

AccMon: Automatically Detecting Memory-related Bugs via Program Counter-based Invariants^{*}

Pin Zhou, Wei Liu, Long Fei[†], Shan Lu, Feng Qin, Yuanyuan Zhou, Samuel Midkiff[†] and Josep Torrellas

Department of Computer Science,

University of Illinois at Urbana-Champaign

{pinzhou,liuwei,shanlu,fengqin,yzzhou,torrellas}@cs.uiuc.edu

[†]School of Electrical and Computer Engineering,

Purdue University

{lfei,smidkiff}@purdue.edu

Abstract

This paper makes two contributions to architectural support for software debugging. First, it proposes a novel statistics-based, on-the-fly bug detection method called *PC-based invariant detection*. The idea is based on the observation that, in most programs, a given memory location is typically accessed by only a few instructions. Therefore, by capturing the invariant of the set of PCs that normally access a given variable, we can detect accesses by outlier instructions, which are often caused by memory corruption, buffer overflow, stack smashing or other memory-related bugs. Since this method is statistics-based, it can detect bugs that do not violate any programming rules and that, therefore, are likely to be missed by many existing tools. The second contribution is a novel architectural extension called the *Check Look-aside Buffer (CLB)*. The CLB uses a Bloom filter to reduce monitoring overheads in the recently-proposed iWatcher architectural framework for software debugging. The CLB significantly reduces the overhead of PC-based invariant debugging.

We demonstrate a PC-based invariant detection tool called *AccMon* that leverages architectural, run-time system and compiler support. Our experimental results with seven buggy applications and a total of ten bugs, show that AccMon can detect all ten bugs with few false alarms (0 for five applications and 2-8 for two applications) and with low overhead (0.24-2.88 times). Several existing tools evaluated, including Purify, CCured and value-based invariant detection tools, fail to detect some of the bugs. In addition, Purify’s overhead is one order of magnitude higher than AccMon’s. Finally, we show that the CLB is very effective at reducing overhead.

1. Introduction

Software bugs significantly affect system reliability and availability, accounting for as many as 40% of computer system failures [24]. According to NIST, software bugs cost the U.S. economy an estimated \$59.5 billion annually, or 0.6% of the GDP [27]. Memory-related bugs are among the most prevalent and difficult to catch of all software bugs, particularly in programs written in an unsafe language such as C/C++. In addition, they are often exploited to launch security attacks [7].

As micro-architectural innovations have significantly improved performance, interest has recently risen in the architecture commu-

nity to use transistors to improve software robustness. For example, Prvulovic and Torrellas proposed ReEnact [31], which uses the state buffering, rollback and re-execution features of Thread-Level Speculation (TLS) to detect data races on the fly. Xu et al designed the “flight data recorder” [39], which enables off-line deterministic replay and can be used for postmortem analysis of a bug. Our previous work on iWatcher [40] provides a convenient and efficient architectural framework for dynamic monitoring. While recent work provides a good foundation, architectural support for software debugging is still far from providing a complete solution. This paper takes another step toward the goal of improving software robustness.

Many methods have been proposed to detect bugs dynamically during execution. These methods can be classified into two categories: the *programming-rule-based* approach and the *statistics-rule-based* approach. Methods in both categories check for violations of certain rules at run time, but they focus on different types of rules. The programming-rule-based approach focuses on rules that should be followed when programming in a specific language such as C/C++. “An array pointer cannot move out-of-bounds” is an example of these rules. Much work has been conducted on this approach, including Purify [15], CCured [6, 28], SafeC [1] and Jones and Kelly’s tool [19].

The statistics-rule-based approach is a newly explored direction that extracts rules (e.g., invariants) statistically from multiple successful executions (e.g., in-house regression tests) or multiple periods of a single long-running execution, and then uses these rules to check for violations in a later execution (or later in the same long-running execution). This approach is promising because it can catch bugs that may not violate any programming rules. Many statistics-based rules such as value-based invariants (i.e., a variable’s value always falls in a certain range during normal runs) are related to applications semantics. Such information is difficult to infer from the code, and is too tedious to be documented or annotated by programmers.

Only a few studies have been conducted on the statistics-rule-based approach, and almost all are software-only solutions. Liblit et al [23] uses statistical analysis to find the difference between abnormal and normal runs for the purpose of providing more information for postmortem bug analysis. DAIKON [11, 12] and DIDUCE [14] focus on detecting bugs on the fly by automatically extracting invariants and detecting violations during execution. Both DAIKON and DIDUCE consider only value-based invariants, and therefore can miss bugs that do not violate these invariants.

