Profiling for Detecting Performance Anomalies in Concurrent Software

Faheem Ullah
Department of Computer Science
ETH Zurich, Switzerland
ullahf@inf.ethz.ch

Thomas R. Gross
Department of Computer Science
ETH Zurich, Switzerland
trg@inf.ethz.ch

Abstract

Understanding and identifying performance problems is difficult for parallel applications, but is an essential part of software development for parallel systems. In addition to the same problems that exist when analysing sequential programs, software development tools for parallel systems must handle the large number of execution engines (cores) that result in different (possibly non-deterministic) schedules for different executions. Understanding where exactly a concurrent program spends its time (esp. if some aspects of the program paths depend on input data) is the first step towards improving program quality. State-of-the-art profilers, however, aid developers in performance diagnosis by providing hotness information at the level of a class or method (function) and usually report data for just a single program execution.

This paper presents a profiling and analysis technique that consolidates execution information for multiple program executions. Currently, our tool's focus is on execution time (CPU cycles) but other metrics (stall cycles for functional units, cache miss rates, etc) are possible, provided such data can be obtained from the processor's monitoring unit. To detect the location of performance anomalies that are worth addressing, the average amount of time spent inside a code block, along with the statistical range of the minimum and maximum amount of time spent, is taken into account.

The technique identifies performance bottlenecks at the fine-grained level of a basic block. It can indicate the probability of such a performance bottleneck appearing during actual program executions. The technique utilises profiling information across a range of inputs and tries to induce performance bottlenecks by delaying random memory accesses. The approach is evaluated by performing experiments on the data compression tool pbzip2, the multi-threaded download accelerator axel, the open source security scanner Nmap and Apache httpd web server. An experimental evaluation shows the tool to be effective in detecting performance bottlenecks at the level of a basic block. Modifications in the block that is identified by the tool result in performance improvement of over 2.6x in one case, compared to the original version of the program. The performance overhead incurred by the tool is a reasonable 2-7x in majority of the cases.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; D.1.5 [*Programming Techniques*]: Concurrent Programming

Keywords Performance bugs, Profiling, Localisation, Measurement, Dynamic binary instrumentation, Software defects, Parallel programming

1. Introduction

Performance problems are becoming more prevalent in modern software systems [8, 14, 16, 18] as software grows in size. The widespread adoption of multi-core processors and the use of concurrent and multi-threaded programming to utilise these machines have contributed heavily to performance problems. Software correctness and safety constraints in the presence of concurrency may exacerbate the situation even further and lead to performance anomalies in released software. These performance anomalies render software slow and unresponsive. Performance problems may lead to poor user experience, lower system throughput, and a waste of computational resources[4]. Software that is slow may not only frustrate the end users, but also cause financial losses [14]. Following the convention adopted by other researchers in dealing with performance issues[14, 21, 25], we consider an anomaly to be a performance bug if it affects the speed or responsiveness of a program and relatively simple source code changes may lead to significant improvement in performance, while preserving functionality.[1]

SEPS'15, October 27, 2015, Pittsburgh, PA, USA

http://dx.doi.org/10.1145/2837476.2837478

[1] Our goal is to identify possible performance anomalies but possible transformations to address the performance issue are beyond the scope of this

Performance anomalies exist widely in production software. For example, Jin et al.[14] report that Mozilla developers have fixed 5-60 performance bugs reported by users every month over the past 10 years. There are many sources of inefficiency (e.g., poor design, inefficient data structures), and these problems may plague sequential programs as well. Concurrent software is subject to the same issues, but a software developer must in addition deal with unexpected or rare resource contention scenarios involving multiple program threads. The rare resource contention scenarios cause performance anomalies in programs and may only manifest under rare circumstances, such as a unique interleaving of the program threads. Furthermore, as has been observed by others, safety/correctness and performance are two goals for software developers that sometimes exert contradicting pressures. The goal of correctness requires programs to be thread-safe - and as a consequence may require additional synchronization; the goal of fast execution may encourage developers to minimize synchronization in a program.

Performance bugs have only rarely been the focus of research on software defects. There are a number of reasons why performance bugs have received considerably less attention than their functional counterparts. First, developers can check for functional correctness by relying on well established testing techniques for detecting functional bugs[5, 9, 15, 26]. Second, there exist formal approaches to software verification and testing based on symbolic execution that allow (some degree of) automation[6, 7]. Third, most approaches for performance modelling of parallel and distributed systems offer limited support for performance oriented software engineering, or they are capable of dealing with only small programs - as previously pointed out[22]. Lastly, performance bugs are hard to diagnose because of their non-failure semantics i.e., they do not cause failures or program crashes. Furthermore, existing approaches for detecting software performance problems are limited in their functionality and report on resource consumption for a single program execution. The resource consumption of a program is usually dependent on the efficiency of the algorithm being used, and the size of program input. Concurrent programs further complicate matters because of their non-deterministic nature and large number of thread schedules. Different program inputs or thread schedules may lead to different program behaviour. The state-of-art for performance bug diagnosis is the use of profiling tools (e.g., oprof or gprof [12, 17]). However, profiling tools are generally limited in their functionality because they base their reports on a single execution profile of the program being tested. Furthermore, CPU time profilers usually report performance bugs at the level of hot methods/routines. These reported methods can be arbitrarily long. Locating the source of these computational intensive and time-consuming code regions inside a method can prove to be a challenging task in large software systems.

