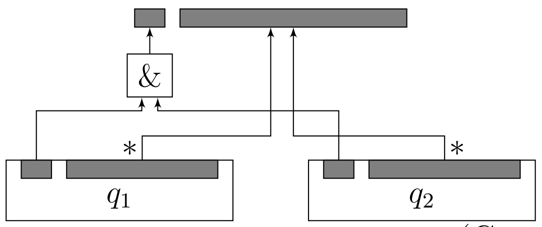
The tuple concatenation operator $\circledcirc$ is a device mainly intended to express multiple aggregates within the same `SELECT` clause. The query

```
SELECT min(Price),max(Price)
  FROM Trades [SIZE 600 ADVANCE 60 TIME]  ,
 WHERE Symbol="UBSN"
```

for instance, could be expressed using the query plan shown here following column on the right. On the hardware side, the semantics of $q_1 \circledcirc q_2$ is straightforward to implement. We simply direct the signals from all input fields to a common output register set. A tuple generated this way only is meaningful if both input tuples were valid. Hence, we use a logical 'and' gate to combine them:


NumUBSTrades
$\boxplus^{\text{time}}_{x|600,60}$
Trades
$\circledcirc$
$\mathtt{min}_{\mathsf{Price}}$
$\mathtt{max}_{\mathsf{Price}}$
σ_a
$\ominus_{a:(\mathsf{Symbol},\texttt{"UBSN"})}$
x

$q_1 \circledcirc q_2 \mapsto$



.

(Concat)

Again, the *and* gate easily finishes within a single cycle. Hence, latency and issue rate are both 1.

7.4 Auxiliary Components

While the previous section provided a compositional scheme to translate a query body into a hardware circuit, actually running the circuit requires some glue logic that lets the execution plan communicate with its environment. *Glacier* includes such logic for commonly used setups.

7.4.1 Network Adapter

In a *commodity* computing system, the communication between a network interface card (NIC) and its host CPU is performed using a multi-step protocol. In a nutshell, the network card transfers a received packet into the main memory of the host system using DMA, then informs the CPU about the arrival by raising an interrupt. The interrupt lets the operating system switch into kernel mode, where the operating system does all necessary packet decoding, before it hands the data off into user space where the payload can finally be processed.

For latency-critical applications (such as algorithmic trading) or ones with high data volumes, such a long processing stack may be prohibitive. Therefore, we decided to implement our own *network adapter* on the FPGA as a soft-core (see Section 5.4). The soft-core directly connects to the Ethernet MAC component of the physical network interface. From there, we grab raw Ethernet network frames immediately when they arrive. We implemented a small UDP/IP stack in the softcore. This allows us to receive UDP datagrams without the help of the CPU. From the decoded UDP datagrams we can extract the data tuples and feed them to the circuit that represents the compiled execution plans. The host CPU only gets involved for the data that remains after the end of the query pipeline, where it is typically faced with a significantly reduced data load due to filtering and aggregation. In Section 7.6, we will see how this enables us to process data at gigabit Ethernet wire speed.

Likewise, we could use the same functionality to build a *data sink* that transmits result data over the network without any involvement of the host CPU.

7.4.2 CPU Adapter

Our system setup in Section 7.2.1 assumes the host CPU as the other end of the processing pipeline. To send (result) data to the CPU, we use a strategy that is similar to the one used by network cards, as sketched above. We write all data into a FIFO that is accessible by the host CPU via a memory-mapped register. Whenever we have prepared new data, we raise an interrupt to inform the CPU. Code in the host's *interrupt service routine* then reads out the FIFO and hands the data over to the user program.

Two different approaches are conceivable to implement a communication in the other direction, i.e., from the CPU to the FPGA. Memory-mapped *slave registers* allow the CPU to push data directly into an FPGA circuit by writing the information into a special virtual memory location. While this provides intuitive and low-latency access to the FPGA engine, the necessary synchronization protocols incur sufficient overhead to fall behind a *DMA-based* implementation if data volumes become high. In this case, the data is written into (external) memory, where logic on the FPGA picks it up autonomously after it has received a *work request* from the host CPU.

7.4.3 Stream De-Multiplexing

Actual implementations may depend on specialized functionality that would be inefficient to express using standard algebra components. In our use case, algorithmic trading, input data is received as a *multiplexed* stream, encoded in a compressed variant of the FIX protocol [FIX09]. Expressed using the StreamBase syntax, the multiplex stream contains actual streams like

```
CREATE INPUT STREAM NewOrderStream (
  MsgType      byte, -- 68: new order
  ClOrdId      int,  -- unique order identifier
  OrdType      char, -- 1:market,2:limit,3:stop
  Side         char, -- 1:buy,2:sell,3:buy minus
  TransactTime long) -- UTC Timestamp

CREATE INPUT STREAM OrderCancelRequestStream (
  MsgType      byte, -- 70: order cancel request
  ClOrdId      int,  -- unique order identifier
  OrigClOrdId  int,  -- previous order
  Side         char, -- 1:buy,2:sell,3:buy minus
  TransactTime long) -- UTC Timestamp
```

We have implemented a *stream de-multiplexer* component as part of the *Glacier* library. The de-multiplexer interprets the `MsgType` field (first field in every stream) and dispatches the tuple to the proper plan part.

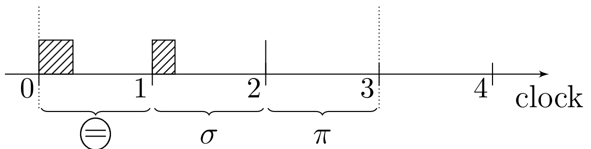
7.5 Optimization Heuristics

In Section 7.3 we focused on providing a complete and fully compositional set of compilation rules. With these rules arbitrary stream queries can be compiled into a logic circuit. It is not surprising that “hand crafting” a specific plan sometimes may lead to plans with lower latency and/or better issue rate. It turns out that rather simple optimization heuristics already suffice to make the output of our compiler close to hand-optimized plans.

7.5.1 Reducing Clock Synchronization

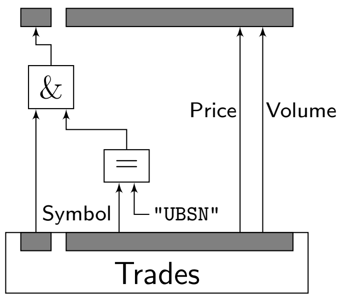
Our compilation rules assume strict *synchronization* of every operator implementation. Every operator is expected to have its result ready after an integer number of clock cycles (the operator's latency). Even though simple computations could finish in less time than a full cycle, their result is always buffered in a flip-flop register, where it waits until the end of the clock cycle.

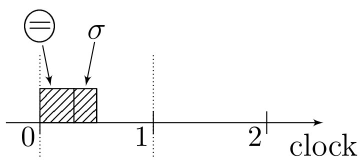
Example. Consider again the compiled circuit for Query Q_1 (Figure 7.4). As discussed earlier, this circuit requires three clock cycles to execute. Little of that time is used for actual processing, however. In the following timing diagram, we illustrate when each of the three plan parts perform actual processing (indicated as ▨):



Equality comparison takes slightly longer to evaluate than the logical 'and' (which is what σ essentially does). There is no actual work to be done for projection at all, still all three plan parts occupy a full clock cycle each. □

If no component inside a plan step is inherently clock bound (such as access to clocked memory components), a plan optimizer can trivially eliminate intermediate registers and run (part of) a sub-plan using combinational logic only. Applying this idea to the plan for Query Q_1 results in the plan we use in our actual implementation:

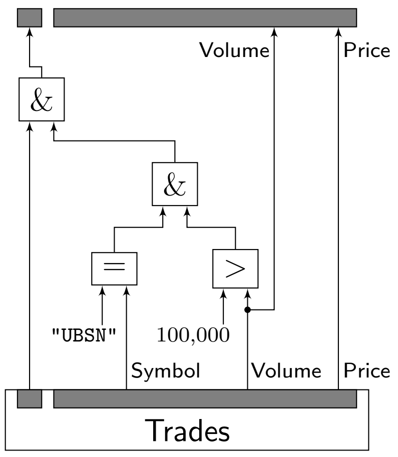


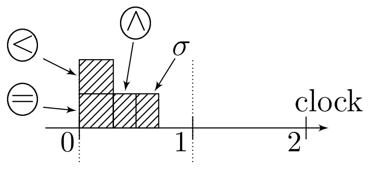


As shown in the timing diagram on the right, this sub-plan now runs both processing steps directly in succession and finishes within a single clock cycle. The most apparent effect of this optimization is the reduction of latency. The plan for Query Q_1 has now a latency of one. In addition, we saved a small amount of FPGA resources, primarily flip-flops that were needed for buffering before.

7.5.2 Increasing Parallelism

The elimination of intermediate registers often automatically leads to *task parallelism*. With registers removed, the hardware circuit for Query Q_2 looks as follows:





In this circuit, the two value comparisons run truly in parallel (whereas they would execute sequentially in the non-optimized plan). In the timing diagram on the right, one can see how we packed additional work into the same clock cycle. In effect, Query Q_2 executes in a single cycle, too.

7.5.3 Trading Unions For Multiplexers

When translating the windowing operator ⊞ (Rule Wind), we used the *union* circuit of Section 7.3.4 as a convenient tool to merge all window outputs into a single result stream. Except in exotic cases, only one of these outputs is actually producing data at any point in time, and we know which one.

We can take advantage of this knowledge by replacing the union circuit with a *multiplexer* component in such cases. As the name suggests, a multiplexer is the counterpart to the de-multiplexer we saw in Section 7.3.7. Provided an index i, it routes the signal at the ith input to its output port. In windowing circuits, we know the index of the data-producing sub-plan from the shift register CSR2. Using a multiplexer, we can now feed the output of this sub-plan directly into the output register of the windowing circuit.

As discussed in Section 7.3.4, the hardware circuit for union uses FIFO queues at each of its n inputs. By using a multiplexer instead, we can free the resources occupied by the n FIFOs. Depending on the FIFO implementation chosen, this may free mostly flip-flop registers or block RAMs, plus the necessary logic (LUTs) that drives the FIFOs. In addition, we save one clock cycle of latency that was originally spent in the input FIFOs. Applied to the plan in Figure 7.6, this reduces the latency from 5 to 4 (eliminating registers on the bottom half of the plan saves another two cycles of latency).

7.5.4 Group By/Windowing Unnesting

The $\boxplus$-grp pattern shown in the algebraic query plan for Query Q_5 (see Figure 7.1(e)) is a common combination in stream processing. A straightforward application of the WIND and GRPBY rules to this pattern will replicate the *group by* circuit for each of the n sub-plan instances in Rule WIND. The resulting query execution plan will thus use n de-multiplexers and content-addressable memories and route tuples independently for each group.

Typically, all windows will contain roughly the same groups, and all CAM instances will contain roughly the same data items. It therefore makes sense to “pull out” the individual DEMUX/CAM pairs of the replicated sub-plans and use a global instance of each instead. In a sense, we swap the roles of $\boxplus$ and grp in the algebraic plan.

The primary effect of “unnesting” the tuple dispatching functionality of the *group by* operation is a considerable resource saving. The penalty we pay is a slight increase in the number of groups, since the union of all groups in individual windows is now held in a single CAM.

7.6 Evaluation

Compiling stream queries into logic circuits is only meaningful if the resulting circuits solve the problems that we motivated in Section 7.1. This evaluation section thus focuses on the relevant performance metrics *latency* and *throughput* (our subject for Section 7.6.1). In Section 7.6.2, we verify that the integration of an FPGA into the data path of a streaming engine leads to an actual improvement in *end-to-end performance*.

7.6.1 Latency and Throughput

Other than in software-based setups, the performance characteristics of hardware execution plans can accurately be derived by solely analyzing the circuit design. Thereby, the performance of a larger circuit is determined by the performance of its sub-plans. In the following, we first concentrate on latency, then investigate throughput.

	Latency		**Issue Rate**	
Query	non-opt.	opt.	non-opt.	opt.
Q_1	3	1	1	1
Q_2	5	1	1	1
Q_3	5	2	1	1
Q_4	5	2	1	1
Q_5	$6 \ldots 6 + 16G$	$5 \ldots 5 + 16G$	1	1

Table 7.2: Latencies and issue rates for optimized query plans of Q_1–Q_5

Latency

We measure the latency of a hardware circuit in the number of clock cycles that occur from the time a tuple enters the circuit until the time a result item is produced. For the case of *group by* queries, the relevant input tuple is the last tuple of the input stream. Our FPGA is clocked at a rate of 100 MHz. Each latency cycle thus implies an observable latency of 10 nanoseconds.

