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 (≈)1 Kbyte of extra hardware structures.**

I. Introduction

A performance bug is a programming error that causes significant performance degradation [6]. 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 [13]. 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 [23], 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 [7]. 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 (≈)1 Kbyte of extra hardware structures.

II. Background

a) Performance Bug Detection: There has been significant research on profiling based performance diagnosis. TraceAnalyzer [11] provides a general framework to compose different filter functions to analyze traces. Hauswirth et al. [14] 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 [8], DCPI/ProfileMe [9] etc.) that use hardware performance counters to profile an application's performance.

There has been some notable progress in detecting performance bugs without any profiles. Jin et al. [16] propose to extract efficiency rules form patches of known performance bugs and use those rules to Nistor et al. [23] 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. Therefore, Prometheus needs to rely on special hardware structures to make it suitable for production run deployment. Finally, unlike Toddler, Prometheus detects a performance bug even if the memory locations are accessed out of sequence.

b) Bloom Filter Based Signature: A signature is a long hardware register (e.g., 2Kbits long) based on Bloom Filter [5]. Signatures have been used in the Bulk system [7] to dynamically disambiguate groups of addresses accessed by different processors. Signatures have been shown to be useful in the context of various hardware based bug detection proposals [20].

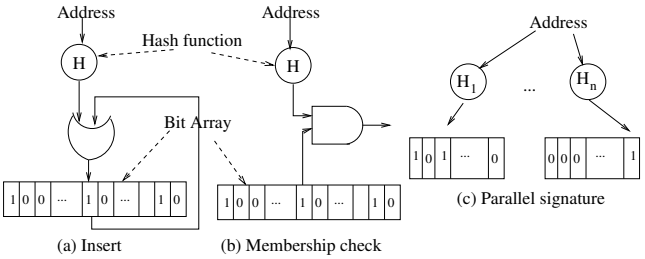


Fig. 1: (a), (b) show insertion and membership checking operation on a single signature, (c) shows a parallel signature.

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. A good signature design is a a parallel one (Figure 1(c)), where multiple hash functions operate independently [25].

III. Prometheus: Detecting Performance Bugs

A. An Example

Let us consider a simplified code fragment from Apache Common Collections Bug 406 [3]. This is referred to as Bug 1 in Section V.

```
List<E> subtract(List<E> list1,
                 List<E> list2) {
  ArrayList<E> result =
            new ArrayList<E>(list1);
  for(E e: list2) result.remove(e);
  return result;
}
```

The `subtract` function takes two `List` objects (`list1` and `list2`), creates a new object `result`, populates it with the elements of `list1` that are not present in `list2`. Each iteration of the `for` loop takes one element of `list2` and removes it from `result` which was initialized with the elements of `list1`. The `remove` function is shown below.

```
boolean remove(Object o) {
  for(int index=0; index<size; index++)
    if(o.equals(elementData[index])
     // remove the element and return
}
```

The `remove` function reads each element of `result`. When the desired element is found, it is removed from `result`. Thus, for each element of `list2`, the same elements of `result` are read starting from the beginning. Therefore, the `for` loop wastes time by doing redundant memory read accesses.

B. 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.

C. 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 (A_{TH}) number of memory read accesses is called a Data Intensive Iteration (DII). Here, A_{TH} 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 (R_{TH}), then the iteration is called a Redundant Iteration (RI). Here, R_{TH} is another tunable parameter set by a programmer.

D. 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"', 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 A_{TH}, 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 R_{TH}. If the ratio is at least as large as R_{TH}, 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 (RI_{TH}). If the ratio is at least as large as RI_{TH}, then the loop has a significant fraction of redundant iterations and is therefore, identified as having a performance bug. Note that RI_{TH} is another programmer tunable parameter for Prometheus. In case of nested loops, when the outer loop starts, all annotations of inner loops are ignored, essentially treating all memory read accesses of inner loops as part of a single iteration of the outer loop.

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 [25] 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.

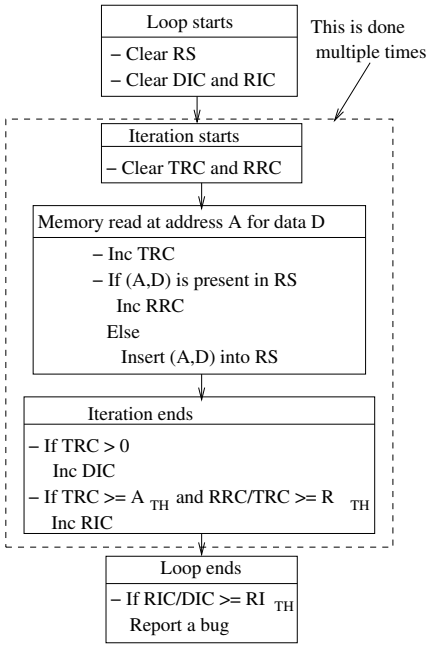


Fig. 2: Detailed algorithm of Prometheus.