Profiling tools often fail to discover dormant performance problems that may appear under rare circumstance. Similarly, compilers might miss out on optimising a routine that has (real or assumed) side-effects, and identifying the places where cycles are wasted is far from easy. One of the key reason for the lack of better profiling and detection tools for performance bugs is that developers focus on removing functional bugs more often since correctness often outweighs the need for performance, and better tool support is available for functional bugs. In contrast to procedures and tools for detecting functional bugs, tools and procedures for performance bugs are seriously lacking in functionality.

1.1 Contributions

Although current software tools for profiling performance anomalies are useful, they are still severely lacking in functionality. One of the major problem with software profiling techniques is that they provide information only about a specific program run. However, in realistic settings, it is quite likely that different executions of a program with different inputs and/or different thread schedules may lead to different execution profiles of the program. (The program still computes the intended result – otherwise there is a functional bug as well.) The approach presented in this paper detects performance bottlenecks by dynamically collecting profiling information about multiple program executions. The approach aggregates information about multiple profiling executions of the program to detect non-deterministic or rare performance bottlenecks. Rare bottlenecks are those bugs that appear rarely in program executions and require some specific program input and/or environment conditions to manifest. Performance bottlenecks that are deterministic are detected and localised with a single profiling run of the program.

In short the approach presented in this paper makes the following key contributions.

- The approach presented here combines multiple execution profiles of the program to provide a better picture of resource consumption for a wide range of program inputs. The use of multiple execution profiles also enables our approach to report on rare resource contention scenarios that would otherwise be missed. Our profiler does not rely on hotness information for a single input or execution alone. Instead, the profiler uses information about the amount of CPU execution time taken by a basic block for a range of inputs during different executions.
- The approach provides information about performance bugs at a finer-grain level than earlier tools. This is done by identifying the relevant basic block(s) instead of reporting an entire function or class. In certain cases modifying the identified basic block leads to an improvement in program performance of more than 2.6x. (details in Section 4). The approach is not prone to report false pos-

paper. We expect a user to analyse the situation and to modify the source code, employ a different data structure, or to use a different algorithm.

itives since it reports exactly what the user indicates, i.e. basic blocks with a contribution to execution time that exceeds a given threshold (i.e., percentage of the program's total execution time). Of course, not every time-consuming block or set of blocks reported leads to a performance bottleneck that can be fixed; some computations may take a lot of time but this property is due to the problem at hand and the program may already contain the most appropriate algorithm. Our approach attempts to eliminate such false positives by looking at the statistical range of the CPU execution time spent inside a basic block for all executions.

2. Approach and Methodology

The approach to obtaining basic blocks that contribute the most to the overall execution time of a program is divided into two phases. These two phases are concisely summarised in Algorithm 1 and Algorithm 2. The **first phase** of our approach is concerned with on-line profiling of the program under test. In this phase, the program under test is executed multiple times with different inputs[2]. For each execution of the program, our tool records *hotness* information and the amount of CPU execution time taken by every basic block. In the context of our tool, *hotness* of a basic block refers to how often (i.e., frequently) a basic block is executed during a given program execution. The execution frequency of a basic block contributes to its hotness, i.e., the more frequently a basic block is executed, the hotter it becomes. The tool records and maintains the minimum, maximum, and average amount of CPU execution time for each basic block in the program under test. The average time is calculated by maintaining a moving average of the CPU execution time, making it possible to capture the effects of non-deterministic events (e.g., a call to *sleep* for some random amount of time) on a basic block and help eliminate outliers. The diversity of inputs to the program as well as the number of times the program is to be executed is for the user to decide. In this phase the tool tries to influence the scheduler to cause different schedules of the executing program threads. Influencing the scheduler makes it possible for the tool to unearth rare resource contention scenarios that may otherwise go unnoticed during normal testing. Influencing the scheduler is achieved by delaying random shared memory accesses for a small amount of time. At the end of this phase, the profiled information for each execution is written to disk and is available for analysis in the second phase of our approach.