In a sequential data flow, the latencies of all sub-plans behave cumulatively: the overall latency of the full plan can be obtained by summing up the latencies of all sub-plans. Parallel circuits (such as the sub-plan instances in a *group by* plan) are determined by the latency of the slowest sub-plan. Without applying any of the optimization techniques of Section 7.5, this yields the latencies reported in Table 7.2 as "non-opt." (we will discuss the details of Query Q_5 in a moment).

Non-Optimized Circuits. For the simple circuits (Queries Q_1 and Q_2), the total latency corresponds to the number of flip-flop registers along the data path. For Queries Q_3 and Q_4, the union operators at the top of the plan add another latency cycle due to their built-in FIFOs (Section 7.3.4).

In Query Q_5, the difference in read and write speed of our content-addressable memory introduces a data dependence of the circuit latency. Thus, Table 7.2 reports lower and upper bounds for the latency at runtime. Once the circuit has seen all possible group identifiers (and thus has filled its CAM), no write access occurs and the circuit responds after six cycles. By contrast, if G different new groups arrive in succession, their group identifiers queue up in the input FIFO of the *group by* circuit and each one adds 16 cycles for the CAM write.

After Optimization. The optimizations we described in Section 7.5 reduced latency by eliminating intermediate flip-flop registers. As listed in Table 7.2, this reduces latency down to one or two clock cycles for Queries Q_1–Q_4. The use of a multiplexer as described in Section 7.5.3 saves one latency cycle for Query Q_5.

Observations. Table 7.2 reports single-digit latencies for most queries. The latency of Query Q_5 clearly tends toward the optimum case in practice, since the arrival of a large number of new groups right before the end of a window is rare. With a cycle time of 10 ns, our FPGA typically responds in less than a micro-second.

Throughput

The maximum throughput of a circuit is directly dependent on its issue rate. With a 100 MHz clock, an issue rate of 1 means that the circuit can process 100 million input tuples per second.

All our plans are fully pipelineable. As can be seen in Table 7.2, this leads to an issue rate of 1 for all five example queries. In the upcoming section, we are going to demonstrate how this enables us to process very high data rates at wire speed in a network-attached configuration (shown in (Figure 7.2(a)).

7.6.2 End-To-End Performance

A key aspect of using an FPGA for data stream processing is that the hardware circuit can directly be hooked into an existing data path. As already sketched in Section 7.2.1, we are particularly interested in using the FPGA as a preprocessor that operates between the physical network interface and a general-purpose CPU (though the idea could be applied to other data sources, too). To verify the effectiveness of this setup, we implemented it using an FPGA development board, then measured the data rates it can sustain.

The biggest challenge in commodity systems is to process network data with high *package rates* (as opposed to large-sized packages). Actual application setups in software start suffering at data rates of $\gtrsim 100,000$ packets/s because of the high intra-host communication overhead for every packet. By contrast, our query execution circuit is directly connected to the physical network interface. The experiments in the following show how this enables us to process significantly higher package rates at wire speed.

Our experiments are based on a Xilinx XUPV5 development board that ships with the FPGA mentioned in Section 7.2 and includes a fast 1 GBit Ethernet interface (described in Section 5.2.3 on page 156). We implemented the system configuration shown in Figure 7.2(a) as an embedded system by instantiating the necessary hardware components as soft-cores inside the FPGA chip. Our CPU in this setup is a Xilinx MicroBlaze CPU.

It turns out that it is fairly difficult to generate really high package rates in a lab setting. With a NetBSD-based packet generator, we managed to generate a maximum of 1,000,400 packets/s (all UDP traffic). Still, this was not sufficient to

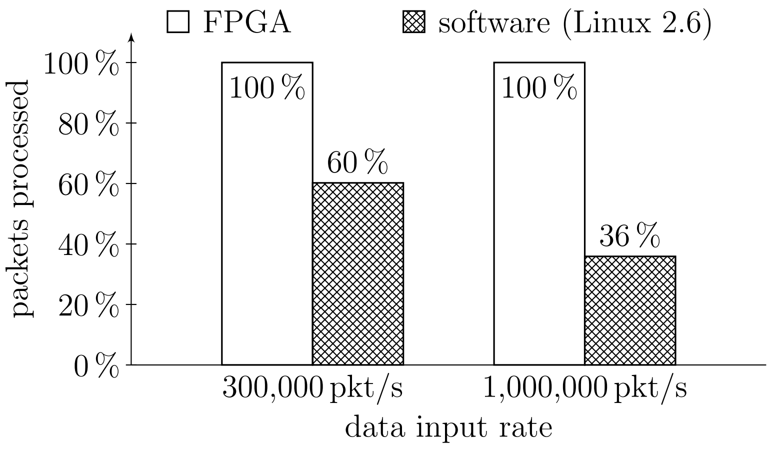


Figure 7.8: Number of packages successfully processed for two given input loads. The hardware implementation is able to sustain both package rates.

saturate our hardware implementation. As illustrated in Figure 7.8, no data was lost when processed on the FPGA.

For comparison, we hand-crafted a light-weight network client under Linux (2.6 kernel), designed to accept and process the same input data at high speed. Yet, as shown in Figure 7.8, this client was not able to sustain the load we applied. For high package rates, it dropped more than half of all input tuples.

Our results clearly demonstrate that our circuit can meet the expectations we set. This makes FPGAs particularly attractive for common application scenarios. If the FPGA is used as a rate-reducing component in the data input path, the remainder of the system faces only a fraction of the input load. This significantly increases the applied load that the system can sustain.

7.7 Stream Joins

In this section we describe our implementation of a window-join operator for FPGAs. Even though we present the window join at a higher level, it it has a compatible interface and fits into the Glacier library.

Joins in streaming contexts require windowing. The concept of a window on a stream allows looking at finite subsets of the stream tuples. This has the advantage that aggregates on that window are not blocking despite the potentially unbounded stream. In fact, traditional relational operators can be applied on the window. Figure 7.9 (adopted from [KNV03]) illustrates this for the case of a join operation. The join in the middle is always evaluated only over finite subsets taken from both input streams. Windows over different input data can span different numbers of

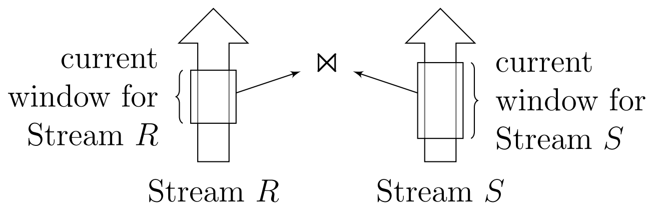


Figure 7.9: Window join (figure adopted from [KNV03])

tuples, as indicated by the window sizes in Figure 7.9.

Various ways have been proposed to define suitable window boundaries depending on application needs. In this work we focus on *sliding windows*, which, at any point in time, cover all tuples from some earlier point in time up to the most recent tuple. Usually, sliding windows are either time-based, i.e., they cover all tuples within the last τ time units, or tuple-based, i.e., they cover the last w tuples in arrival order. Depending on the arrival rate in the stream, time-based windows can have unbounded state requirements. The window join operator we developed for *Glacier* supports tuple-based windows. Time-based windows in traditional streaming systems require sophisticated buffer management techniques and dynamic memory management. Given the fairly limited amount of storage for state on an FPGA (LUTs, BRAM blocks, and flip-flops), on-chip solutions would only work on a small scale. As in traditional systems, external memory, such as DDR memory, has to be used. Unfortunately, this is significantly more complex on an FPGA than on a general-purpose CPU. Therefore, we limit our discussion here to tuple-based windows, which have well defined state requirements. In the following discussion, we always assume tuple-based windows.

7.7.1 Sliding-Window Joins

The exact semantics of window-based joins (precisely which stream tuple could be paired with which?) in existing work was largely based on how the functionality was implemented. For instance, windowing semantics is implicit in the three-step procedure devised by Kang et al. [KNV03]. The procedure is performed for each tuple r that arrives from input stream R:

1. *Scan* stream S's window to find tuples matching r.
2. *Insert* new tuple r into window for stream R.
3. *Invalidate* all expired tuples in stream R's window.

Tuples s that arrive from input stream S are handled symmetrically. Sometimes, a transient access structure is built over both open windows, which accelerates

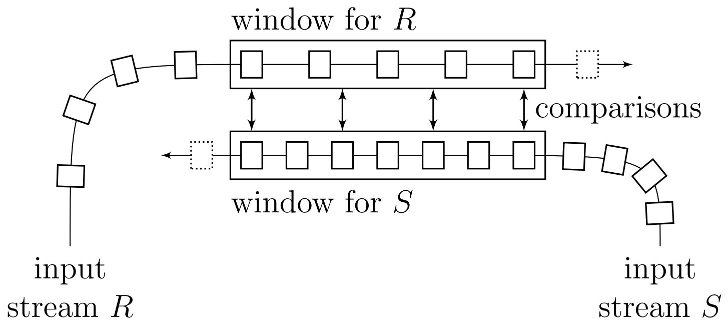


Figure 7.10: Handshake join idea. Streams flow by each other in opposite directions; comparisons (and result generation) happens in parallel as the streams pass by.

Step 1 at the cost of some maintenance effort in Steps 2 and 3. The three-step procedure carries an implicit semantics for window-based stream joins:

Semantics of Window-Based Stream Joins. *For $r \in R$ and $s \in S$, the tuple $\langle r, s\rangle$ appears in the join result $R \bowtie_p S$ iff*

(a) r arrives after s and s is in the current S-window at the moment when r arrives or

(b) r arrives earlier than s and r is still in the R-window when s arrives

and r and s pass the join predicate p.

A problem of the three-step procedure is that it is not well suited to exploit the increasing degree of *parallelism* that FPGAs provide. Furthermore, *local* availability of data, i.e., data next to the computation, inherently seems to contradict the nature of the join problem, where *any* tuple in the opposite window represents a possible match.

7.7.2 Handshake Join

It turns out that we can learn from soccer players here. Soccer players know very well how all pairs of players from two opposing teams can be enumerated without any external coordination. Before the beginning of every game, it is tradition to shake hands with all players from the opposing team. Players do so by *walking by* each other in opposite directions and by *shaking hands* with every player that they encounter. Very naturally, the procedure *avoids bottlenecks* (each person has to shake only one hand at a time), keeps all interaction *local* (fortunately—people's arm lengths are limited), and has a very simple *communication pattern* (players only have to walk one step at a time and there is no risk of collision). All three aspects are desirable also in the design of parallel algorithms.

Stream Joins and Handshaking. The handshake procedure used in sports games inspired the design of *handshake join*, whose idea we illustrated in Figure 7.10. Tuples from the two input streams R and S, marked as rectangular boxes □, are pushed through respective join windows. Upon window entrance, each tuple pushes all existing window content one step to the side, such that always the oldest tuple "falls out" of the window and expires. In software, this process could be modeled with help of a *ring buffer* or a *linked list*; in FPGA-based setups, a *shift register* would serve the same purpose. Both join windows are lined up next to each other in such a way that window contents are pushed through in opposing directions as shown in Figure 7.10.

Whenever two stream tuples $r \in R$ and $s \in S$ encounter each other (in a moment we will discuss what that means and how it can be implemented), they "shake hands", i.e., the join condition is evaluated over r and s, and a result tuple $\langle r, s \rangle$ appended to the join result if the condition is met. Many "handshakes" take place at the same time, work that we will parallelize over available compute resources. To keep matters simple, we assume that only a new item is inserted into only *one* window at a given time instant. An implementation is free to lift this restriction, provided that it properly deals with race conditions.

Semantics. While a stream item $r \in R$ travels along its join window, it will always encounter at least those S-tuples s_i that were already present in S's join window when r entered the arena. Likewise, if r arrived earlier than some S-tuple s, r and s will meet eventually (and thus form a join candidate) whenever r is still in the R-window at the moment when s arrives.

Observe how this semantics *coincides* with the window semantics implied by the three-step procedure of Kang et al. [KNV03]. Thus, handshake join implements the *same* semantics as existing techniques; only the order of tuples in the result stream (and thus also per-tuple latencies) might change.[1] In addition, handshake join *(a)* gives a new *intuition* on what window-based joins mean and *(b)* opens opportunities for effective *parallelization* on modern hardware. Next, we will demonstrate how to exploit the latter.

Parallelization. Figure 7.11 illustrates how handshake join can be parallelized over available compute resources. Each processing unit (or "core") is assigned one *segment* of the two join windows. Tuple data is held in local memory (if applicable on a particular architecture), and all tuple comparisons are performed locally.

This parallel evaluation became possible because we converted the original *control flow* problem (or its procedural three-step description) into a *data flow*

[1]This disorder can easily be corrected with help of punctuations [LMT$^+$05].