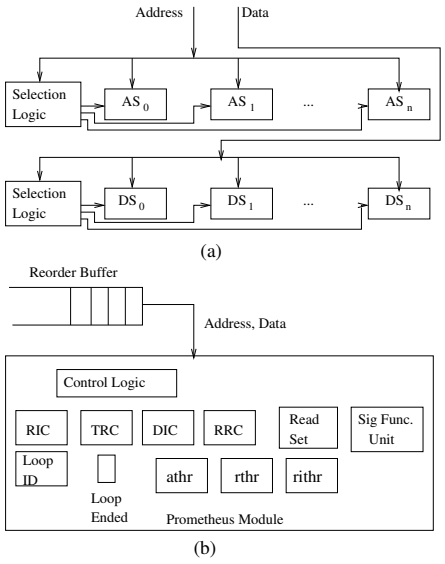


Fig. 3: Implementation of (a) Read Set and (b) Prometheus Module.

A processor pipeline is equipped with a module called, Prometheus Module (Figure 3(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 PrometheusIn 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. Finally, the

module contains some Control Logic.

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.

Instruction	Description
lpstart	Save instruction address of *lpstart* in *lid*. Clear DIC, RIC, and RS. Set LE flag. Clear RRC and TRC.
itrend	If TRC$\geq$0, inc DIC. If TRC$\geq [athr]$ and RRC/TRC$\geq [rthr]$, inc RIC. Clear RRC and TRC.
lpend	If RIC/DIC$\geq [rithr]$, log bug at loop [*lid*]. Clear *lid* and LE.
cple	Copy LE flag to Zero Flag.
setle	Set LE flag.
clrle	Clear LE flag.

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

V. Evaluation

A. Experimental Setup

We model Prometheus's architecture using a PIN [17] based cycle-accurate execution-driven simulator [24]. 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. When we have a choice, we use the boldfaced values as defaults.

Core pipeline	Out-of-order; 2-issue/2-retire.
ROB size	64 entries.
Load buffer size	32 entries.
Write buffer size	32 entries.
Private L1 cache	32KB WT, 4-way, 6-cycle round trip.
L1 MSHR size	64 entries.
L2 cache	512KB WB, 8-way, 14-cycle round trip.
L2 MSHR size	64 entries.
Cache line size	32B.
Memory	300 cycle round trip.
Signature	**512**, 1024 or 2048 bit using H3 hash [20].
Total	4, **8** or 16 of each type.
Thresholds	A_{TH}=10, R_{TH}=90%, RI_{TH}=80%

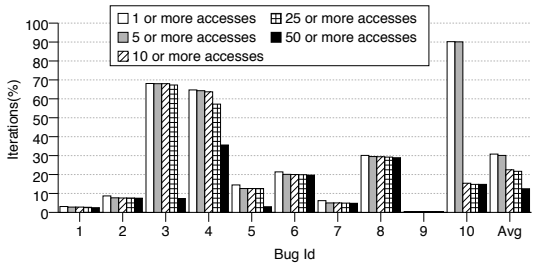
TABLE II: Architectural parameters evaluated.

B. Benchmarks

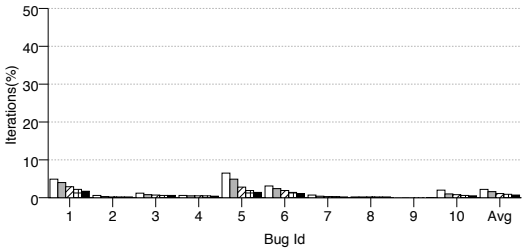
We use two sets of applications for evaluation. The first set has 10 real world performance bugs from Apache Common Collections Library [4], Google Core Library [12], and Ant Build Tool [2]. For each of these, we write a test program to exercise the bugs. Table III gives a description of the bugs. The other set has 7 SPEC benchmarks. We use them to determine execution overhead of Prometheus.