The **second phase** of our approach is outlined in Algorithm 2. This phase performs off-line analysis of the data recorded during the first phase. In this phase, the recorded information is analysed to identify potential *performance bottlenecks*. The bottlenecks correspond to basic blocks in-

[2] Inputs can be supplied by the user or a separate program to generate test input, see the extensive literature on test generation, but this topic is beyond the scope of this paper.

input : PUT - Program under test
output: $profile$ - A log file containing hotness and timing information per basic block

```
1  instrument(PUT) ;
2  while PUT.isAlive do
3      for each BB in PUT do
4          BB.execFreq+=1;
5          execTime=BB.exitTime-BB.entryTime;
6          if BB.execFreq >1 then
7              BB.avgTime= (execTime +
                (BB.avgTime*BB.execFreq))/(BB.execFreq+1);
8              BB.minTime = min(BB.minTime,
                execTime);
9              BB.maxTime = max(BB.maxTime,
                execTime);
10         end
11         else
12             BB.avgTime = BB.minTime=
                BB.maxTime = execTime;
13         end
14     end
15 end
16 WriteProfileLog(PUT);
```

Algorithm 1: Phase 1: On-line profiling of the programs under test.

side the program under test. Performance bottlenecks for our approach are those basic basic blocks where the program spends a *large*[3] portion of its overall execution time. The overall execution time is the total CPU execution time taken by the entire program in a single execution. If the portion of execution time spent in a basic block, and the statistical range of the execution time spent in that basic block, is above a certain threshold (i.e., percentage of the total execution time), then the basic block is reported as a potential performance bottleneck. The threshold value of the total execution time may be specified by the user. Any basic block for which the total execution time and the statistical range of execution time is equal to or above the threshold specified by the user is reported for further inspection. For a basic block to be reported as a performance bottleneck, its significance value as well as range must be higher than the specified threshold. The significance value is a product of the observed average time and the *execution frequency* for a basic block in the recorded program executions (line 21 of Algorithm 2). The statistical range is the difference between the lowest and highest amount of CPU execution time observed in the recorded executions (line 22 of Algorithm 2). Exe-

[3] The programmer decides what is considered to be *large* by specifying a threshold.

```
input : profileLog - A log of profiled data from Alg.1
input : eThreshold - A threshold percentage value of total program execution time
output: pbList - A list of potential performance bottlenecks
1  totalExecTime = readProfile(profileLog) ;
2  eThreshold ← (totalExecTime*eThreshold)/100;
3  bbList ← empty ;
4  while more profileLog exist do
5      read(profileLog);
6      for each BB in profileLog do
7          if bbList.exists(BB) then
8              prevAvg = bbList.BB.avgTime;
9              prevFreq = bbList.BB.execFreq ;
10             prevMin = bbList.BB.minTime;
11             prevMax = bbList.BB.maxTime;
12             bbList.BB.avgTime = max(BB.AvgTime,
                prevAvg);
13             bbList.BB.execFreq = max(BB.execFreq,
                prevFreq);
14             bbList.BB.minTime = min(BB.minTime,
                prevMin);
15             bbList.BB.maxTime = max(BB.maxTime,
                prevMax);
16         end
17         bbList.add(BB);
18     end
19 end
20 for each BB in bbList do
21     bbSig ← BB.execFreq * BB.avgTime ;
22     bbRange ← BB.maxTime - BB.MinTime ;
23     if bbSig >= eThreshold and bbRange >=
        eThreshold then
24         pbList.add(BB);
25     end
26 end
27 Report(pbList);
```

Algorithm 2: Phase 2: Off-line analysis and reporting of performance bottlenecks.

cution frequency refers the number of times a basic block is executed during a single program run.

To perform the pruning of false positives and identify potential performance bottlenecks, our approach chooses the highest recorded values for the execution frequency, the average and maximum amount of execution time, and the lowest recorded value for minimum execution time when aggregating these values from multiple profiled program executions. Selecting recorded values in the aforementioned fashion may be important because non-deterministic execution scenarios involving contention for resources which leads to performance bugs may be of more concern to programmers than just average values.

The final step in the second phase of our approach reports potential performance bottlenecks to programmers. The performance bottlenecks report consists of (for each potential bottleneck) the recorded execution frequencies of the basic block in the profiled program runs, as well as the average, minimum, and maximum amount of CPU execution times recorded for each basic block. The execution frequencies for a basic block signify its hotness across different program inputs as well as whether the basic block is executed in a deterministic/non-deterministic fashion in the recorded program executions. The minimum and maximum CPU execution times show the range of time spent in a basic block for different program inputs, while the average amount of CPU times show the average time spent in a basic block in all the recorded execution of the program. The reported values may be further analysed by using a box plot for the reported values, or the standard deviation for the values may be computed before further investigation.

