Hardware Support for Production Run Diagnosis of Performance Bugs

Abdullah Muzahid
Department of Computer Science
University of Texas at San Antonio
San Antonio, TX 78249
Email: abdullah.muzahid@utsa.edu

Abstract—Performance bugs cannot be easily debugged in the same way correctness bugs are debugged. They are debugged mostly by analyzing execution profiles which is slow, tedious, and heavily involved. As a result, even for a mature program, performance bugs often slip into production systems. This paper presents a hardware based approach, called Prometheus, that detects loop related performance bugs during production runs with negligible overhead. Prometheus works by detecting redundant memory read accesses in loop iterations. If many loop iterations access the same set of memory locations and the same set of values, then Prometheus reports a performance bug. Prometheus detects the redundant accesses in hardware using bloom filter based signatures. Prometheus is automatic and does not require a programmer to analyze large execution profiles. Moreover, Prometheus is parameterized to achieve different levels of accuracy and detection ability. Prometheus is the first hardware based scheme for automatically detecting performance bugs related to redundant accesses. This paper presents a detailed design and implementation of Prometheus hardware. Prometheus is evaluated on a variety of real world performance bugs. It detects 8 out of 10 performance bugs. Once the bugs are fixed, Prometheus does not falsely detect any bug except in one case. It has a negligible execution overhead of 1.87%. Last but not the least, Prometheus requires only \( \approx 1 \) Kbyte of extra hardware structures.

I. INTRODUCTION

A performance bug is a programming error that causes significant performance degradation [2]. Performance bugs can lead to increased latency, reduced throughput, and wasted resources (e.g., energy, memory etc.) of a computer system. A program can have low performance because of the high complexity of the algorithm, high I/O intensity, wrong choice of data structures, redundant memory accesses and computations, slow hardware structures, over use of synchronizations, overloading of the system etc. Although many of these factors can be eliminated by making changes in the program, others cannot be eliminated without changing system infrastructure substantially. Therefore, in the context of this paper, the factors that can be eliminated/reduced by making changes in the program are referred to as Performance Bugs.

Unlike correctness bugs, performance bugs often get little or no attention of the programmers. Therefore, even mature programs have many performance bugs escaped into the production systems. As an example, Windows 7’s Internet Explorer has several high impact performance bugs that escaped into production systems and remain undiscovered for a long period of time [6]. Performance bugs can slip into production systems due to several reasons. First, well known software testing techniques based on test cases are not suitable for performance bugs. Second, a conventional profiling based approach to pinpoint performance bugs is quite slow and tedious. Third, performance bugs are often evident when a program needs to process huge amount of data/inputs. This coupled with the fact that performance bug debugging requires a programmer to analyze execution profiles mostly manually, often discourages him/her to spend much of the debugging effort on this issue. Finally, due to a strict deadline of product delivery, a programmer is often satisfied with good-enough performance in testing and development environments. Such practices can lead to performance bugs even in mature programs, which get exposed when those programs run in high stress production environments.

Detecting performance bugs during production runs warrants a mechanism that has several features. First, it should automatically detect performance bugs without requiring a programmer to analyze profiles. Second, it should not impose significant execution overhead. Finally, it should detect performance bugs with high accuracy. Toward this end, we propose Prometheus. Prometheus is a hardware based approach to detect performance bugs automatically during production runs with negligible execution overhead. Since 90% of the performance bugs are related to loops [10], Prometheus focuses on loop related performance bugs. At the high level, Prometheus works by detecting redundant memory read accesses in loop iterations. If a significant fraction of loop iterations reads the same set of memory locations and the same set of values, then Prometheus reports a performance bug. The rationale is that if some memory locations are read multiple times and the same set of values are returned, they are redundant accesses and can be optimized to improve overall performance. Prometheus detects these redundant accesses in hardware using bloom filter based signatures [3]. As a result, Prometheus has a very low execution overhead and can be used during production runs. Prometheus does not require a programmer to analyze large execution profiles. Moreover, Prometheus is parameterized. Therefore, it can be tuned to achieve different levels of accuracy and detection ability. Prometheus is the first hardware based scheme for automatically detecting performance bugs related to redundant accesses. This paper presents a detailed design and implementation of Prometheus hardware. We implement it in a cycle accurate execution driven simulator. We evaluate Prometheus on a variety of real world performance bugs. It detects 8 out of 10 performance bugs. Once the bugs are fixed, Prometheus does not falsely detect any bug except in one case. It has a negligible execution overhead of 1.87% which makes it suitable for production run diagnosis. It adds only \( \approx 1 \) Kbyte of extra hardware structures.
II. BACKGROUND