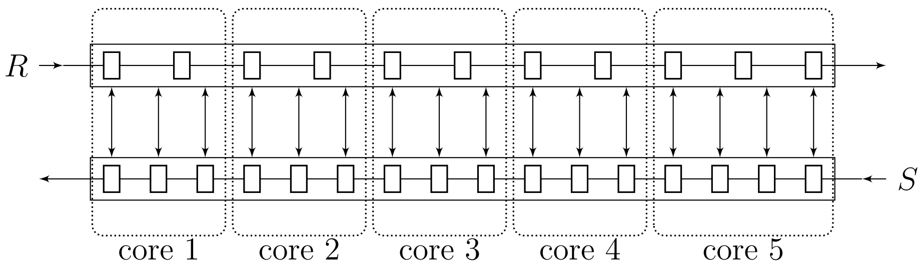


Figure 7.11: Parallelized handshake join evaluation. Each compute core processes one segment of both windows and performs all comparisons locally.

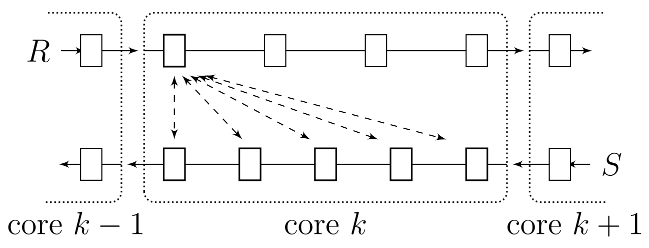
representation. Rather than synchronizing join execution from a centralized coordinator, processing units are now driven by the flow of stream tuples that are passed on directly between neighboring cores. Processing units can observe locally when new data has arrived and can decide autonomously when to pass on tuples to their neighbor. In addition, we have established a particularly simple *communication pattern.* Processing units only interact with their immediate neighbors, which may ease inter-core routing and avoid communication bottlenecks. We can therefore expect a good circuit performance on the FPGA as this communication pattern applies little pressure on the signal routing.

Both properties together, the representation as a data flow problem and the point-to-point communication pattern along a linear chain of processing units, ensure scalability to large numbers of processing units. Additional cores can either be used to support larger window sizes without negative impact on performance, or to reduce the workload per core, which will improve throughput for high-volume data inputs.

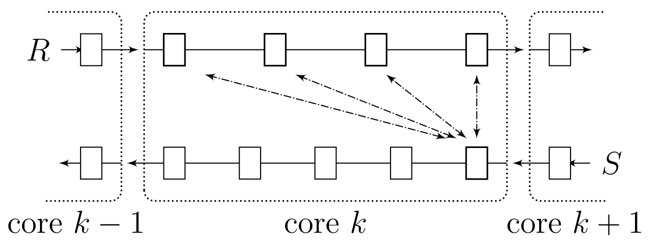
Encountering Tuples. For proper window join semantics, the only assumption we have made so far is that an item that enters either window will encounter all current items in the other window *eventually.* That is, there must not be a situation where two stream items can pass each other without being considered as a candidate pair. Thus, any local processing strategy that prevents this from happening will do to achieve correct overall window semantics.

In Figure 7.12 we illustrate how to process a segment k. In this illustration, we assume that every tuple $r \in R$ is compared to all S-tuples in the segment at the moment when r enters the segment. Figure 7.12(a) shows all tuple comparisons that need to be performed when a new R-tuple is shifted into the segment.

Likewise, when a new tuple $s \in S$ enters the segment, it is immediately compared to all R-tuples that are already in the segment, as illustrated in Figure 7.12(b). This approach will operate correctly regardless of the window size



(a) Tuple from stream R entered segment.



(b) Tuple from stream S entered segment.

Figure 7.12: Eager scan strategy. A tuple entering from stream R or S will trigger comparisons $\swarrow\!\!\nearrow$ or $\nwarrow\!\!\searrow$ in the segment of processing core k (respectively).

configuration. Similarly, segmentation can be chosen arbitrarily, which might be useful for very particular data characteristics or in systems with a heterogeneous set of processing cores.

7.7.3 Handshake Join Implementation on FPGAs

Figure 7.13 illustrates the high-level view of our handshake join implementation on an FPGA. The windows of the R and S streams are partitioned among n cores. The cores are driven by a common clock signal that is distributed over the entire chip. The synchronous operation of the cores avoids any buffering (such as FIFO) between the cores and, thus, reduces the complexity of the implementation. The tuples move in lock-step through the window. The windows represent large shift registers, which can be efficiently implemented in hardware.

For each core we need to provide a hardware implementation of the segment for the R and S windows, a digital circuit for the join-predicate, and scheduling logic for the tuples and the window partitions. The figure shows the two 64 bit-wide shift registers (labeled 'R window' and 'S window', respectively) that hold the $\langle k, v_R \rangle$ and $\langle k, v_S \rangle$ tuples. In this discussion we assume 32-bit wide keys k and values v_R and v_S. When a new tuple is received from either stream, the tuple is inserted in the respective shift register and the key is compared against all keys in the opposite window. Here, we use a simple nested-loop join, i.e., the elements

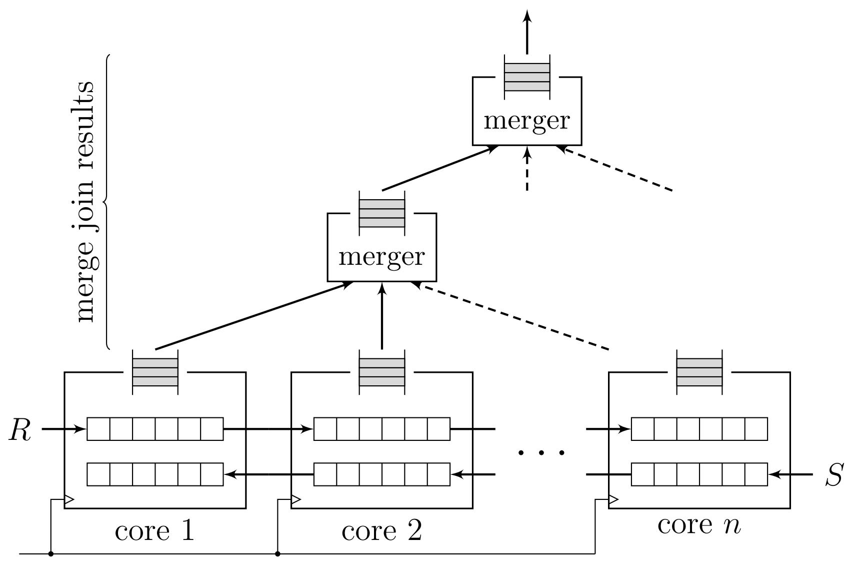


Figure 7.13: FPGA Implementation of Handshake Join for Tuple-based Windows

of the window are compared sequentially. This is done by a simple sequencer that implements the tuple scheduling logic.

Assuming two tuple-based windows with sizes W_R and W_S, the comparison requires $\max\left(\lceil W_R/n \rceil, \lceil W_S/n \rceil\right)$ clock cycles. The number of clock cycles can be reduced at the cost of increased circuit complexity by instantiating multiple predicate evaluation sub-circuits, thereby allowing for a parallel execution of the predicates. As illustrated in the top half of Figure 7.13, each join core will send its share of the join result into a FIFO queue (indicated as ▤). A merging network will merge all sub-results in to the final join output at the top of Figure 7.13.

Join Core. As depicted in Figure 7.14, the stream data for R and S is directly fed into and out of the cores. An additional “valid” line is used to represent whether the data lines contain a valid tuple. Our current implementation allows processing one tuple per clock cycle. The “enable” signals specify whether in any given clock cycle a new tuple is shifted along the R or S stream. These two signals are asserted by a simple admission control circuit when the data streams enter the chip. Every join core can raise a “throttle” signal when its FIFO is close to become full. The admission controller can use this information to either (1) discard new tuples when they enter the chip or (2) drop tuples waiting in the output FIFO. As such, the throttle signal allows us to handle overload situations in a controlled manner.

The signal is asserted before the FIFOs are actually full such that enough free slots are available for all join tuples that could be generated from the current input

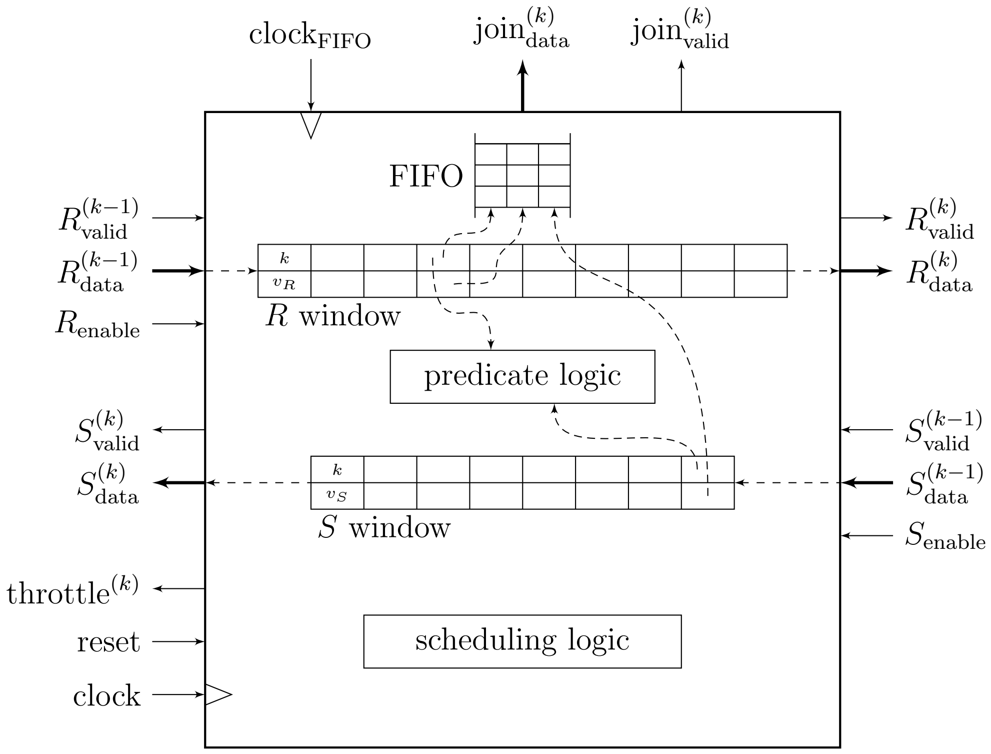


Figure 7.14: Join Core Implementation on FPGAs

tuple. In other words, we assert the throttle signal if less than $\max\left(\lceil W_R/n \rceil, \lceil W_S/n \rceil\right)$ entries are free in the output FIFO of a core.

Merging Network. Joins can potentially generate a large amount of result tuples. It is therefore crucial to siphon off the result data from the join cores to avoid overflows in the local output FIFOs. The output bandwidth is determined by the speed of the top-most merger FIFO, which can accept and deliver one tuple per clock. The merging network is driven by a different clock than the shift-register and the tuple scheduling logic. This allows us to balance throughput and latency depending on the join hit-rate. For example, a high clock frequency in the merging network and a comparatively small shift-clock can be used for joins that are known to yield a high hit-rate. In the scenario of a low hit-rate, a small data volume is generated and the shift-circuit can be operated at higher speeds.

A merger element is essentially a union operator as described in Section 7.3.4. The merger elements in the merging network each consists of a FIFO element and control logic that reads from a number of inputs FIFOs, i.e., the child mergers. We vary the fan-in of the mergers in the range of $2, \ldots, 8$ elements. The mergers can only accept one tuple per clock. Thus, if more than one FIFO has data

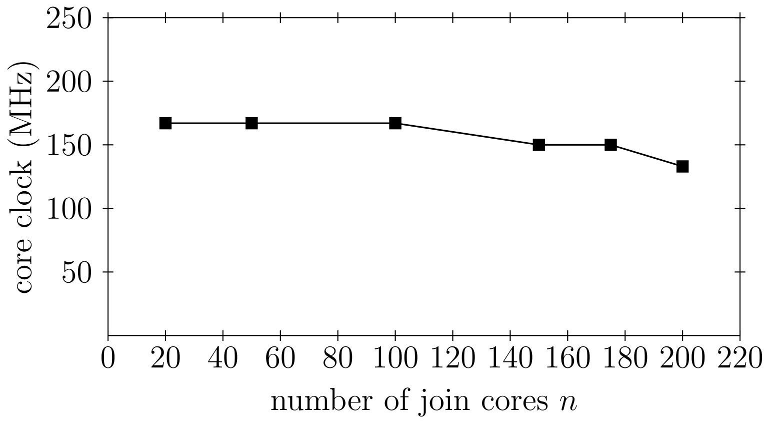


Figure 7.15: Scalability of FPGA Handshake join with a constant segment size of 8 tuples per window and core

available, access needs to be controlled such that starvation is avoided. As for the union operator a *round-robin token passing arbiter* [SMR02] is used to guarantees fairness.