C. Characterization of Loop Iterations

We run programs (mentioned in Table III) with performance bugs. We also run the same programs with developers' patches applied to fix the performance bugs. We refer to the former set as *Buggy Programs* and the later set as *Fixed Programs*. The patch for Bug 9 requires a significant rewriting of code which ends up removing the loop. Therefore, we exclude it from fixed programs. We count the number of memory read accesses in each loop iteration. Figure 4(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. On the other hand, fixed programs, shown in Figure 4(b), have very few iterations with at least 1 read access. On average, 2%, 1%, 0.9%, and 0.7% iterations have at least 1, 5, 10, 25, and 50 read accesses respectively. We also count the number of redundant read accesses in each iteration. Figure 5(a) and 5(b) shows what fraction of iterations have certain percentage of redundant read accesses. For buggy programs, 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. For fixed programs, on average, 58% iterations have at least 50% redundant read accesses. For these programs, 54% 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.



(a) Read accesses for iterations of buggy programs.



(b) Read accesses for iterations of fixed programs.

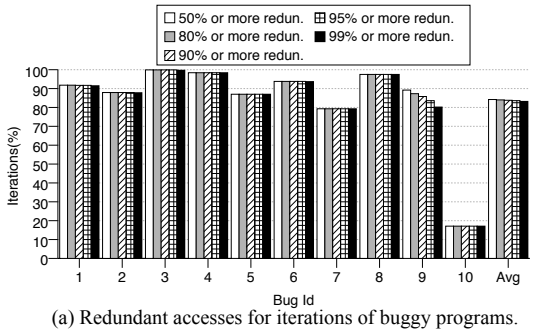
Fig. 4: Distribution of read accesses among iterations.

D. Sensitivity to Signature

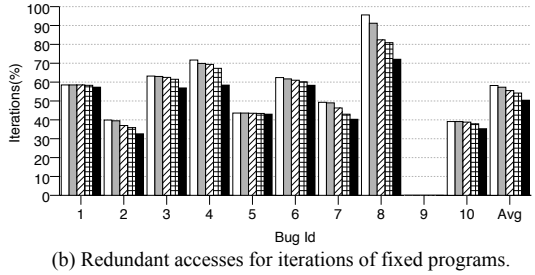
We experiment with different size and number of signatures. First, we keep the total number of signatures fixed at 8 (for each type). We vary the size of signature from 512 bit to 2048 bit. The result is shown in Figure 6(a). It shows what fraction of total iterations of buggy programs are redundant. Note that if a signature has a high false positive rate, then more read accesses will be found in the read set (Section **??**). As a result, more read accesses will be considered as redundant and more iterations will be identified as redundant iterations. Figure 6(a) shows that as signature size increases, relatively fewer iterations are identified as redundant. This due

Bug #	Program	Related Class	Description
1	Apache Com. Col.	ListUtils	*subtract* function reads one element of *list1* and compares against each element of *list2* in every iteration.
2	Apache Com. Col.	ListOrderedSet	*removeAll* function iterates over all elements of *coll* and can cause unnecessary scan of *setOrder*'s elements.
3	Apache Com. Col.	ListOrderedSet	*addAll* function iterates over all elements of *coll* and inserts them in an *ArrayList*. This causes repeated read of the elements of the list.
4	Apache Com. Col.	SetUniqueList	*addAll* function iterates over all elements of *coll* and inserts them in a *LinkedList*. This causes repeated read of the elements of the list.
5	Apache Com. Col.	CollectionUtils	*subtract* function reads one element of an *Iterable* and compares against all elements of an *ArrayList*. This is done in every iteration.
6	Apache Com. Col.	DualHashBidiMap	For each element of a *View* object, *removeAll* function checks against the elements of a *Collection* object. For a list type *Collection* object, the checking causes repeated read of every element.
7	Apache Com. Col.	AbstractLinked-List	For each element of the list, *removeAll* function checks against the elements of a *Collection* object. For a list type *Collection* object, the checking causes repeated read of every element.
8	Apache Com. Col.	ListOrderedMap	*remove* checks against every element of *ArrayList* even when not required.
9	Google Core Lib.	LinkedHashMulti-Map	For smaller map sizes, *removeAll* function of *AbstractSet* is slow because of the repeated read of the set elements.
10	Ant Build Tool	VectorSet	For each element of a *VectorSet*, *retainAll* function searches every element of a *Collection* object causing redundant read operations.

TABLE III: Bug descriptions.