a) Performance Bug Detection: There has been significant research on profiling based performance diagnosis. TraceAnalyzer [5] provides a general framework to compose different filter functions to analyze traces. Hauswirth et al. [7] propose a technique that collects performance and behavioral data from all components of a system (e.g., application, virtual machine, and hardware) and then uses statistical and visualization techniques to understand the overall system performance. There are several commercial and open source tools (e.g., Intel vTune, DCPI/ProfileMe [4] etc.) that use hardware performance counters to profile an application’s performance. Nistor et al. [10] propose a technique, called Toddler, that detects loop related performance bugs. A programmer writes many test cases and the tool finds the test cases where many loop iterations access the same sequence of memory locations and read the same sequence of values. Although Prometheus bears some similarities with Toddler, there are some major differences. Toddler is designed to be used in testing environment whereas Prometheus is designed to be used on-the-fly during production runs. In other words, Prometheus targets hard-to-detect performance bugs that escape into production systems.

b) Bloom Filter Based Signature: A signature is a long hardware register (e.g., 2Kbits long) based on Bloom Filter [1]. Signatures have been used in the Bulk system [3] to dynamically disambiguate groups of addresses accessed by different processors.

Fig. 1: (a), (b) show insertion and membership checking operation on a single signature, (c) shows a parallel signature.

Typically, signatures are used to accumulate addresses accessed by a processor. A signature contains a bit array (Figure 1(a) and (b)). Figure 1(a) shows the insertion of an address into a signature. As an address is generated by a processor, a hash function (e.g., H) is applied to the address. According to the output of the hash function, one of the bits in the array is set using logical OR operation. Figure 1(b) shows the checking of an address in a signature. After applying the hash function, the corresponding bit in the bit array is checked (using logical AND operation) to determine whether the bit is already set. If the bit is already set, the address is considered to be present in the signature. Operations on signatures may produce false positives (i.e. addresses not inserted into a signature might be found there), although not false negatives.

III. PROMETHEUS: DETECTING PERFORMANCE BUGS

A. Overview

At the high level, Prometheus works by calculating redundant memory read accesses in each iteration. A memory read access is considered to be redundant in an iteration if the same location is read by an earlier iteration and the same data is returned. If a significant fraction of iterations performs redundant read accesses most of the time, then Prometheus identifies the loop as having a performance bug.

B. Definitions

A loop iteration that performs at least one memory read access is called a Data Iteration (DI). A data iteration that performs at least Access Threshold (AT) number of memory read accesses is called a Data Intensive Iteration (DII). Here, AT is a tunable parameter set by a programmer. If the ratio of redundant and total memory read accesses of a data intensive iteration is greater than or equal to Redundancy Threshold (RT), then the iteration is called a Redundant Iteration (RI). Here, RT is another tunable parameter set by a programmer.

C. Detailed Algorithm

The start and end of a loop as well as its iterations are annotated with special instructions. Prometheus maintains a set, called Read Set (RS), that contains the addresses as well as data values of all the read accesses performed inside the loop. In other words, each element of RS is a pair (A, D) where A denotes the memory address of a read access and D denotes the actual data returned by the read. The detailed algorithm is shown in Figure 2. Prometheus associates two counters with a loop - Data Iteration Counter (DIC) and Redundant Iteration Counter (RIC). When a loop starts, Prometheus clears RS, DIC, and RIC. Every time an iteration starts, Prometheus initializes two counters - Total Read Counter (TRC) and Redundant Read Counter (RRC). TRC is incremented every time a memory read access is performed. If the read accesses address A and returns data D, then (A, D) is checked against the contents of RS. If RS already has an element (A’, D’) such that A’ = A and D’ = D, then this read is identified as a redundant read and RRC is incremented. It should be noted that we allow RS to contain multiple elements whose addresses are the same (e.g., A) but the data values are different (e.g., D’, D”, D’”, ... etc.). If the newly performed read causes a match with one of these elements, then Prometheus identifies the read access as redundant. However, if no match is found in RS, a new element corresponding to (A, D) is inserted into RS. This ensures that if a future iteration has a read access that reads the same location and value, that access will be identified as a redundant one.