3. Implementation

The prototype implementation of our approach uses dynamic binary instrumentation using the PIN [13] instrumentation framework to collect run-time information about the execution behaviour of the program. At run-time the tool keeps track of the hotness for each basic block by keeping track of how often the basic block is executed during a single program run. Basic blocks are delineated by conditional branches taken by the program (i.e., executing a conditional jump in the program creates a new basic block). When a new conditional branch is taken, the previous basic block ends. Figure 1 shows the process of collecting dynamic execution profiles of the program under test. The tool keeps track of the amount of CPU time consumed by each basic block during a single execution, as well as the total time consumed by the whole program. For each basic block the tool stores the minimum and maximum amount of CPU time consumed by each basic block, apart from computing and storing a moving average of the time spent in the basic block during a single execution. For each execution of the program with the same or different input, this information is stored permanently and written to a file on disk. Storing information in a permanent location (disk) ensures that profiling information is not lost, since profiling takes a large portion of the procedure. At the end of the profiling phase, the log files containing the recorded information from several executions of the program are available for the analysis phase.

During the analysis phase the tool reads the information from log files containing the profiling information and feeds it into the analysis engine. During this phase occurrence probabilities are calculated for each basic block. The occurrence probability for a basic block indicates the likelihood

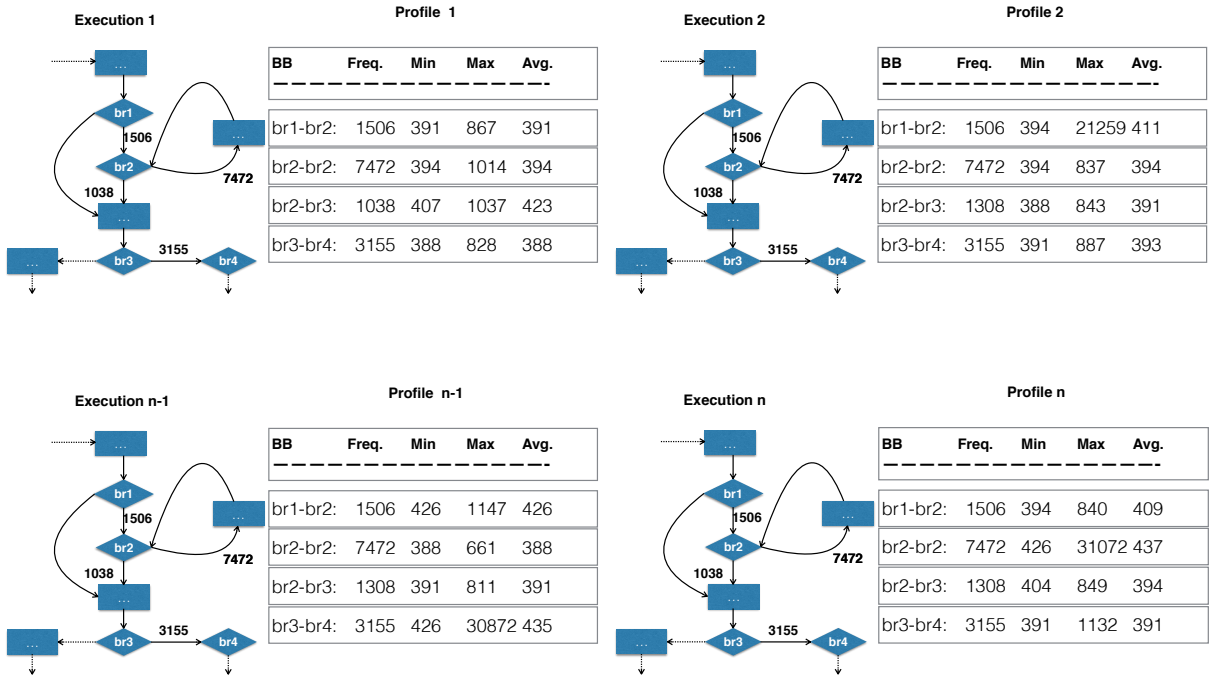

Figure 1: Samples of captured dynamic program profiles (Profiles 1 and 2 as well as (n-1) and n are shown).

for a basic block to be executed during actual program runs. The analysis keeps track of the overall minimum and maximum amount of CPU time, as well as the maximum value for the frequency and the average in all of the recorded executions. Figure 2 graphically depicts the process of selecting these aforementioned values, that are later used to perform pruning of basic blocks that are below the threshold specified by the user.

The analysis phase makes use of a *threshold* value to judge whether a basic block is a potential performance bottleneck. The threshold corresponds to a percentage portion of the total execution time of the program. For example, the programmer might be concerned about a basic block that takes more than a certain portion of the total execution time. For each basic block, the tool compares the product of its hotness and average amount of CPU time as well as the statistical range of the CPU time spent in the basic block with the *threshold* value. If the product and the range are higher than or equal to the threshold value, then the basic block is reported to the programmer, otherwise, it is removed from the list of reported basic blocks. The format for the report containing potential performance bottlenecks that is presented to the user is shown graphically in Figure 3. The analysis utilises the product of maximum average time profiled in all executions of the program, with the maximum frequency for a basic block to ensure that basic blocks consuming a high portion of the program execution time are identified. However, the basic blocks identified us-