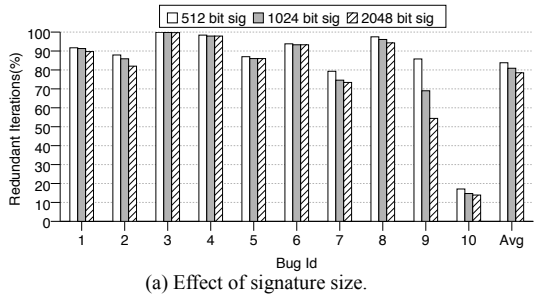


(a) Redundant accesses for iterations of buggy programs.

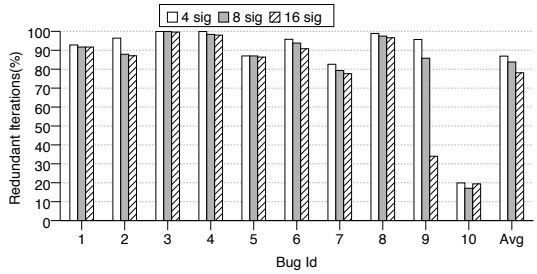


(b) Redundant accesses for iterations of fixed programs.

Fig. 5: Distribution of redundant read accesses.



(a) Effect of signature size.



(b) Redundant accesses for iterations of fixed programs.

Fig. 6: Impact of size and number of signatures.

to less false positives of larger signatures. 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. Figure 6(b) shows the fraction of redundant iterations for different number of total signatures. When we use 4 signatures, on average, 87% iterations are identified as redundant. For 8 and 16 signatures, the number is 84% and 78% respectively. For Prometheus, we use a total of 8 signatures to keep the hardware requirement low while maintaining acceptable accuracy. 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.

E. Bug Detection

The most important metric is the ability to detect performance bugs. Table IV shows the data. 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. We compare against a recent software based performance bug detection scheme, Toddler [23]. For Toddler, we only run the buggy programs. Prometheus detects 8 out of 10 bugs. It does not detect bug 7 and 10. Recall that a bug is detected if the fraction of redundant iterations is greater than or equal to RI_{TH}. 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. This clarifies the importance of choosing an appropriate threshold 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. Of course,

lower threshold might report some bugs that end up being false bugs. On the other hand, if a programmer is under tight timing constraint, (s)he should choose a higher threshold to detect only the true bugs. But, such a higher threshold might end up missing some real bugs. Bug 10, on the other hand, have only 17% redundant iterations. Therefore, even with a lower threshold, this bug cannot be detected. This is due to the fact that our test program for this bug failed to generate redundant read accesses. This demonstrates the importance of having hardware support for production run diagnosis so that a bug can be detected as soon as it occurs.

Id	Buggy Prog.		Fixed Prog.		Toddler
	Red. Iter. (%)	Bug Det?	Red. Iter. (%)	Bug Det?	Bug Det?
1	92	Yes	58	No	Yes
2	89	Yes	37	No	Yes
3	99	Yes	62	No	Yes
4	98	Yes	69	No	Yes
5	87	Yes	44	No	Yes
6	94	Yes	61	No	Yes
7	79	No	46	No	Yes
8	97	Yes	82	Yes	Yes
9	86	Yes	-	-	Yes
10	17	No	39	No	Yes

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 has some redundant read accesses but eliminating them would not improve performance significantly. So, the programmer chose to keep it that way. Toddler, a software based approach, detects all 10 bugs correctly. In a nutshell, although Prometheus misses 2 bugs, its detection ability is comparable to a software based approach.

F. Bug Analysis

We identified two reasons for redundant accesses in the buggy code - inefficient data structures and inefficient algorithm. As an example of inefficient data structure, in bug 1, the programmer creates an array, populates it with necessary data, and then removes some of the elements one at a time. Each removal process traverses the whole array repeatedly. The fixed version of this bug uses a hash table instead of an array and thus, prevents the repeated traversal of the whole structure during the removal process. Bug 5, 6, 7, and 10 are like this. As an example of inefficient algorithm, in bug 2, the programmer removes an element from one data structure and then from another internal data structure. However, if the first removal shows that the element is not present in the first data structure, the second removal becomes unnecessary. But the programmer does it anyway. The fixed version basically eliminates the unnecessary second removal operation. As an another example, in bug 3, the code inserts one element at a time in an array instead of inserting all elements together at once. Bug 4,8, and 9 result from inefficient algorithms. Note that for the fixed programs, there are often significant number of redundant accesses. We identified mainly two reasons - programmer effort and redundancy in test cases. Often a read operation reads the same set of values but in a different sequence. Eliminating this type of redundancy requires a lot of programming effort with very little potential benefit and hence, the programmers choose not to do it. Some of the test cases used by the developers contain significant redundancies to expose the bug. This is exactly the case for bug 8. Therefore, even a fixed version has some redundant accesses.