Our Virtex-6 chip provides FIFOs as 36 kbit block RAM (BRAM) elements. They can be configured to a maximum width of 36 bits and a depth of 1024 elements. Therefore, in order to store 96-bit result tuples three 36 kbit BRAM are required. The XC6VXL760 chip has capacity for 240 96-bit result FIFOs (either as output FIFOs of the cores or in the mergers). In our design they represent the limiting factor that determines the maximum number of cores we can instantiate.

7.7.4 Evaluation of Handshake Join Operator

FPGAs provide a very direct measure of the scalability that an algorithm can provide. In a scalable design, the *maximum clock frequency* at which a circuit can be operated is independent of the configured size of the circuit. For algorithms that scale less favorably, the clock frequency will have to be turned down as the configuration size increases. This behavior could also be observed for sorting networks in Figure 6.17 (page 190).

The maximum clock frequency that can be achieved for our circuit is shown in Figure 7.15 for different core numbers. For this experiment we scaled up the number of cores n, but left the per-core window size constant at eight tuples per core (for $n = 100$, e.g., the overall window size will be $100 \times 8 = 800$ tuples per stream). As can be seen in the figure, clock frequencies remain largely unaffected by the core count, which confirms the scalability of handshake join.

On the right end of Figure 7.15, our design occupies more than 96 % of all

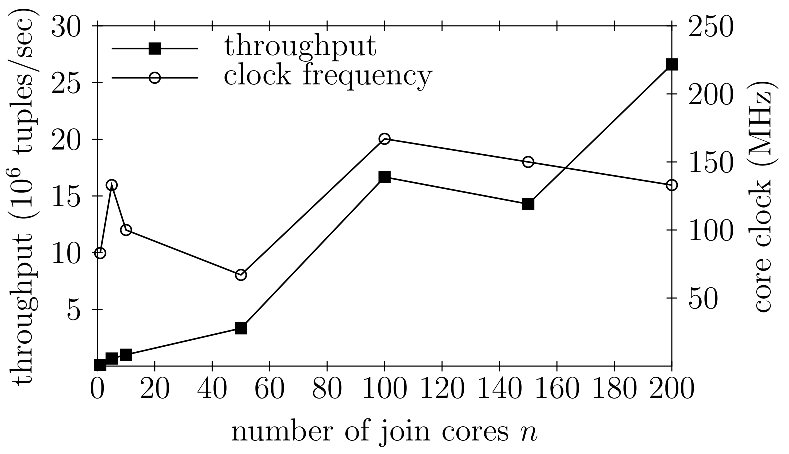


Figure 7.16: Throughput and clock frequency for two windows of size 1,000 implemented using different numbers of cores

available FPGA BRAM resources. As such, we are operating our chip far beyond its maximum recommended resource consumption (70–80 %) [DeH99]. The fact that our circuit can still sustain high frequencies is another indication for good scalability. Earlier work by Qian et al. [QXD+05] on FPGA-based stream join processing suffered a significant drop in clock speeds for the window sizes we consider, even though their system operated over only 4 bit-wide tuples.

7.7.5 Optimal Number of Cores

In order to obtain a high throughput for given window sizes W_R and W_S the number of join cores n needs the maximized and thereby reducing the size window partition per core. The throughput can be computed as

$$\text{throughput} = \frac{1}{\max\left(\lceil W_R/n \rceil, \lceil W_S/n \rceil\right)} \cdot f_{\text{clk}}$$

(the denominator is the maximum number of clock cycles needed to scan the local join windows.

The remaining parameter that determines the overall achievable throughput is the clock frequency f_{clk} of the join circuit. In VLSI the highest possible clock frequency is determined by complexity of the circuit, i.e., the number of components and density of circuit connections. It is not clear how the clock frequency depends on the core number $\leftrightarrow$ window size trade-off. Few cores have comparatively large window partitions and therefore a higher complexity inside the cores. When a large number of cores is instantiated the complexity of each core is low, while the wiring complexity between the cores is higher.

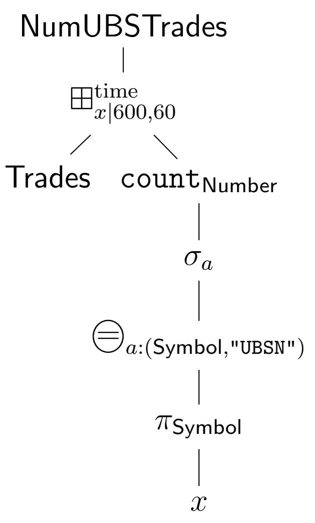
In order to measure this impact, we choose two fixed-sized global windows $W_R = W_S = 1,000$ elements. We vary the number of cores between 1–200 cores partition the global windows evenly over the cores. We then determine the highest possible clock frequency of the join circuit. For this analysis we choose a moderate timing for the result circuit of 83 MHz, resulting in an output tuple rate of 83 M tuples/sec. We synthesize each circuit for the Virtex-6 XC6VLX760 FPGA chip.

Figure 7.16 shows the clock frequency obtained and the resulting tuple throughput per input stream. The highest clock frequency is reached for $n = 100$ cores and 10 tuples per window and core. At this point the complexity of a core and the interconnect are in balance. When increasing the number of cores the clock frequency decreases. For 150 cores the decrease in clock frequency cannot be compensated by the speed-up of the additional processing cores, resulting in a lower throughput number than for $n = 100$ cores.

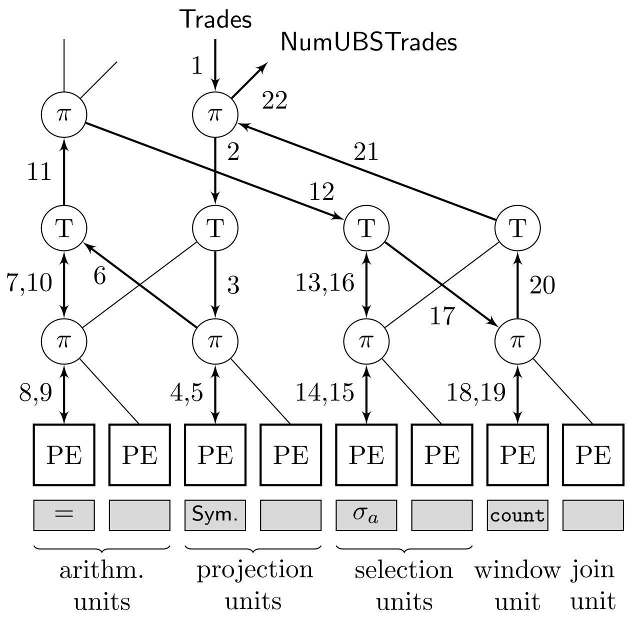
7.8 High-level Architectures

In the Glacier approach the translation of queries into circuits is done at the bit and wire level. As discussed, this results in high throughput and very efficient designs due to the global optimization in the synthesis and placement FPGA tools. On the flip side, however, this translation step can be time consuming. First of all, the Glacier compiler and the entire FPGA design flow have to be repeated for each new query. Second, the FPGA design flow itself can require a significant amount of time, for example, the map and place-and-route stage can require minutes to several hours processing time.

In this section, alternative approaches are outlined that operate on a coarse-grained level. Instead of generating full-custom designs for each circuit, a fixed architecture is used for all queries. The architecture is only configured through runtime parameters or micro-programs depending on the queries at hand. To this end, the Glacier query circuits are replaced by set of processing elements (PEs) and an on-chip network, a such called *network on chip* (NoC) [BM02]. The NoC represents an *overlay network on top of the physical interconnect* of the FPGA. This allows the implementation of a data flow-oriented processing model similar to systolic arrays [KL80] just by configuration of the processing elements and the overlay network. At the very extreme end, modern *general purpose computing on graphics processor units* (GPGPUs) can be considered as an extension of this idea. In the following two sections the advantages and disadvantages of such high-level approaches are discussed and compared to Glacier.



(a) Query Plan



(b) Parameterizable processing elements (PE) with Butterfly Fat Tree overlay network.

Figure 7.17: Query plan (a) mapped onto a fixed architecture consisting of a set of parameterizable processing elements and an overlay network (b). The data flow in the overlay network is indicated by the labeled edges 1–22.

7.8.1 Overlay Architectures

An obvious approach is to map each operator onto a processing element and then build up data flow graph by configuration of the overlay network. This strategy is illustrated in Figure 7.17. The query plan for the simple temporal aggregation query show in Figure 7.17(a) is mapped onto a set of processing elements depicted in Figure 7.17(b). For now, we assume that the on-chip system contains eight fixed function processing elements, in particular, two arithmetic, two projection and selection units, one window, and one join unit. These units are implemented similar to Glacier components but in a generic way. The function the perform can be be selected by parameterization, e.g., by writing to a set of configuration registers at runtime.

The PEs are connected over a *Butterfly Fat Tree* network. The network is implemented using T and π switching elements. [2] The network is implemented using FPGA primitives and explicitly placed on the chip. Kapre et al. [KMd+06]

[2]Note that π in this context refers to the switching elements not to the relational projection operator.

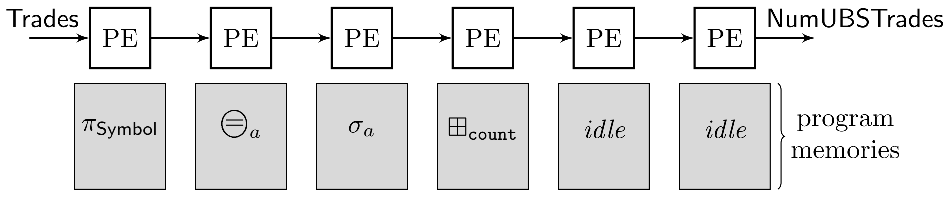


Figure 7.18: Mapping of the query plan in Figure 7.17(a) onto a fixed architecture consisting of programmable processing elements (PE) arranged in a chain overlay.

to implement the network using packet-switching or time-multiplexing of the links. Assume in this discussion that all links bidirectional and have a fixed word width, e.g., 32 bit. The fixed function processing elements and the overlay network are synthesized and placed onto the FPGA.

At runtime the operators corresponding to the query in Figure 7.17(a) are mapped onto the PEs as shown in Figure 7.17(b). The operation performed by the PEs is set by the configuration written into the shaded registers. The processing elements are connected by using links of the overlay network are used as indicated by the labeled edges 1–22. Since this "virtual" configuration in contrast to a "physical" configuration does not involve any FPGA tools it is very lightweight and thus very fast. The disadvantage is this solution, however, is the lower performance to the fixed width overlay links compared to the wide parallel links used in Glacier. Furthermore, the circuit optimization techniques described in Section 7.5 cannot be applied, as they cannot be used on "virtual" configurations.

Another disadvantage arises from the limited number of fixed function PEs. For example if a query contains more than two arithmetic operators $\circledast$ it cannot be executed on the architecture shown in Figure 7.17 (unless PEs are virtualized). As a solution the fixed function PEs can be replaced by programmable processing elements. They can be designed as application-specific processors. Instead of being parameterizable they are fully programmable, i.e., the operators they implement are expressed as short program sequences. Since every PE can therefore implement all operators the overlay network can be significantly simplified. Figure 7.18 shows the fully programmable PEs connected in a chain overlay. Each PE contains a private program memory, which contains a micro-code sequence that implements the operator. Note that in this example note all PEs in the data flow pipeline are used. The last two units are idle are assumed to be configured to pass through the data. By the use of programmable PEs further runtime flexibility can be gained, however, at the cost of a lower performance. A micro-coded operator is typically less efficient that an implementation in hardware. This type of data flow processing can also be performed on the Cell Broadband Engine [GHF$^+$06] and massively-parallel processor arrays (MPPAs) such as the Ambric Am2045 [But07].