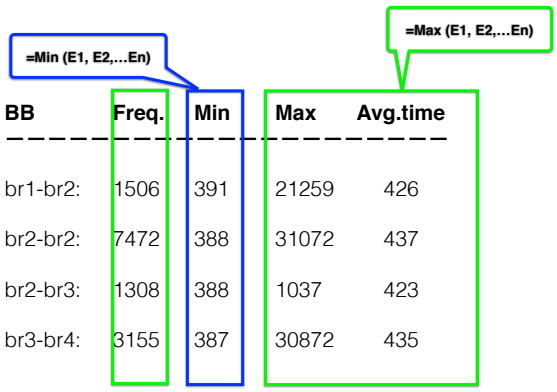

Figure 2: Program profile to prune out basic blocks below the threshold.

ing the product of average time and frequency may very well be a consequence of the algorithm being used by the program. The analysis therefore, utilises the statistical range of the time spent in a basic block in conjunction with the aforementioned product value to ensures that non-deterministic events or contention for resources are accounted for and reported to the programmer.

The reported information is mapped back to the original program so that it may be understood in source language terms and must be inspected by the programmer.

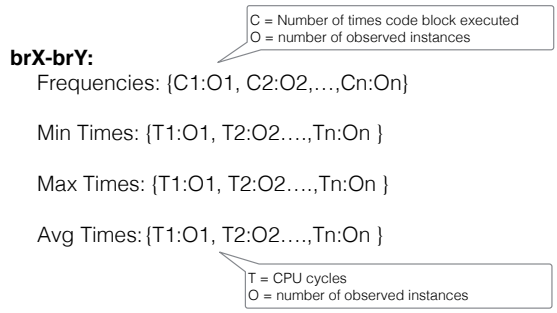

Figure 3: Format of the report for each identified basic block.

4. Evaluation

The tool implementing our approach was applied to different real-world applications written in C/C++. The selected applications include a data compression tool *pbzip2* version 0.9.4 [10], a security scanner for network exploration and audits called *Nmap* version 6.49 [19], and the download accelerator *axel* version 2.3 [1]. The results are presented in Table 1. The first three columns show the results for experiments performed with a single execution profile, while the last two columns show the results for a setup when multiple profiles of the program under test are recorded and analysed.

The experiments for pbzip2 using a single execution profile revealed that it spends around 5% of its total execution time inside of a *for* loop in the block sorting algorithm of the non-parallel bzip2 utility. Bzip2 is the data compression part of the parallel pbzip2 utility. The block sorting algorithm uses Knuth's increments and uses 3 copies before updating the increment. However, the third copy fails to execute most of the time and program execution breaks off back to the main loop to select a new increment. After modifying the bzip2 implementation to remove the third copy, the experiments were performed again with the same input. These experiments revealed a slightly over 0.24% increase in performance with a few inputs. However, this behaviour needs to be investigated further with more inputs before arriving at a solid conclusion. To further validate the technique, an artificial workload was introduced inside of pbzip. The experiments were run again to find out if the newly inserted artificial bottleneck is detected, and whether the tool would point to the exact basic block. The artificial workload consisted of a *while* loop involving some basic arithmetic. The workload added slightly over 5% performance overhead to the original version of the program. The threshold execution portion was set to 5% of the total execution time of the program as before. The bottleneck was correctly identified and localised to the inserted *while* loop using logs from just one execution profile. The bottleneck identified in the original version of the program also appeared in the list of reported bottlenecks as expected. Recording and analysing multiple execution profiles for pbzip2 report another 3 bottlenecks that point to the compression process inside of bzip2.

Experiments with axel[4] reported 4 potential performance bottlenecks. The reported bottlenecks belonged to a *for* loop and the *if* conditionals inside it, for displaying download progress and speed. Since progress is to be displayed and updated continuously, on a per kilobyte basis, it takes a major chunk of the overall program execution. To validate the experiment, the progress and speed reporting routine was changed from per kilobyte to a larger unit, i.e. to per megabyte. The minor changes led to a significant improvement in performance. Although we realise that reporting download progress per megabyte might not be suitable for smaller download sizes, however, for larger files, per kilobyte reporting is probably a nuisance. In our experiments with the modified version of the program, the program was used to download a large file [5]. The average amount of time (normalised over 100 executions) taken by the modified version of the program was compared against the original version. The comparison revealed a performance improvement of 2.6x as compared to the original version of the program. Six new bottlenecks are reported for axel when multiple execution profiles are analysed. All of the six bottlenecks correspond to the main loop inside of axel's *axel_do* method. The reported basic blocks correspond to conditionals that belong to non-deterministic events that deal with making a connection to the server or waiting for one.