G. Execution Overhead

Since our test programs are relatively small, we use 7 SPEC benchmarks for determining execution overhead. We collect the total number of different loops as well as the total number of iterations. We also calculate the resulting execution overhead. Table V shows the data. Prometheus incurs overhead from 0.71% to 3.72%, with an average of 1.87%. This overhead is low enough for production run diagnosis.

Program	Num. Loop	Num Iter. (x1000)	Overhead (%)
bzip2	348	1129430	3.72
gcc	4165	37176	1.57
gzip	168	8899	2.28
mcf	72	7605	1.44
perlbmk	149	237	2.02
twolf	751	5360	1.39
vpr	206	18312	0.71
Average	837	172431	1.87

TABLE V: Execution overhead.

H. Hardware Requirement

Prometheus uses 8 address signatures and 8 data signatures. Each signature is 512 bit long. In addition to this, Prometheus module contains 4 counters, 3 floating point registers, 1 register to hold the id of a loop, and 1 one bit flag. All the counters and registers are assumed to be 32 bit long. So, 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 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. When the bugs are fixed, Prometheus does not falsely detect any bug except in one case. It has a negligible execution overhead of 1.87%. It requires only ($\approx$)1 Kbyte of extra hardware structures.

Acknowledgment

The authors would like to thank...

References

[1] "GNU gprof," https://sourceware.org/binutils/docs/gprof/.

[2] Apache, "Ant Build Tool," http://ant.apache.org/.

[3] ——, "Apache Collections Bug 406," https://issues.apache.org/jira/browse/COLLECTIONS-406.

[4] ——, "Apache Common Collection Library," https://commons.apache.org/proper/commons-collections/.

[5] B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 11, no. 7, July 1970.

[6] Bugzilla@Mozilla, "Bugzilla keyword descriptions," https://bugzilla.mozilla.org/describekeywords.cgi.

[7] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas, "Bulk Disambiguation of Speculative Threads in Multiprocessors," in *ISCA*, June 2006.

[8] I. Corporation, "Intel vTune," http://software.intel.com/en-us/intel-vtune.

[9] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos, "Profileme: hardware support for instruction-level profiling on out-of-order processors," in *MICRO*, Dec 1997.

[10] J. Demme and S. Sethumadhavan, "Rapid identification of architectural bottlenecks via precise event counting," in *ISCA*, June 2011.

[11] A. Diwan, M. Hauswirth, T. Mytkowicz, and P. F. Sweeney, "Traceanalyzer: A system for processing performance traces," *Softw. Pract. Exper.*, vol. 41, no. 3, March 2011.

[12] Google, "Google Core Library," http://code.google.com/p/guava-libraries/.

[13] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces." in *ICSE*, June 2012.

[14] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind, "Vertical profiling: Understanding the behavior of object-priented applications," in *OOPSLA*, October 2004.

[15] Y. Huang, Z. Cui, L. Chen, W. Zhang, Y. Bao, and M. Chen, "Halock: Hardware-assisted lock contention detection in multithreaded applications," in *PACT*, September 2012.

[16] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *PLDI*, June 2012.

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, June 2005.

[18] D. Mituzas, "Embarrassment," http://dom.as/2009/06/26/embarrassment/.

[19] G. E. Morris, "Lessons from the Colorado Benefits Management System Disaster," http://www.ad-mkt-review.com/public_html/air/ai200411.html.

[20] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas, "Sigrace: signature-based data race detection," in *ISCA*, June 2009.

[21] NetBSD Documentation, "How lazy FPU context switch works," http://www.netbsd.org/docs/kernel/lazyfpu.html, 2011.

[22] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *PLDI*, June 2007.

[23] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: Detecting performance problems via similar memory-access patterns," in *ICSE*, June 2013.

[24] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC Simulator," January 2005, http://sesc.sourceforge.net.

[25] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam, "Implementing signatures for transactional memory," in *MICRO*, December 2007.

[26] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky, "Go with the flow: Profiling copies to find runtime bloat," in *PLDI*, June 2009.

[27] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky, "Finding low-utility data structures," in *PLDI*, June 2010.