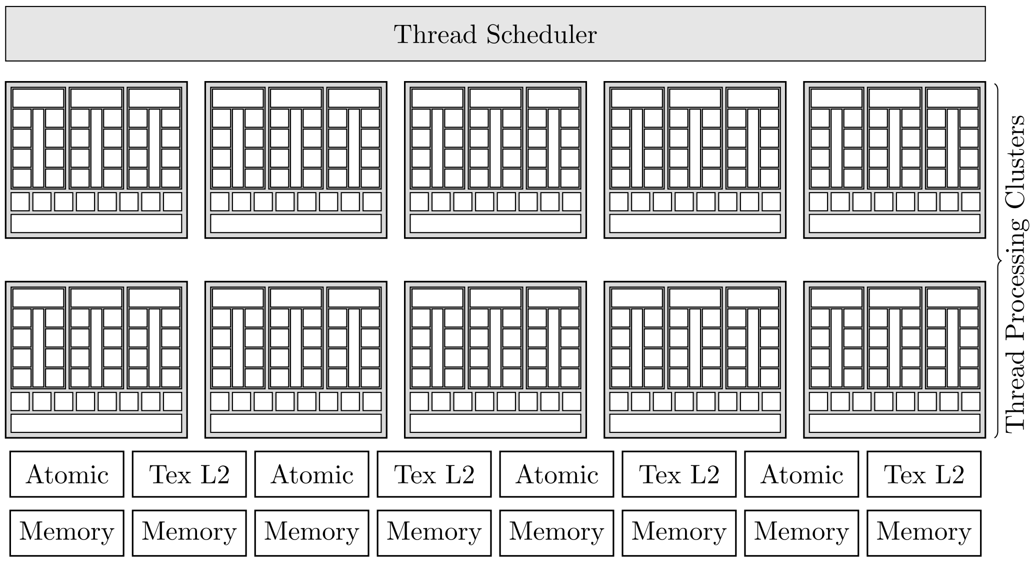


Figure 7.19: Architecture of the NVIDIA GeForce GTX 280 graphics processor [NVI08]

The additional flexibility of programmable PEs was also recognized by manufacturers of graphic processors. As the graphics processing pipeline evolved and became fully customizable, dedicated programmable units, called *shaders* were introduced. Depending on the location in the pipeline they are called *geometry*, *vertex* and *fragment shaders*. Each GPU features a certain number of each of these units on silicon. However, depending on the requirements of a single scene different number of units are required. For example, complex surfaces require require a larger number of vertex shaders, whereas for complex lighting and surface properties more fragment shares are used. In either situation, the silicon inefficiently used. As a consequence, both major graphics hardware vendors decided to replace the different shaders by a *unified shader model.* They represent generic processing elements on silicon that can be used as vertex, geometry, or fragment shader depending on the programming.

7.8.2 Graphic Processors

Modern graphics processors can be used for general purpose computing. Using specialized programming environments such as CUDA [NBGS08] or OpenCL the large number of parallel processing elements can be leveraged to perform general purpose computing. This subsection discusses whether the graphics hardware is suitable to implement query processing following the high-level approach using fully programmable processing elements.

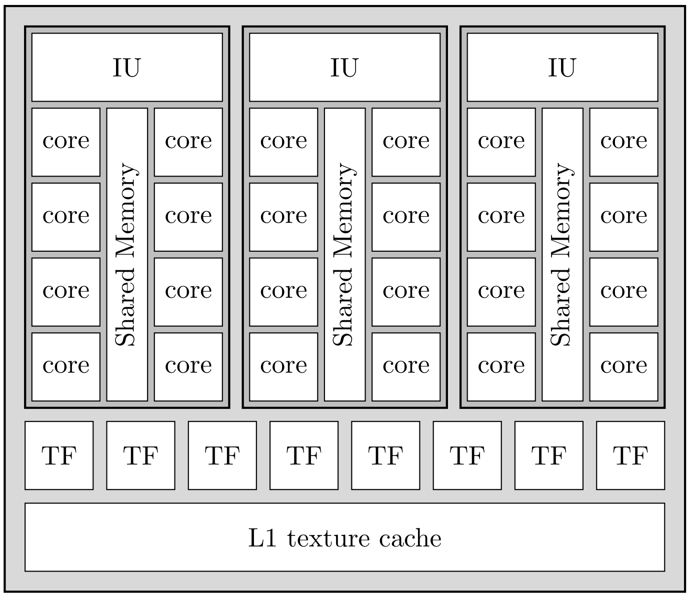


Figure 7.20: A single Thread Processing Cluster (TPC) of the NVIDIA GeForce GTX 280 graphics processor consists of three Stream Multiprocessors with eight cores each. IU: Instruction Unit, TF: Texture Filter. [NVI08]

Figure 7.19 shows the architecture of a modern GPU [NVI08] at the time of writing. The NVIDIA GeForce GTX 280 contains ten *thread processing clusters* (TPCs), an 8-port access to the device-memory, and two levels of cache. Figure 7.20 shows a TPC in more detail. Each cluster consists of three *stream multiprocessors*. Each multiprocessor in turn contains eight processing cores, which results in a total 240 processing cores on the GPU. In order to be able to put such a large number of cores onto a single chip the manufacturer had to make a few compromises. The most limiting factor is the connectivity among the cores. The encapsulation of cores into multiprocessors and TPCs has lead a limited collaboration model for the cores but it has avoided the otherwise difficult signal routing problems during the design of the chip.

Only cores withing the same multiprocessor can collaborate through shared memory. Cores between different multiprocessors can only exchange data through the global device memory. The access can be coordinated through *Atomic Operation Units* (see Figure 7.19), which provide primitives such as addition, exchange, increment, compare-and-swap, etc. The use of such operations, however, introduces a significant delay.

The architectures of GPUs do not facilitate data flow processing on the many cores due to the lack of the direct communication links but also due to the execution model. Graphic tasks are highly parallel and typically also trivial to parallelize, for example, by partitioning the screen and assigning each partition to one core. GPUs are thus designed such that all cores execute the same code. An additional restriction is given by the CUDA architecture. Within each multiprocessor the eight cores share one single *instruction unit* (UI) (Figure 7.20). This unit is responsible to schedule threads in units of *wraps* [NBGS08] with the constraint that all threads within a warp must be at the same instruction address. The use of branching generates *diverging warps*, which significantly reduces performance. A final limitation is that only one single piece of code, called kernel, can be run on the GPU at any time. Kernels are scheduled sequentially. Together with the missing direct core interconnect this renders the GPU unusable to perform the type of processing done in Glacier. Nevertheless, GPUs have shown significant performance improvements for certain applications such as sorting [GGKM06]. However, it is unlikely that they can be used as a generic query execution platform such as FPGAs using Glacier.

7.9 Related Work

The idea of using tailor-made hardware for database processing dates back at least to the late 1970s, when DeWitt explored what was called a "database machine" at the time [DeW79]. His Direct system used specialized co-processors that operated close to secondary storage and provided explicit database support in its instruction set.

While enormous chip fabrication costs rendered the idea not economical at the time, some companies started to commercialize similar setups recently. Sold as "database appliances", their systems provide hardware acceleration mostly for data warehousing workloads. Documentation about the inner workings of any of the available systems is rare, but it seems that some of the appliances have a lot in common with the configurations we considered in our work.

The *Netezza Performance Server* (NPS) system [Net09] is built from a number of "snippet processing units" (SPUs). Each of these snippets includes a magnetic disk, tightly coupled with a network card, a CPU, and an FPGA. Similar to the setup we consider, the FPGA is used to filter data close to the data source (the disk in Netezza's case).

The heart of Kickfire's *MySQL Analytic Appliance* [Kic09] is its so-called "SQL Chip". Judging from the product documentation, this chip seems to be bundled with DDR2 memory and connected to the base system via PCI Express. In essence, this appears to coincide with the co-processor setup that we briefly touched in Section 7.2.1 (see Figure 7.2(b) on page 218).

XtremeData [Xtr09] sells "in-socket accelerators" where an FPGA chip is mounted onto an interface that can be plugged into an AMD HyperTransport socket. From a system architecture point of view, this has the potential of offering a very large bandwidth and low latency for FPGA↔CPU communication. Publicly available documentation, however, does not specify whether the company's "Database Appliance" is indeed based on such accelerators.

Both systems appear to use FPGAs primarily as customized hardware, with circuits that are geared toward very specific (data warehousing) workloads, but are immutable at runtime. In our work, we aim at exploiting the *configurability* of FPGAs. With *Glacier*, we present a compiler that compiles queries from an arbitrary workload into a tailor-made hardware circuit.

The processing model used in glacier resembles the MonetDB/X100 system by Héman et al. [HZdVB07]. MonetDB/X100 processes data from a column-wise storage in a pipelined fashion. We borrowed their idea of *selection vectors*. Invalidating tuples rather than physically deleting them avoids expensive in-memory copy operations in MonetDB/X100. Much like MonetDB/X100, our circuits favor narrow input relations/streams.

Our implementation of *group by* takes particular advantage of an FPGA-based implementation of content-addressable memory. Bandi et al. [BMAE07] have looked at a commercial CAM product and its potential applications in a database context. Though such products can provide high capacity and lookup performance, we think that the missing coupling to a full database infrastructure renders the approach hard to apply in practice. FPGAs, by contrast, provide the flexibility to join an existing infrastructure in a seamless fashion, even at different locations if necessary. The work of Gold et al. [GAHF05] describes a similar approach with similar drawbacks. They suggest the use of *network processors* for database processing, mainly to exploit the thread-level parallelism inherent to network CPUs.

The higher-order nature of the *group by* and windowing operators in our streaming algebra resembles the "Apply" operator that is used inside Microsoft SQL Server and has been discussed by Galindo-Legaria et al. [GLJ01]. Similar rewrite rules as the ones in SQL Server may also help the *Glacier* compiler to improve the quality of generated plans.

The data flow present in the Handshake join flow is similar to the *join arrays* proposed by Kung and Lohman [KL80]. Inspired by the then-new concept of *systolic arrays* in VLSI designs, their proposed VLSI join implementation uses an array of bit comparison components, through which data is shifted in opposing directions.

The only work we could find on stream joins using FPGAs is the *M3Join* of Qian et al. [QXD^{+}05], which essentially implements the join step as a single parallel lookup. This approach is known to be severely limited by on-chip routing bottlenecks [TMA10], which causes the sudden and significant performance drop observed by Qian et al. for larger join windows [QXD^{+}05]. The pipelining mech-

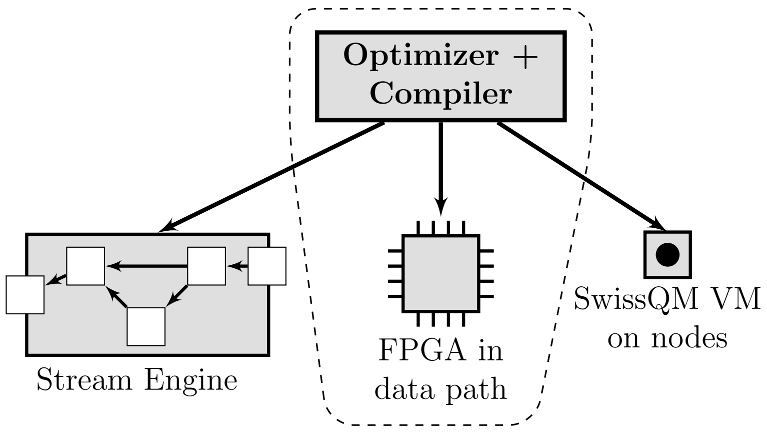


Figure 7.21: Glacier translates queries into hardware execution plans on FPGAs

anism of handshake join, by contrast, does not suffer from these limitations.

7.10 Summary

The Glacier compiler described in this chapter adds the missing piece to the heterogeneous query execution platform and completes the system shown in Figure 7.21.

Glacier provides an operator algebra and transformation rules that can be used to convert meaningful continuous queries into FPGA circuits. Among others, we provide full support for aggregation, grouping, windowing, and stream joins. Since the performance characteristics of the operators implemented as FPGA circuits are very different from those of software operators, we provided an in-depth analysis of the relevant performance metrics.

Our results indicate that using the FPGA as a co-processor in an engine running on conventional CPUs can have significant advantages. The experiments show that most operators have very low latency and that the FPGA as a whole can sustain a very high throughput. The tested setup demonstrates that the FPGA can process streams at network speed (the bottleneck is the network interface, not the data stream processing on the FPGA), something that cannot be done in conventional CPUs.

We presented Handshake join and demonstrated how *window-based stream joins* can be parallelized over very large numbers of processing elements with negligible coordination overhead. The implementation shows good performance and scalability behavior on the FPGA. Key to the scalability of handshake join is to avoid any coordination by a centralized entity. Instead, handshake join only relies on short element-to-element communication. This mode of parallelization is related to data processing *systolic arrays*.

8 Summary and Conclusions

This dissertation has contributed towards a heterogeneous solution for stream processing. The thesis studied the extension of stream processing onto embedded systems platforms: wireless sensor networks and FPGAs. The envisioned architecture of the heterogeneous system introduced in Chapter 1 is shown again in Figure 8.1.

The approach is the same in both cases. Continuous streaming queries are compiled into bytecode programs for sensor networks and into FPGAs circuits. This is made possible by introducing an intermediate abstraction level. For sensor network this additional abstraction level is provided by the bytecode interface of *SwissQM* and the for FPGA by the *Glacier* library that contains well-defined query operator components. Essentially, Glacier can be considered as a different compiler backend of the SwissQM/Gateway.