Experiments with Nmap report six bottlenecks. The bottlenecks mostly belong to basic blocks inside of library routines for the scripting engine used by Nmap. The code regions belong to library routines for the program language *Lua*, which is used for Nmap's scripting engine. These code regions include a loop performing the hash inside the hashing library function, and conditionals belonging to the String library routines of the Lua programming language/environment. When 100 execution profiles are recorded and analysed, the number of reported potential bottlenecks almost doubles. Inspection of the 5 newly reported possible bottlenecks include a few more basic blocks in the Lua programming language's library routines and two basic blocks belonging to the *traceroute* system utility. Overall, no bottlenecks could be identified in Nmap itself with a threshold of 5%.

The tool was also applied to the latest version of Apache httpd web server (version 2.4.16). Experiments were performed on the startup routine of the web server. Experiments with the startup routine report three bottlenecks at a pruning threshold of 5%. The three reported bottlenecks belong to basic blocks inside a string utility routine used by Apache. The routine is invoked multiple times when reading configu-

[4] The file to be downloaded is served by a local machine to avoid perturbation of the results by wide-area network delays.

[5] The downloaded file a CD image (iso) file containing the Debian operating system and has a size of 627 megabytes.

Table 1: Number of performance bottlenecks and false positives.

Application	Profiles	Bottlenecks	Threshold	Profiles	Bottlenecks	Threshold
pbzip2 - original	01	01	5%	100	04	5%
pbzip2 - workload	01	02	5%	100	05	5%
axel - original	01	04	5%	100	10	5%
Apache httpd start	01	03	5%	100	03	5%
Nmap - original	01	06	5%	100	11	5%

ration settings for the web server from the configuration files on disk. The routine itself probably has little opportunity for any further optimisation, since the routines perform basic string manipulation operations, however, the experiments serve to illustrate the effectiveness of our tool to successfully identify these time-consuming portions of the program.

The experiments performed using our tool serve to validate the effectiveness and usefulness of our approach. The key insight gained from these experiments is that our approach succeeds in identifying locations in the program where it spends a significant portion of its execution time. The locations are identified at a fine-grain level, making it possible to inspect and improve program performance without much effort. For deterministic performance bugs a single execution profile should usually suffice. However, to unearth non-deterministic performance bugs multiple executions of the program with different inputs may be needed. Although our approach makes an effort to exercise different interleavings of the program threads, however, there are no guarantees that this may work in practice. Further investigation in the direction of scheduling is outside the scope of our work and was not explored.

4.1 Performance

The performance overhead of both the on-line and off-line analysis of our approach is presented in Table 2. These experiments were performed on a machine with an Intel core i7 3.50GHz processor, 16 gigabytes of memory, running desktop Ubuntu version 12.04.5. The selected applications are of different scale; ranging from a few thousand lines of code to over 400,000 lines of code in the case for Nmap. The first phase of our approach has a higher performance overhead than the second phase. The reason for the higher overhead is that the first phase is concerned with on-line profiling of the program being tested. The on-line profiler uses dynamic binary instrumentation to collect the required information. Furthermore, the information is collected for every basic block in the program under test. The second phase analyses the profiling data collected in the first phase and is very fast, as can be seen from the fourth column in Table 2. The advantage of our two-phase approach is that once all the data is collected in the first phase, the user is free to conduct more off-line analysis. The data collected in the first phase is never invalidated by the off-line analysis. Since the off-line analysis is fast, the user has the advantage of experimenting with different threshold values during the second phase. To fairly judge the performance of the off-line analysis engine, 100 profiling runs were analysed since they represent a fairly large number of program executions.

The overhead incurred by the on-line analysis is presented in the last column of Table 2. The overhead is normalised over 100 executions to allow a fair analysis of the implementation. The overhead is slightly larger in the case of pbzip2, however, it is orders of magnitude less for larger applications like Apache Httpd and Nmap. There are a couple of reasons for the high overhead in the case for pbzip2. First, since pbzip2 is a smaller application, the set up time required by our framework to load and start up the application adds a significant overhead to the overall execution time. Second, pbzip2 uses of large number of conditionals for its size, and the number of times that these conditionals are executed is fairly high, in part because of its sorting and compression algorithms. Since our profiling engine collects data at each conditional branch, this setup leads to a higher overhead for smaller programs with more conditionals. Experiments using the larger applications including Apache and Nmap serve to validate our claim that the overhead remains reasonable as applications grow larger in size.

4.2 Threats to Validity

There are a number of issues to consider when dealing with performance bugs for concurrent programs. First, concurrent programs are inherently non-deterministic in nature, and the number of possible thread schedules are astronomically large. If the program is not properly synchronized and access is not properly guarded, then different thread schedules may lead to different program behaviour - even when the same program input is supplied. Second, performance bugs in concurrent programs behave the same way as concurrency bugs, i.e., they appear only under rare circumstances. These rare circumstances can be a rare interleaving of the program's threads leading to contention for resources or a particular program input. The approach described here attempts to discover performance bugs from execution profiles that are collected at runtime. If a performance bug does not appear during the recorded program executions, then it will be missed by our approach. The tool presented in this paper tries to influence the scheduler by delaying random memory accesses, however, there are no guarantees that this will cause a rare resource contention scenario to surface.