Novel architectural support would provide several benefits for statistics-rule-based bug detection over software-only solutions: (1)

^{*}This work was supported in part by NSF under grants CCR-0325603, EIA-0072102, CHE-0121357, and EIA-0081307; by DARPA under grant F30602-01-C-0078; by an IBM SUR grant; and by additional gifts from IBM and Intel.

Efficiency: Architectural support can significantly lower the overhead of dynamic monitoring because it does not need extensive code instrumentation. Note also that such instrumentation can interfere with compiler optimizations. Moreover, it is possible to use extra hardware to speed up certain operations. Both iWatcher and AccMon are examples that demonstrate this benefit. (2) *Accuracy:* Architectural support can avoid pointer aliasing problems and accurately capture all desired accesses to monitored memory objects. (3) *Portability:* Architectural support can be language-independent, cross-module and easy to use with low-level system code such as the operating system. Moreover, it can be designed to work directly with binary code without recompilation.

Our Contributions. This paper proposes two innovative ideas in architectural support for software bug detection. First, we propose a novel statistics-based method, called *program counter (PC)-based invariance*, to detect memory-related bugs *on the fly*. This idea is based on the observation that, in most programs, a given variable is typically accessed by only a few instructions. We validate this observation using statistical analysis with nine applications (See Section 3). Based on this observation, if we can capture the invariant of the set of PCs that normally access a given key variable, it is possible to detect accesses by outlier instructions that are often caused by memory corruption, buffer overflow, stack smashing or other memory-related bugs. This is regardless of the values that these instructions assign to the variables.

Second, we propose a novel architectural extension, called the *Check Look-aside Buffer (CLB)*, that uses a Bloom filter [3] to reduce the monitoring overhead in iWatcher. This extension takes advantage of the good temporal locality that exists in data accesses to filter out a large percentage of monitored accesses. This extension reduces the overhead by up to 80.6% in our experiments.

Based on the above two ideas, we have built an automatic, low-overhead, low-false-alarm, PC-based invariant detection tool called *AccMon* (Access Monitor, pronounced as “A-k-Mon”) that uses a combination of architectural, run-time system, and compiler support to catch hard-to-find memory-related bugs. First, AccMon leverages the iWatcher framework with the CLB extension to monitor accesses to key variables. Second, the run-time system automatically infers PC-based invariants and detects violations of these invariants. Third, AccMon uses compiler support to provide certain optimizations to reduce the amount of monitoring and prune false alarms.

Our experimental results with seven buggy applications (with a total of ten bugs) show that AccMon can detect all ten bugs with few false alarms (0 for five applications and 2-8 for two applications), whereas several tested existing tools fail to detect some bugs. *In particular, AccMon catches a bug in the bc application that has never been reported.* AccMon also has low overhead (0.24-2.88 times), which is an order of magnitude lower than Purify [15]. Our results also show that the CLB architectural extension and other optimizations significantly reduce overheads.

AccMon complements other existing memory-bug detection tools, including programming-rule-based approaches and statistics-rule-based approaches. This is because AccMon provides several unique advantages, some or all of which are unavailable in other tools:

- Since AccMon is a statistics-based approach, it does not need pointer-type/object information. Therefore, it can detect bugs

that either do not have such information (e.g., because of fine-grained pointer manipulation through various type-casting), or do not violate pointer-type/object association (such as a wrong pointer assignment bug caused by copy-paste). Our experiments identify two such bugs that are detected by AccMon but are missed by programming-rule-based tools such as Purify [15] and CCured [6, 28].

- Since AccMon uses architectural support to detect accesses to monitored memory objects, it can detect memory corruption that occurs in third-party libraries whose source code is unavailable. We have found one such bug in our experiments that is detected by AccMon but missed by the other tested tools.
- AccMon does not rely on variable values, and therefore can detect bugs that do not violate value-based invariants. In our experiments, AccMon detects six bugs that are very difficult to catch using value-based invariant detection tools such as DAIKON [11, 12] and DIDUCE [14].
- Since AccMon relies on architectural support, it is language-independent and easy to use for low-level system code, e.g., operating system code. In our experiments, AccMon is able to catch an extracted version of a real bug that exists in the latest version of Linux.
- Although the current AccMon implementation uses source code in order to exploit certain compiler-based optimizations, it can directly use binary code without recompilation.
- AccMon’s overhead is low. Moreover, AccMon uses the iWatcher framework that can dynamically turn on/off monitoring with little overhead, completely eliminating the overhead in unmonitored code. Therefore, AccMon can be used on production runs.