This allows the compiler to reason about resource consumption of the individual execution plans. For the bytecode interface the resource consumption can be estimated through the program length and the memory requirements. The communication complexity can be estimated depending on aggregation and non-aggregation patterns by considering the current topology parameter of the collection tree. For FPGAs the latency and throughput, i.e., tuple issue rate, can be determined upfront from the circuits. This provides predictable performance values of the designs.

Stream processing tasks are expressed as declarative queries. This can be considered as programming using a domain specific language. The work furthermore illustrates that the complexity of the underlying technology (embedded systems

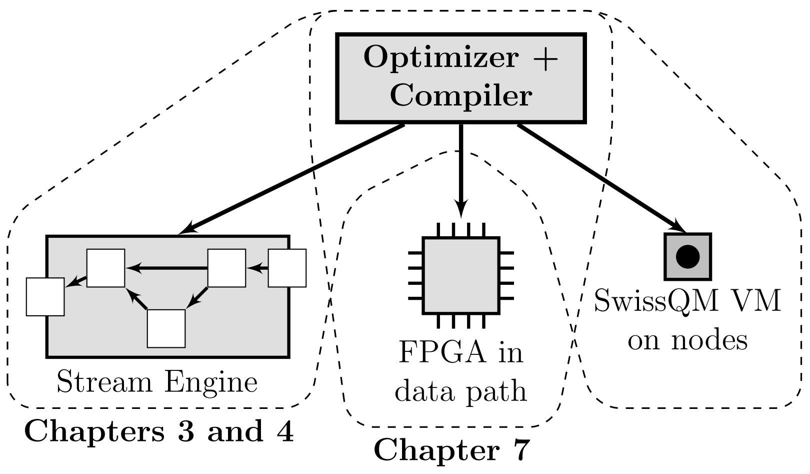


Figure 8.1: Components of heterogeneous query processing platform developed as part of this dissertation

programming) and the design of digital circuits for FPGAs can be hidden. In fact, the abstraction levels and the domain specific language open the inherently complex technology to broader range of users and developers.

Figure 8.1 shows that Glacier consists of the compiler and optimizer component only. The system developed as part of this dissertation is by no means complete in the sense that it is a heterogeneous system that spans all three execution platforms at the same time. The ultimate realization of the architecture requires that the cost models used for optimizing plans for sensor networks and FPGA circuits need to be combined. The thesis contributes to such a solution by characterizing the individual cost models and describes how such as system can be implemented.

8.1 Query Processing in Sensor Networks

Chapter 3 shows that running a virtual machine on sensor nodes provides more flexibility than running a query engine such as TinyDB [MFHH05]. It can provide more general query expressions at a lower implementation complexity on the sensor nodes. It further allows the use of user-defined functions that can be inlined into the bytecode and directly executed on the sensor nodes. User-defined functions in sensor networks can perform important preprocessing and filtering such as cleaning of noisy sensor data [JAF$^+$06].

The query merging approach presented in Chapter 4 allows to reduce the number of concurrent queries that are run in the network. One observation made is that for a large number of concurrent queries, the optimization work is actually reduced, as in this regime the *universal network query* can be used. The universal query retrieves data from all sensors as fast as possible. The evaluation also showed that merging all user queries to one single network execution plan, in general, is not advisable. Instead, queries have to be merged into groups. We presented different strategies to perform this type of multi-query optimization. In order to select the best mapping of queries to network execution plans the strategies make use of a cost model. We defined an energy-based cost model whose parameters were determined for a real sensor platform. Power measurements for the Tmote Sky platform confirm that in sensor networks the energy cost is dominated by radio communication. For this platform, in particular, sampling and computation is free. The presented strategies do not work of equally well for all query loads. We defined a simple heuristic for the optimizer when to switch to a different strategy.

8.2 Query Processing on FPGAs

We first analyzed of the behavior of FPGAs as computing platform. This was shown in the context of sorting, in particular, sorting networks. We studied the impact of different design techniques such as *purely combinational*, *synchronous*, and *pipelined* implementations on chip area and performance. For pipelined designs we could obtain very high throughputs ($>$ 50 GB/s for sorting 64 32-bit elements) in our sorting core. However, when considering the end-to-end performance after integrating that core into a complete system the performance advantage is severely reduced. When connected to the system bus and streaming in the data from external DRAM memory on our prototyping board, the effective bandwidth is reduced to 400 MB/s. This illustrates the importance an efficient attachment of the custom FPGA logic. We also analyzed the attachment of an accelerator core to the execution pipeline of an embedded CPU. This allows a tighter coupling to the traditional software stack running on the CPU, however, in this case it does not solve the memory bottleneck problem.

After the analysis we defined an algebra that we later used to express streaming queries. We then introduced translation rules for the operators of the algebra to translate algebraic plans into digital hardware circuits. The operators are part of our hardware component library *Glacier*. It contains traditional operators such as projection, selection, windowing, grouping, and window-based joins. Additional helper operators can be used to interface the circuit with the CPU and the network, to align tuple streams and to compute unions. The operators can be used for queries that have constant space requirements, e.g., tuple-based windows and an a priori known number of groups. The thesis also presents a novel idea for implementing window-based stream joins. The approach leads to a very efficient and scalable implementation due to the inherent parallelism and the locality in the communication pattern.

The advantage of the *Glacier* approach is that the performance behavior can be directly determined from the components after composition. Latency can be computed by counting the register stages and throughput by well-defined issue rates of the individual operators. This provides up front guarantees on the circuits performance behavior. Both, latency and throughput are given in units of clock cycles. The actual clock rate is determined during synthesis by the tools.

8.3 Directions for Future Work

In *Glacier* queries are compiled into HDL code that is then statically compiled into a FPGA bitstream using the traditional FPGA tool flow. Some FPGAs provide functionality that allow the dynamic partial reconfiguration of the chip, i.e., the functionality of the entire chip or parts thereof can be modified at runtime. An extension of this work could wrap the components from the Glacier library into partial reconfiguration modules. These modules can then be placed in partial reconfiguration regions on the chip such that query plans can be connected on the fly at runtime. The difficulty is that partial reconfiguration imposes many additional constraints to the layout of the modules. A suitable approach to place *bus macros* has to be devised. While partial reconfiguration introduces runtime adaptation it also comes at a cost. The static compilation of execution plans shown in this thesis has the advantage that optimization can be applied on a global scale during synthesis because module boundaries are removed. By connecting individual, pre-compiled components onto the chip using partial reconfiguration this global view is not available during optimization. This introduces the research question of how much loss in circuit performance partial reconfiguration costs for the query plans expressed through *Glacier*.

As pointed out earlier, an efficient attachment of the FPGA is crucial to the overall performance of the accelerator. For a full system implementation in a commodity computer system of today the attachment to the embedded CPU as used in this thesis is not suitable. Although, the use of the embedded CPU does not have an impact on statements made in thesis about the compiled query circuits, in a commercial solution, the FPGA has to be connected a host system. To this end, the set of adapter components of *Glacier* has to be extended by, e.g., a PCI-Express bus interface. Additionally, this work assumes queries with constant space requirements. Since *Glacier* uses on-chip storage this amount of space available for query state is fairly limited, e.g., the amount of Block RAM available on the Virtex-5 FX130T chip used in the experiments is 1.3 MB. Hence, external memory (DRAM) has to be used for execution plans with large state requirements (e.g., large windows). This will lead to memory hierarchies similar to caches in traditional CPUs. The query execution model needs to account for this. It is also possible that very large plans do not fit onto the FPGA even though they have small state needs (but use a significant amount of logic). In this case, virtualization techniques such as PipeRench [SWT+02] by Schmit et al. could be used. Virtualization of FPGAs further increases the complexity of the cost model and the implementation.

Data stream processing as presented in this dissertation is related to signal processing. The relation can be explored in more detail. Data rates in SwissQM are in the order of a tuple every few seconds. An extension of this work could consist of increasing the sampling rate to a few 100 Hz. The will lead to a different messaging in the sensor network. Filtering, both by evaluating predicates as well as processing in the signal processing sense has to be provided efficiently on the sensor nodes, using application-specific instructions. Additionally, shift-register like windowing as used in signal processing also needs to be provided as an extension to SwissQM.

Asynchronous rate conversion provides the most general realization of the rate conversion operator ρ. Asynchronous rate conversion can be used in a stream engine on time series data with tuple data from a continuous range. Rate conversion provides an alternative to load shedding, i.e., dropping tuples in overload situations [TcZ+03]. Rothacher describes [Rot95] VLSI implementations of a conversion that can also be used on an FPGA. FPGAs have already been collocated with embedded micro-controllers on a sensor node for signal processing applications [SMAL+04]. For battery-driven sensor nodes low-powered flash-based FPGAs could be used.

The query compilation approach can be extended to different hardware architecture such as graphics processors, massively parallel processor arrays (MPPAs) [But07] and many-core multi-processors such as the Intel SCC [HDH+10]. For the different platforms new cost models have to be developed. Even today's multicore computers start to exhibit a communication pattern among cores that are very similar the one of a network. In fact, Baumann et al. [BBD+09] also use tree-based communication for exchanging cache-line-sized messages among cores. The techniques described in this dissertation many also be applied in the context of chip multi-processor systems.

Bibliography

[AAB$^+$05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Çetintemel, Mitch Cherniack, Jeong-Hwang Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the Borealis stream processing engine. In *CIDR '05: Proceedings of the 2rd biennial conference on Innovative data systems research*, pages 277–289, January 2005.

[AAJG07] Asad Asad Awan, Suresh Jagannathan, and Ananth Grama. Macroprogramming heterogeneous sensor networks using COSMOS. *SIGOPS Oper. Syst. Rev.*, 41(3):159–172, 2007.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[ACc$^+$03] Daniel J. Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, July 2003.

[AFSS00] Ray Andraka, Philip Friedin, Satnam Singh, and Tim Southgate. The john henry syndrome: humans vs. machines as FPGA designers (panel session). In *FPGA '00: Proceedings 8th international symposium on Field programmable gate arrays*, page 101, Monterey, California, United States, 2000. ACM.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD '00: Proceedings of the SIGMOD international conference on Management of data*, pages 261–272, Dallas, TX, USA, May 2000. ACM.

[AKS83] Miklos Ajtai, Janos Komlós, and Endre Szemerédi. An O(n log n) sorting network. In *STOC '83: Proceedings of the 15th annual ACM* *symposium on Theory of computing*, pages 1–9, Boston, MA, USA, April 1983. ACM.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, Atlantic City, New Jersey, May 1968. ACM.

[BBD$^+$09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP '09: Proceedings of the 22nd symposium on Operating systems principles*, pages 29–44, Big Sky, Montana, USA, June 2009. ACM.

[BCD$^+$05] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10(4):563–579, 2005.

[BCKL98] Wayne P. Burleson, Maciej Ciesielski, Fabian Klass, and Wentai Liu. Wave-pipelining: a tutorial and research survey. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 6(3):464–474, September 1998.

[BCSS99] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. *SIGPLAN Not.*, 34(1):174–184, 1999.

[BDMT05] Jan Beutel, Matthias Dyer, Lennart Meier, and Lothar Thiele. Scalable topology control for deployment-support networks. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 47, Los Angeles, CA, USA, 2005. IEEE Press.

[Beu06] Jan Beutel. Fast-prototyping using the BTnode platform. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 977–982, Munich, Germany, 2006. European Design and Automation Association.

[BGH$^+$05] Phil Buonadonna, David Gay, Joseph M. Hellerstein, Wei Hong, and Samuel Madden. TASK: Sensor network in a box. In *EWSN '05: Proceedings of the 2nd European Workshop on Wireless Sensor Networks*, pages 133–144, Istanbul, Turkey, January 2005.

[BGS00] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Querying the physical world. *Personal Communications, IEEE*, 7(5):10–15, October 2000.

[BGS01] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *MDM '01: Proceedings of the 2nd international conference on Mobile Data Management*, pages 3–14. Springer-Verlag, 2001.

[Bit09] Ray Bittner. Bus mastering PCI express in an FPGA. In *FPGA '09: Proceeding of the international symposium on Field programmable gate arrays*, pages 273–276, Monterey, California, USA, February 2009. ACM.

[BLC02] Eric Bruneton, Romain Lenglet, and Theirry Coupaye. ASM: un outil de manipulation de code pour la réalisation de systèmes adaptables. In *Jounées Composants '02: 4ème Conférence Francophone autour des Composants Logiciels*, Grenoble, France, November 2002.

[BM02] Luca Benini and Giovanni De Micheli. Networks on chips: a new SoC paradigm. *Computer*, 35(1):70–78, January 2002.

[BMAE07] Nagender Bandi, Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Fast data stream algorithms using associative memories. In *SIGMOD '07: Proceedings of the SIGMOD international conference on Management of data*, pages 247–256, Beijing, China, June 2007. ACM.