Table 2: Performance overhead of the on-line profiling and off-line analysis routine.

Application	Profiles	Threshold	Offline Analysis	Online Profiling
pbzip2	100	5%	0.100 seconds	28.71x
Apache Httpd	100	5%	0.591 seconds	3.38x
axel	100	5%	0.0464 seconds	1.97x
Nmap	100	5%	1.07 seconds	7.01x

The technique presented here is not prone to false positives, because it reports events that are above the user specified threshold. However, these events could very well be legitimate program statements and unavoidable for certain programs. The involvement of a threshold value means that our approach relies on the user's intuition of what they perceive to be a performance bug. For example, certain users might not consider a basic block that takes 5% of the total program execution time to be a performance bug. The approach only reports information according to the criteria that is specified by the programmer, and is based on profiling information collected during the actual program executions.

5. Related Work

Performance bugs have largely been ignored in previous research on software defects. Many research efforts on software defects focus on functional rather than performance bugs. A recent empirical study on performance bugs [14] presents a good overview of a wide range of bugs collected from real-world applications and provides a guidance study for performance bugs detection; the authors explore a rule-based performance bug detection technique to uncover previously unknown performance problems.

Most work on detecting performance bugs focuses on identifying code locations that take a long time to execute in certain program executions. Profilers like oprof and gprof [12, 17] periodically sample the program counter during a single execution of the program. The samples are then propagated through the call graph during post-processing to arrive at estimates on how much of the total running time was spent in each function of the program. Such profilers are the standard way to find optimisation opportunities to improve the performance of a program. Recently, profiling techniques involving dynamic analysis to detect excessive memory usage have been proposed. These include techniques for optimising the creation of similar data structures over the lifetime of a program [28], as well as techniques that track the life-time of objects to uncover run-time inefficiencies in programs [29]. Other profiling techniques work on recording memory access behaviour of programs that use recursive data structures [24], by capturing the run-time behaviour of the individual instances of recursive data structures such as lists and trees. More recent work by Nistor et al. [21] present a technique for detecting code loops whose computation has repetitive and partially similar memory-access patterns across loop iterations. Nistor et al. suggest that such repetitive work is likely to be unnecessary and may be performed faster. Mitchel et al. [20] focus on analysing data structures at runtime to find their execution cost. Others track data structures to perform post-mortem analysis of the collected information [27] and suggest improvements to the design of data structures. Pradel et al. [23] suggest a performance regression testing technique for thread-safe classes. The technique works by generating multi-threaded performance tests and comparing two versions of a class to detect performance gains.

Recently, researchers have presented techniques for identifying the root cause of a performance bugs within applications. These include tools like *X-ray* [2]. X-ray is built using techniques of dynamic information flow analysis and deterministic record and replay. X-ray uses performance summarisation techniques to automatically diagnose the root causes of performance bugs. The tool assigns costs to each basic block within the program using certain metrics like CPU utilisation or network activity by using binary instrumentation. The tool then records intervals of program execution to analyse. The execution recorded by X-ray may then be replayed during analysis.

Profiling techniques for detecting performance bugs that utilise multiple program runs include techniques like algorithmic profiling [30]. Algorithmic profiling works by automatically determining an approximate cost function based on multiple program runs. Other techniques [11] focus on calculating empirical computational complexity of programs by running a program with several inputs, and fitting the costs to a curve to arrive at performance as a function of workload size.

Our work combines the approach used by state-of-the-art profilers and those approaches that analyse multiple executions of the program for detecting performance bugs.

6. Extensions

Experimental results of the implementation for our approach revealed certain aspects that would benefit from further investigation in the future. First, the main factor contributing to the high overhead incurred by our approach is the number of conditional branches that are executed in a single program run. The approach relies on collecting profiling information for basic blocks at the boundary of conditional branches in the program under test. Thus, the overhead incurred by our approach is directly proportional to the number of conditional branches that are executed for some program input.

One way to reduce the overhead would be to collect profiling information at a slightly higher granularity than a basic block. The granularity could be that of defined paths through the program, e.g., using ideas from the path profiling algorithm presented in [3], instrumentation does not need to occur at every conditional branch. Implementation at a granularity higher than that of a basic block would significantly reduce the overhead of the first phase in our approach. Furthermore, if traditional profiling information at the class or method level is available, then this information could be used to mask monitoring the execution of those parts of a program that contribute less than the threshold to program execution time.

Second, our approach currently collects profiling information about hotness, along with the minimum, maximum, and average amount of CPU time spent in a basic block. The approach would benefit from collecting more information at these code regions, including information regarding cache misses and stall cycles during the execution.