At the end of an iteration, if TRC is greater than zero, then the iteration is identified as a data iteration (DI, according to Definition 1). DIC is incremented in such a case. In addition, if TRC is greater then AT, then this iteration is a data intensive iteration (DII, according to Definition 2). In that case, the ratio of RRC and TRC is compared against RT. If the ratio is at least as large as RT, then a significant number of read accesses of this iteration is redundant. So, the iteration is identified as a redundant iteration (RI, according to Definition 3) and RIC is incremented. At the end of a loop, the ratio of RIC and DIC is calculated and compared against a threshold, called Redundant Iteration Threshold (RT). If the ratio is at least as large as RT, then the loop has a significant fraction of redundant iterations and is therefore, identified as having a performance bug. Note that RT is another programmer tunable parameter for Prometheus. In case of nested loops, when the outer loop starts, all annotations of inner loops are
ignore, essentially treating all memory read accesses of inner loops as part of a single iteration of the outer loop.

![Diagram](a)

**Fig. 2: Detailed algorithm of Prometheus.**

**IV. IMPLEMENTATION**

Each element of RS is a pair (A, D) where A is the address and D is the data of a memory read operation. When a load is about to retire from the reorder buffer, a processor concatenates the address and the corresponding data and sends it to RS. We use signature [12] to implement RS. One signature is used to store addresses (Address Signature or AS) and another is used to store data values (Data Signature or DS). To check whether an element (A, D) is already present in RS, A is checked against AS and D is checked against DS. If any one of the intersections is null, then (A, D) is not an element of RS. We use n physical ASs to implement one logical AS. Similar is for DS. A processor pipeline is equipped with a module called, Prometheus Module (Figure 2(b)). The module contains four counters - Data Iteration Counter (DIC), Redundant Iteration Counter (RIC), Total Read Counter (TRC), and Redundant Read Counter (RRC). The module also contains three floating point registers - Access Threshold Register (athr), Redundancy Threshold Register (rthr), and Redundant Iteration Threshold Register (rithr). They contain tunable parameters of Prometheus. In addition, the module contains RS and the associated Signature Functional Unit. The module contains a flag, called Loop Entered (LE) and a register, called Loop Id (lid). LE is used to indicate whether the execution is inside a loop. The register lid stores information to identify a loop.

Prometheus adds several new instructions in the ISA. They are listed in Table I. This instructions are inserted automatically during compilation time by a compiler.

**V. EVALUATION**

**Experimental Setup:** We model Prometheus’s architecture using a PIN [8] based cycle-accurate execution-driven simulator [11]. We model an out-of-order core. The cache hierarchy consists of a write through L1 cache and a write back L2 cache. Table II shows the architectural parameters. We use two sets of applications for evaluation. The first set has 10 real world performance bugs from Apache Common Collections Library, Google Core Library, and Ant Build Tool shown in Table III. We number the bugs in order according to the table. 7 SPEC applications are also used to evaluate execution overhead.