[But07] Mike Butts. Synchronization through communication in a massively parallel processor array. *IEEE Micro*, 27(5):32–40, 2007.

[CABM03] Douglas S. J. De Couto, Daniel Aguayo, John Bicket, and Robert Morris. A high-throughput path metric for multi-hop wireless routing. In *MobiCom '03: Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 134–146, San Diego, CA, USA, 2003. ACM.

[CBGM03] Arturo Crespo, Orkut Buyukkokten, and Hector Garcia-Molina. Query merging: Improving query subscription processing in a multicast environment. *IEEE Transactions on Knowledge and Data Engineering*, 15(1):174–191, 2003.

[CJ03] Thomas H. Clausen and Phillippe Jacquet. *Optimized Link State Routing Protocol (OLSR) (IETF RFC 3626).* The Internet Society, October 2003.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 2nd edition, 2001.

[CNL$^+$08] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proc. VLDB Endowment*, 1(2):1313–1324, 2008.

[CP34] C. J. Clopper and E. S. Pearson. The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, 26(4):404–413, 1934.

[CSCM00] Lars Ræder Clausen, Ulrik Pagh Schultz, Charles Consel, and Gilles Muller. Java bytecode compression for low-end embedded systems. *ACM Trans. Program. Lang. Syst.*, 22(3):471–489, 2000.

[CSS03] Koen Claessen, Mary Sheeran, and Satnam Singh. Using lava to design and verify recursive and periodic sorters. *International Journal on Software Tools for Technology Transfer (STTT)*, 4:349–358, 2003.

[DeH99] André DeHon. Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100 % lut utilization). In *FPGA '99: Proceedings of the 7th international symposium on Field programmable gate arrays*, pages 69–78, Monterey, CA, USA, February 1999. ACM.

[DeW79] David J. DeWitt. DIRECT—a multiprocessor organization for supporting relational database management systems. *IEEE Trans. on Computers*, 28(6):395–406, June 1979.

[DGM$^+$05] Amol Deshpande, Carlos Guestrin, Samuel Madden, Joseph M. Hellerstein, and Wei Hong. Model-based approximate querying in sensor networks. *VLDB Journal*, 14(4):417–443, 2005.

[DGV04] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki–a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, November 2004.

[DM06] Amol Deshpande and Samuel Madden. MauveDB: supporting model-based user views in database systems. In *SIGMOD '06: Proceedings of the SIGMOD international conference on Management of data*, pages 73–84, Chicago, IL, USA, June 2006. ACM.

[DPD10] Marguerite Doman, Jamie Payton, and Theresa Dahlberg. Leveraging fuzzy query processing to support applications in wireless sensor networks. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 764–771, Sierre, Switzerland, 2010. ACM.

[FAN07] Timothy Furtak, Joseé Nelson Amaral, and Robert Niewiadomski. Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms. In *SPAA '07: Proceedings of the 19th annual ACM symposium on Parallel algorithms and architectures*, pages 348–357, San Diego, California, USA, 2007. ACM.

[FIX09] FIX Protocol Organization. FIX protocol specification, April 2009. `http://fixprotocol.org/specifications`.

[FSG02] Wai Fu Fung, Daivd Sun, and Johannes Gehrke. COUGAR: the network is the database. In *SIGMOD '02: Proceedings of the SIGMOD international conference on Management of data*, pages 621–621, Madison, WI, USA, June 2002. ACM.

[GAHF05] Brain T. Gold, Aanastassia Ailamaki, Larry Huston, and Babak Falsafi. Accelerating database operators using a network processor. In *DaMoN '05: Proceedings of the 1st international workshop on Data management on new hardware*, pages 1–6, Baltimore, Maryland, June 2005. ACM.

[GBLP96] Jim Gray, Adam Bosworth, Andrew Lyaman, and Hamid Pirahesh. Data Cube: A relational aggregation operator generalizing GROUP-BY, CROSS-TAB, and SUB-TOTALS. In *ICDE '96: Proceedings of the 12th international Conference on Data Engineering*, pages 152–159, New Orleans, LA, USA, February 1996. IEEE Computer Society.

[GBY07] Buğra Gedik, Rajesh R. Bordawekar, and Philip S. Yu. CellSort: High performance sorting on the Cell processor. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1286–1297, Vienna, Austria, September 2007. VLDB Endowment.

[GCM$^+$08] John F. Gantz, Christopher Chute, Alex Manfrediz, Stephen Minton, David Reinsel, Wolfgang Schlichting, and Anna Toncheva. The diverse and exploding digital universe. IDC white paper, International Data Corporation, Framingham, Massachusetts, USA, March 2008. `http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf` (accessed May 6, 2010).

[GFJ$^+$09] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection tree protocol. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, Berkeley, California, 2009. ACM.

[GGG05] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming wireless sensor networks using *airos*. In *DCOSS '05: Proceedings of the 1st IEEE international conference on Distributed Computing in Sensor Systems*, pages 126–140, Marina del Rey, CA, USA, June 2005. Springer.

[GGKM06] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High performance graphics co-processor sorting for large database management. In *SIGMOD '06: Proceedings of the SIGMOD international conference on Management of data*, pages 325–336, Chicago, IL, USA, June 2006. ACM.

[GHF$^+$06] Miachel Gschwind, H. Peter Hofstee, Brian Flachs, Marin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[GJP$^+$06] Omprakash Gnawali, Ki-Young Jang, Jeongyeup Paek, Marcos Vieira, Ramesh Govindan, Ben Greenstein, August Joki, Deborah Estrin, and Eddie Kohler. The Tenet architecture for tiered sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 153–166, Boulder, CO, USA, 2006. ACM.

[GKS03] Saurabh Ganeriwal, Ram Kumar, and Mani B. Srivastava. Timing-sync protocol for sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 138–149, Los Angeles, CA, USA, 2003. ACM.

[GKW$^+$02] Deepak Ganesan, Bhaskar Krishnamachari, Alec Woo, David Culler, Deborah Estrin, and Stephen Wicker. An empirical study of epidemic algorithms in large scale multihop wireless networks. Technical Report IRB-TR-02-003, Intel Research, March 2002.

[GLD00] Steve Guccione, Delon Levi, and Daniel Downs. A reconfigurable content addressable memory. In *RAW: '00; Proceedings of 7th Reconfigurable Architectures Workshop*, pages 882–889, Cancún, Mexico, May 2000. Springer.

[GLJ01] César Galindo-Legaria and Milind Joshi. Orthogonal optimization of subqueries and aggregation. In *SIGMOD '01: Proceedings of the SIGMOD international conference on Management of data*, pages 571–581, Santa Barbara, CA, USA, May 2001. ACM.

[GLvB+03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, San Diego, CA, USA, 2003. ACM.

[GLW+04] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *SIGMOD '04: Proceedings of the SIGMOD international conference on Management of Data*, pages 215–226, Paris, France, June 2004. ACM.

[GM04] Johannes Gehrke and Samuel Madden. Query processing in sensor networks. *Pervasive Computing, IEEE*, 3(1):46–55, January–March 2004.

[GMN+07] Lewis Girod, Yuan Mei, Ryan Newton, Stanislav Rost, Arvind Thiagarajan, Hari Balakrishnan, and Samuel Madden. The case for a signal-oriented data stream management system. In *CIDR '07: Proceedings of the 3rd biennial conference on Innovative data systems research*, pages 397–406, January 2007.

[GS08] David J. Greaves and Satnam Singh. Kiwi: Synthesis of FPGA circuits from parallel programs. In *FCCM '08: Proceedings of the IEEE Symposium on Field-programmable custom computing machines*, pages 3–12, Stanford, CA, USA, 2008.

[HC04] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, Baltimore, MD, USA, 2004. ACM.

[HDH+10] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabrice Pailet, Shailendra Jain, Tiji Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Apel, Kersten Henrissi, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob Van Der Wijngaart, and Timothy Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *ISSCC '10: Proceedings of the IEEE international Solid-State Circuits Conference*, pages 108–109, February 2010.

[HHBR08] Shan Shan Huang, Amir Hormati, David F. Bacon, and Rodric Rabbah. Liquid Metal: Object-oriented programming across the hardware/software boundary. In *ECOOP '08 Proceedings of the 22nd European conference on Object-oriented programming*, Paphos, Cyprus, July 2008.

[HL10] Gertjan Halkes and Koen Langendoen. Practical considerations for sensor network algorithms. Technical Report ES-2010-01, ISSN 1877-7805, Delft University of Technology, Delft, The Netherlands, 2010.

[HP02] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 3rd edition, 2002.

[HSA05] Stravros Harizopoulos, Vlaislav Shkapenyuk, and Anastassia Ailamaki. QPipe: A simultaneously pipelined relational query engine. In *SIGMOD '05: Proceedings of the SIGMOD international conference on Management of data*, pages 383–394, Baltimore, MD, USA, June 2005. ACM.

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of ASPLOS-IX 2000*, pages 93–104, Cambridge, MA, USA, 2000.

[HZdVB07] Sándor Héman, Marcin Zukowski, Arjen P. de Vries, and Peter A. Boncz. Efficient and flexible information retrieval using monetdb/x100. In *CIDR '07: Proceedings of the 3rd biennial conference on Innovative data systems research*, pages 96–101, January 2007.

[IEE97] IEEE. *Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications (ANSI/IEEE Std 802.11-1997).* Institute of Electrical and Electronics Engineers, June 1997.

[IEE03] IEEE. *Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs) (ANSI/IEEE Std 802.15.4-2003)*. Institute of Electrical and Electronics Engineers, October 2003.

[IGE$^+$03] Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, John Heidemann, and Fabio Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Trans. Netw.*, 11(1):2–16, 2003.

[IMKN07] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. AA-Sort: A new parallel sorting algorithm for multi-core SIMD processors. In *PACT '07: Proceedings of the 16th international conference on Parallel Architecture and Compilation Techniques*, pages 189–198, September 2007.

[JAF$^+$06] Shawn R. Jeffery, Gustavo Alonso, Michael J. Franklin, Wei Hong, and Jennifer Widom. Declarative support for sensor data cleaning. In *PERVASIVE '06: Proceedings of the 4th international conference on Pervasive Computing*, pages 83–100, Dublin, Ireland, May 2006.

[JOW$^+$02] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with ZebraNet. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 96–107, San Jose, CA, USA, 2002. ACM.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[KDD04] Uew Kubach, Christian Decker, and Ken Douglas. Collaborative control and coordination of hazardous chemicals. In *SenSys '04: Proceedings of the 2nd International Conference on Embedded networked sensor systems*, pages 309–309, Baltimore, MD, USA, 2004. ACM.

[Kic09] Kickfire, 2009. `http://www.kickfire.com/`.

[KKP99] Joseph M. Kahn, Randy H. Katz, and Kristofer S. J. Pister. Next century challenges: mobile networking for "smart dust". In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 271–278, Seattle, WA, USA, 1999. ACM.

[KL80] H. T. Kung and Philip L. Lohman. Systolic (VLSI) arrays for relational database operations. In *SIGMOD '80: Proceedings of the SIGMOD international conference on Management of data*, pages 105–116, Santa Monica, CA, USA, May 1980. ACM.

[KMd+06] Nachiket Kapre, Nikil Mehta, Michael deLorimier, Raphhael Rubin, Henry Barnor, Michael J. Wilson, Michael Wrighton, and Andrew DeHon. Packet switched vs. time multiplexed fpga overlay networks. In *FCCM '06: Proceedings of the IEEE Symposium on Field-Programmable custom computing machines*, pages 205–216, Napa, CA, USA, 2006.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, 2nd edition, 1998.

[KNV03] Jaewoo Kang, Jeffrey F. Naughton, and Stratis D. Viglas. Evaluating window joins over unbounded streams. In *ICDE '03: Proceedings of the 19th international Conference on Data Engineering*, pages 341–352, Bangalore, India, March 2003.

[LBV06] Koen Langendoen, Aline Baggio, and Otto Visser. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *IPDPS '06: Proceedings of the 20th IEEE international symposium on Parallel and Distributed Processing*, pages 8 pp., Rhodes, Greece, April 2006.

[LC02] Philip Levis and David Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, San Jose, California, 2002. ACM.

[LGC05] Philip Levis, David Gay, and David Culler. Active sensor networks. In *NSDI '05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 343–356. USENIX Association, 2005.

[Li08] Jun Li. VM-based event-processing in sensor networks. Master's thesis, ETH Zurich, Switzerland, July 2008.

[LLWC03] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, Los Angeles, California, USA, 2003. ACM.

[LMT$^+$05] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Peter A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD '05: Proceedings of the SIGMOD international conference on Management of data*, pages 311–322, Baltimore, MD, USA, June 2005. ACM.