The current implementation will need to be updated to accommodate these changes. However, since the basic infrastructure is already in place, only a moderate amount of effort will be required to incorporate these changes in the infrastructure.

7. Concluding Remarks

Performance bugs are hard to fix and tools to aid programmers in detecting these bugs are scarce or lacking in functionality. This paper presents an approach for detecting and localising performance bugs. A prototype implementation of the approach is presented to conduct an experimental evaluation of the system. The tool profiles and detects performance problems in applications using profiled execution information. The evaluation of the approach presented in Section 4 shows that the tool indeed detects performance problems in real-world applications at the level of a basic block.

The implementation employs binary rewriting to collect data about a program's execution. Binary instrumentation, although it incurs an overhead, nevertheless allows the use of this technique in realistic settings. The overhead is probably too high to allow the system to be used by default, but the overhead is low enough to allow its use as soon as a problem is suspected.

The decision to use basic blocks bounded by conditional branches in the prototype implementation is based on the fact that current support for monitoring the execution behaviour of programs is limited, especially for multi-threaded programs. Other options (e.g., to replace binary instrumentation by a compiler framework to keep track of the conditional branches that are executed and to directly use the processor's program monitoring unit) are possible and worthy of further investigation. However, the difficulties to obtain valuable data about a program's execution point to the need for better program monitoring units. Given the proliferation of multi-core processors, and the ability to provide additional functionality in processor implementations, architects of future processors should pay more attention to the needs of tools that help software engineers isolate and understand program performance defects.

References

[1] Axel download accelerator project.

[2] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 307–320, Berkeley, CA, USA, 2012. USENIX Association.

[3] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[4] Randal E Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. *Computer systems: a programmer's perspective*, volume 2. Prentice Hall Upper Saddle River, 2003.

[5] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 167–178, New York, NY, USA, 2010. ACM.

[6] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1066–1071, New York, NY, USA, 2011. ACM.

[7] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

[8] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 607–622, New York, NY, USA, 2015. ACM.

[9] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[10] J Gilchrist. Parallel bzip2 (pbzip2) data compression software. *URL http://compression. ca/pbzip2*.

[11] Simon F. Goldsmith, Alex S. Aiken, and Daniel S. Wilkerson. Measuring empirical computational complexity. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 395–404, New York, NY, USA, 2007. ACM.

[12] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM.

[13] Intel. Pin - a dynamic binary instrumentation tool.

[14] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.

[15] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 110–120, New York, NY, USA, 2009. ACM.

[16] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. Finding latent performance bugs in systems implementations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 17–26, New York, NY, USA, 2010. ACM.

[17] John Levon, Philippe Elie, et al. Oprofile, a system-wide profiler for linux systems. *Homepage: http://oprofile. sourceforge. net*, 2008.

[18] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1013–1024, New York, NY, USA, 2014. ACM.

[19] Gordon Lyon. Nmap–free security scanner for network exploration & security audits, 2009.

[20] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. Making sense of large heaps. In Sophia Drossopoulou, editor, *ECOOP 2009 Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 77–97. Springer Berlin Heidelberg, 2009.

[21] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 562–571, Piscataway, NJ, USA, 2013. IEEE Press.

[22] Sabri Pllana, Ivona Br, and Siegfried Benkner. A survey of the state of the art in performance modeling and prediction of parallel and distributed computing systems. In *Euro-Par 2004 Parallel Processing. Springer LNCS 3149:183188*. Springer, 2004.

[23] Michael Pradel, Markus Huggler, and Thomas R. Gross. Performance regression testing of concurrent classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 13–25, New York, NY, USA, 2014. ACM.

[24] Easwaran Raman and David I. August. Recursive data structure profiling. In *Proceedings of the 2005 Workshop on Memory System Performance*, MSP '05, pages 5–14, New York, NY, USA, 2005. ACM.

[25] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 561–578, New York, NY, USA, 2014. ACM.

[26] Wenwen Wang, Zhenjiang Wang, Chenggang Wu, Pen-Chung Yew, Xipeng Shen, Xiang Yuan, Jianjun Li, Xiaobing Feng, and Yong Guan. Localization of concurrency bugs using shared memory access pairs. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 611–622, New York, NY, USA, 2014. ACM.

[27] Xiao Xiao, Jinguo Zhou, and Charles Zhang. Tracking data structures for postmortem analysis (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 896–899, New York, NY, USA, 2011. ACM.

[28] Guoqing Xu. Finding reusable data structures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 1017–1034, New York, NY, USA, 2012. ACM.

[29] Dacong Yan, Guoqing Xu, and Atanas Rountev. Uncovering performance problems in java applications with reference propagation profiling. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 134–144, Piscataway, NJ, USA, 2012. IEEE Press.

[30] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 67–76, New York, NY, USA, 2012. ACM.