**TABLE I:** New instructions added by Prometheus. [.] is used to denote the content of a register.

**TABLE II:** Parameters with default being boldfaced.

**Characterization of Loop Iterations:** We run test programs that exercise the bugs. We count the number of memory read accesses in each loop iteration. Figure 3(a) shows what fraction of total iterations of buggy programs perform certain number of read accesses. Except for bug 1 and 9, other applications have a significant fraction of iterations with at least 1 read access. On average, 30% iterations have at least 1 read access. In reality, almost all of those iterations have at least 5 read accesses. Moreover, 23%, 22%, and 12% of all iterations have at least 10, 25, and 50 read accesses respectively. We also count the number of redundant read accesses in each iteration. Figure 3(b) shows what fraction of iterations have certain percentage of redundant read accesses. Except for bug 10, more than 80% iterations have at least 50% redundant read accesses. On average, 84% iterations of buggy programs have at least 50% redundant accesses. Almost all of those iterations have at least 90% redundant read accesses. We choose redundancy threshold (R_{TH}) to be 90% i.e. if an iteration has 90% redundant read accesses, it will be a redundant iteration.

**Sensitivity to Signature:** We experiment with different sizes of signatures. We keep the total number of signatures fixed at 8 (for each type). We vary the size of signature from 512 bit to 2048 bit. The result is shown in Figure 3(c). It shows what fraction of total iterations of buggy programs are redundant. On average, 512 bit signatures cause 84% iterations.
to be identified as redundant. For 1024 and 2048 bit signatures, the number is 81% and 79% respectively. Since the impact of larger signatures is not very significant, we choose 512 bit signatures for Prometheus. On a related note, the data also justifies our choice of redundant iteration threshold \((RI_{TH})\) to be 80% for the default signatures. This implies that with this threshold, Prometheus will be able to detect most of the performance bugs when the fraction of redundant iterations exceeds this threshold.

**Bug Detection:** We experiment with buggy programs to determine how many bugs are detected. In addition, we experiment with fixed programs to determine whether Prometheus falsely identifies any of them as buggy. Prometheus detects 8 out of 10 bugs. It does not detect bug 7 and 10. Bug 7, although not detected with our default value of \((RI_{TH})\) (i.e. 80%), can be easily detected if the threshold is set to a slightly lower value. If a programmer can afford to spend more time on debugging, then (s)he should choose a lower threshold to detect all potential bugs. Otherwise, (s)he should choose a higher threshold to detect only the true bugs. Bug 10, on the other hand, has only 17% redundant iterations and therefore, is not detected. Our test program for this bug failed to generate redundant read accesses. This demonstrates the importance of having hardware support for production run diagnosis.

![Fig. 3: (a) & (b) shows distribution of read accesses among iterations. (c) shows redundant iterations for different signatures.](image)

<table>
<thead>
<tr>
<th>Id</th>
<th>Huggy Prg</th>
<th>Fixed Prg</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Red. Iter. (%)</td>
<td>Bug Det</td>
</tr>
<tr>
<td>1</td>
<td>24</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>89</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>99</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>67</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>94</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>97</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>97</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>96</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>77</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**TABLE IV:** Detection of bugs. Fixed version of bug 9 does not have any loop.

For the fixed programs, Prometheus does not falsely report any bug except for bug 8. This bug fix has some redundant read accesses but eliminating them would not improve performance significantly. So, the programmer chose to keep it that way.

**Overhead:** We use 7 SPEC benchmarks for determining execution overhead. The overhead is mainly due to additional loop related instructions. Prometheus incurs overhead from 0.71% to 3.72%, with an average of 1.87%. This overhead is low enough for production run diagnosis. Prometheus uses 8 address signatures and 8 data signatures. Each signature is 512 bit long. In addition to In total, Prometheus requires 1056 byte \(\approx 1\) Kbyte of extra hardware.

VI. CONCLUSION

This paper proposed Prometheus, the first hardware based scheme for automatically detecting performance bugs related to redundant accesses. Prometheus works by detecting redundant memory read accesses in loop iterations. If a significant fraction of loop iterations reads the same set of memory locations and the same set of values, then Prometheus reports a performance bug. Prometheus detects the redundant read accesses in hardware by using bloom filter based signatures. Prometheus is automatic, does not require a programmer to analyze large execution profiles and can be tuned to achieve different levels of accuracy and detection ability. We evaluated Prometheus on a variety of 10 real world performance bugs. It detects 8 out of 10 bugs.

ACKNOWLEDGMENT

The author is supported in part by NSF Grant CCF-1319983.

REFERENCES