[LPCS04] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *NSDI '04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 2–2, San Francisco, California, 2004. USENIX Association.

[LPL$^+$09] Hong Lu, Wei Pan, Nicolas D. Lane, Tanzeem Choudhury, and Andrew T. Campbell. Soundsense: scalable sound sensing for people-centric applications on mobile phones. In *MobiSys '09: Proceedings of the 7th international conference on Mobile systems, applications, and services*, pages 165–178, Kraków, Poland, 2009. ACM.

[LY98] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley Professional, second edition, 1998.

[MA06] Rene Mueller and Gustavo Alonso. Efficient sharing of sensor networks. In *MASS '06: Proceedings of 3rd IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, pages 109–118, Vancouver, Canada, October 2006.

[MAK07a] Rene Mueller, Gustavo Alonso, and Donald Kossmann. SwissQM: Next generation data processing in sensor networks. In *CIDR '07: Proceedings of the 3rd biennial conference on Innovative data systems research*, pages 1–9, 2007.

[MAK07b] Rene Mueller, Gustavo Alonso, and Donald Kossmann. A virtual machine for sensor networks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 145–158, Lisbon, Portugal, 2007. ACM.

[MBK00] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: Memory access. *The VLDB Journal*, 9(3):231–246, December 2000.

[McG06] Dylan McGrath. FPGA market to pass $2.7 billion by '10, In-Stat says. *EE Times*, May 2006.

[MCP$^+$02] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, Atlanta, Georgia, USA, 2002. ACM.

[ME09] Rene Mueller and Ken Eguro. FPGA-accelerated deserialization of object structures. Technical Report MSR-TR-2009-126, Microsoft Research Redmond, September 2009.

[MFHH02] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A Tiny AGgregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002.

[MFHH05] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[Mic] Microsoft. StreamInsight: SQL Server 2008 — complex event processing technology. `http://www.microsoft.com/sqlserver/2008/en/us/r2-complex-event.aspx` (accessed May 1, 2010).

[MP06] Luca Mottola and Gian Pietro Picco. Logical neighborhoods: A programming abstraction for wireless sensor networks. In *DCOSS '06: Proceedings of the 2nd IEEE international conference on Distributed Computing in Sensor Systems*, pages 150–168, San Francisco, CA, USA, June 2006. Springer.

[MRDA07] Rene Mueller, Jan S. Rellermeyer, Michael Duller, and Gustavo Alonso. Demo: A generic platform for sensor network applications. In *MASS '07: Proceedings of 3rd IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, pages 1–3, Pisa, Italy, October 2007.

[MSHR02] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD '02: Proceedings of the SIGMOD international conference on Management of data*, pages 49–60, Madison, WI, USA, June 2002. ACM.

[MTA09a] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on FPGAs. *Proc. VLDB Endow.*, 2(1):910–921, 2009.

[MTA09b] Rene Mueller, Jens Teubner, and Gustavo Alonso. Streams on wires: a query compiler for FPGAs. *Proc. VLDB Endow.*, 2(1):229–240, 2009.

[MVB$^+$09] Abhishek Mitra, Marcos R. Vieira, Petko Bakalov, Vassilis J. Tsotras, and Walid Najjar. Boosting XML filtering through a scalable FPGA-based architecture. In *CIDR '09: Proceedings of the 4rd biennial conference on Innovative data systems research*, 2009.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[Net] Netronome. Netronome flow engine acceleration cards, white paper. `http://www.netronome.com`.

[Net09] Netezza, 2009. `http://www.netezza.com/`.

[NMW07] Ryan Newton, Greg Morrisett, and Matt Welsh. The Regiment macroprogramming system. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 489–498, Cambridge, Massachusetts, USA, 2007. ACM.

[NVI08] NVIDIA. NVIDIA GeForce GTX 200 GPU architectural overview. Technical Report TB-4044-001_v01, NVIDIA Inc., May 2008.

[Ofl83] Kemal Oflazer. Design and implementation of a single-chip 1-d median filter. *IEEE Trans. on Acoustics, Speech and Signal Processing*, 31:1164–1168, October 1983.

[OPR09] Options Price Reporting Authority OPRA. Traffic projections 2009/2010, January 2009. `http://www.opradata.com/specs/Traffic_Projections_2009_2010.pdf`.

[Ora] Oracle. Oracle streams. `http://www.oracle.com/`.

[OSG09] OSGi Alliance. *OSGi Service Platform: Core Specification.* aQute Publishing, release 4.2 edition, December 2009.

[PB09] Rajesh K. Panta and Saurabh Bagchi. Hermes: Fast and energy efficient incremental code updates for wireless sensor networks. In *INFOCOM '09: Proceedings of the 28th IEEE conference on Computer Communications*, pages 639–647, April 2009.

[PBRD03] Charles E. Perkins, Elizabeth M. Belding-Royer, and Samir R. Das. *Ad hoc On-Demand Distance Vector (AODV) Routing (IETF RFC 3561).* The Internet Society, July 2003.

[Q6707] Intel. *Intel Core 2 Extreme Quad-Core Processor XQ6000 Sequence and Intel Core 2 Quad Processor Q600 Sequence Datasheet*, August 2007.

[QXD$^+$05] Jiang-Bo Qian, Hong-Bing Xu, Yi-Sheng Dong, Xue-Jun Liu, and Yong-Li Wang. FPGA acceleration window joins over multiple data streams. *Journal of Circuits, Systems, and Computers*, 14(4):813–830, 2005.

[RA07] Jan S. Rellermeyer and Gustavo Alonso. Concierge: a service platform for resource-constrained devices. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 245–258, Lisbon, Portugal, 2007. ACM.

[RAR07] Jan S. Rellermeyer, Gustavo Alonso, and Timothy Roscoe. R-OSGi: distributed applications through software modularization. In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 1–20, Newport Beach, California, 2007. Springer-Verlag New York, Inc.

[RFMB04] Kay Römer, Christian Frank, Pedro J. Marrón, and Christian Becker. Generic role assignment for wireless sensor networks. In *EW 11: Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 2, Leuven, Belgium, 2004. ACM.

[Rot95] Markus F. Rothacher. *Sample-Rate Conversion: Algorithms and VLSI Implementation.* PhD thesis, ETH Zurich, 1995. Diss. Techn. Wiss., Nr. 10980.

[RSS75] Lawrence R. Rabiner, Marvin R. Sambur, and Carolyn E. Schmidt. Applications of a nonlinear smoothing algorithm to speech processing. *IEEE Trans. on Acoustics, Speech and Signal Processing*, 23(6):552–557, December 1975.

[RWR03] Shad Roundy, Paul K. Wright, and Jan M. Rabaey. *Energy Scavenging for Wireless Sensor Networks: with Special Focus on Vibrations.* Springer, first edition, 2003.

[SCAG08] Yunhem Shi, Kevin Casey, Anton M. Ertl Anton, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4(4):1–36, 2008.

[SG08] Ryo Sugihara and Rajesh K. Gupta. Programming models for sensor networks: A survey. *ACM Trans. Sen. Netw.*, 4(2):1–29, 2008.

[SJ04] Andrew Sixsmith and Neil Johnson. A smart sensor to detect the falls of the elderly. *IEEE Pervasive Computing*, 3(2):42–47, 2004.

[SMAL$^+$04] Gyula Simon, Miklós Maróti, Ákos Lédeczi, György Balgoh, Branislav Kusy, András Nádas, Gábor Pap, János Sallai, and Ken Frampton. Sensor network-based countersniper system. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 1–12, Baltimore, MD, USA, November 2004. ACM.

[SMR02] Eung S. Shin, Vincent J. Mooney, and George F. Riley. Round-robin arbiter design and generation. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 243–248, Kyoto, Japan, November 2002. ACM.

[Str] StreamBase Systems. `http://www.streambase.com/`.

[SWT$^+$02] Herman Schmit, David Whelihan, Andrew Tsai, Matthew Moe, Benjamin Levine, and R. Reed Taylor. PipeRench: A virtualized programmable datapath in 0.18 micron technology. In *CICC '02: Proceedings of the 24th IEEE Custom Integrated Circuits Conference*, pages 63–66, Orlando, FL, USA, May 2002.

[SY07] Adam Silberstein and Jun Yang. Many-to-many aggregation for sensor networks. In *ICDE '07: Proceedings of the 23rd International Conference on Data Engineering*, pages 986–995, Istanbul, Turkey, April 2007.

[Syb] Sybase/Aleri. Corel8 engine. `http://www.aleri.com/products/aleri-cep/coral8-engine/overview` (accessed May 15, 2010).

[TcZ$^+$03] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 309–320, Berlin, Germany, September 2003. VLDB Endowment.

[THGT07] Igor Talzi, Anreas Hasler, Stephan Gruber, and Christian Tschudin. Permasense: investigating permafrost with a WSN in the swiss alps. In *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors*, pages 8–12, Cork, Ireland, 2007. ACM.

[TMA10] Jens Teubner, Rene Mueller, and Gustavo Alonso. FPGA acceleration for the frequent item problem. In *ICDE '10: Proceedings of the 26th international Conference on Data Engineering*, pages 669–680, Long Beach, CA, USA, March 2010.

[TPS$^+$05] Gilman Tolle, Joseph Polastre, Robert Szewczyk, David Culler, Neil Turner, Kevin Tu, Stephen Burgess, Todd Dawson, Phil Buonadonna, David Gay, and Wei Hong. A macroscope in the Redwoods. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 51–63, San Diego, California, USA, November 2005. ACM.

[TRL$^+$09] Arvind Thiagarajan, Lenin Ravindranath, Katrina LaCurts, Samuel Madden, Hari Balakrishnan, Sivan Toledo, and Jakob Eriksson. VTrack: accurate, energy-aware road traffic delay estimation using mobile phones. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 85–98, Berkeley, California, 2009. ACM.

[Tru] TruViso. Truvisio continuous analytics. `http://www.truvisio.com/`.

[Tuk77] John W. Tukey. *Exploratory Data Analysis.* Addison-Wesley, 1977.

[TV05] David Tse and Pramond Viswanath. *Fundamentals of Wireless Communication.* Cambridge University Press, first edition, 2005.

[TYD$^+$05] Niki Trigoni, Yong Yao, Alan J. Demers, Johannes Gehrke, and Rajmohan Rajaraman. Multi-query optimization for sensor networks. In *DCOSS '05: Proceedings of the 1st IEEE international conference on Distributed Computing in Sensor Systems*, pages 307–321, Marina del Rey, CA, USA, June 2005. Springer.

[WCG86] Peter D. Wendt, Edward J. Coyle, and Neal C. Gallagher, Jr. Stack filters. *IEEE Trans. on Acoustics, Speech and Signal Processing*, 34(4):898–911, August 1986.

[Wei99] Mark Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, 1999.

[Wir96] Niklaus Wirth. The language Lola, FPGAs and PLDs in teaching digital circuit design. In *Proceedings of the 2nd international Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 2–20. Springer-Verlag, 1996.

[Wir98] Niklaus Wirth. Hardware compilation: Translating programs into circuits. *Computer*, 31(6):25–31, 1998.

[WM04] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *NSDI '04: Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, pages 3–3, San Francisco, California, 2004. USENIX Association.

[WTC03] Alec Woo, Terence Tong, and David Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 14–27, Los Angeles, California, USA, 2003. ACM.

[Xil99] Xilinx Inc. *An Overview of Multiple CAM Designs in Virtex Family Devices. Application Note 201*, September 1999.

[Xil09a] Xilinx Inc. *Virtex-5 FGPA Data Sheet: DC and Switching Characteristics*, v5.0 edition, 2009.

[Xil09b] Xilinx Inc. *Virtex-5 FPGA User Guide*, v4.5 edition, 2009.

[XLTZ07] Shili Xiang, Hock Beng Lim, Kian-Lee Tan, and Yongluan Zhou. Two-tier multiple query optimization for sensor networks. In *ICDCS '07: Proceedings of the 27th international conference on Distributed Computing Systems*, pages 39–47, Toronto, Canada, June 2007. IEEE Computer Society.

[Xtr09] XtremeData, 2009. `http://www.xtremedatainc.com/`.

[YG02] Yong Yao and Johannes Gehrke. The Cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18, 2002.

[YG03] Yong Yao and Johannes Gehrke. Query processing in sensor networks. In *CIDR '03: Proceedings of the 1st biennial conference on Innovative data systems research*, January 2003.

[YRBL06] Yang Yu, Loren J. Rittle, Vartika Bhandari, and Jason B. LeBrun. Supporting concurrent applications in wireless sensor networks. In

SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems, pages 139–152, Boulder, Colorado, USA, 2006. ACM.

[ZR02] Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD '02: Proceedings of the SIGMOD international conference on Management of data*, pages 145–156, Madison, WI, USA, June 2002. ACM.