2. Background

2.1. Invariant-Based Bug Detection

Similar to previous invariant-based bug detection work such as DAIKON [11, 12] and DIDUCE [14], AccMon can be used in two scenarios. The first one is debugging programs that fail on some inputs. It is common for many programs to work correctly on some inputs (especially those tested in-house) but to fail on others. Invariant detection tools can be used to automatically provide debugging information on failing cases by checking for invariants inferred from successful cases. The second one is debugging failures in long-running programs. Some bugs occur only after the program has executed for a long time. These bugs are very common in server programs, and are usually hard to track down because they cannot be easily (or quickly) reproduced. Automatic invariant detection and checking tools can use a period of execution time before the bug occurs to extract invariants, and then continuously check for violations of these invariants during the remainder of the execution to detect bugs.

For the above two usage models, the dynamic invariant detection and checking process has two phases: the training phase and the bug-detection phase. The training phase tries to extract invariants from the program’s execution using good inputs in the first

usage scenario, or from the initial execution (before a bug occurs) in the second usage scenario. The bug-detection phase checks for violations of invariants during the execution on failing or untested inputs, or the remaining execution after the training phase.

2.2. iWatcher

Our work is based on the iWatcher framework [40], which is an architecture for dynamically monitoring memory locations. We use iWatcher because it provides several advantages described in Section 1, namely efficiency, accuracy and portability.

The main idea of iWatcher is to associate programmer-specified monitoring functions with monitored memory objects. When a monitored object is accessed, the monitoring function associated with this object is automatically triggered and executed by the hardware without generating an exception to the operating system. iWatcher is flexible because monitoring functions are not hard-wired into the architecture, but are provided by programs or external software tools.

Programs can use *iWatcherOn* and *iWatcherOff* to turn on and off the monitoring of a memory object. These operations can be inserted into programs either automatically by a compiler or an instrumentation tool, or manually by a programmer. The interfaces of *iWatcherOn* and *iWatcherOff* are:

```
iWatcherOn(MemAddr, Length, WatchFlag, MonitorFunc,
           Param1, Param2, ... ParamN);
iWatcherOff(MemAddr, Length, WatchFlag,
            MonitorFunc);
```

When *iWatcherOn* is called, it associates a monitoring function *MonitorFunc()* with the memory object which begins at *MemAddr* and has size *Length*. The *WatchFlag* specifies what types of accesses (read, write, or both) to this memory object should trigger the specified monitoring function *MonitorFunc*. After the *iWatcherOff* call, monitoring of the memory object with the specified monitoring function is disabled. There are two more operations, *EnableMonitoring()* and *DisableMonitoring()*, that enable and disable system-wide monitoring. After *DisableMonitoring()* is called, no access will trigger a monitoring function. In this case, there is no monitoring overhead. Monitoring can be re-initiated by *EnableMonitoring()* when desired.

3. PC-Based Invariants

When observing the behavior of programs, we found an interesting characteristic: program location and data accessed are highly correlated. This characteristic has two aspects. First, for most memory objects, only a few instructions access a given object. Second, in short-running programs, for runs with different inputs, the sets of instructions that access a given object are remarkably similar; in long-running programs, the set of instructions that access a given object is relatively stable across different execution periods (of duration long enough to capture at least one cycle of most computation phases). The latter is especially the case for long-running server programs.

Intuitively, this characteristic makes sense. In most programs, a memory object is accessed at only a few places. For example, a linked list is usually accessed by the list manipulation functions. Also, from the programmers' point of view, it is very difficult to

write or understand a program where a memory object can be accessed in many places. For convenience, we refer to the set of instructions that normally access a given memory object as its *AccSet*.

Based on this observation, this paper proposes a new type of invariant, the Program Counter-based (PC-based) invariant. Generally speaking, a PC-based invariant captures the relationship between a memory object and its *AccSet*. Based on this relationship, it is possible to detect "illegal" accesses by an outlier instruction (an instruction that is not in the *AccSet* of the accessed memory object) due to buffer overflow, stack smashing, dangling pointers, memory corruption or other memory-related bugs.

To validate this observation and understand the characteristics of *AccSets*, we have analyzed the behavior of nine programs (six real applications used in our evaluation of *AccMon* and three SPEC2000 benchmarks). In particular, we examine the average size and stability of *AccSets*. If the average *AccSet* size is large, it will be hard to detect bugs because the confidence of identifying an outlier instruction will be low. Similarly, if most *AccSets* are not stable across different inputs or different execution periods, they cannot be used to detect bugs because they may introduce many false alarms.

To find the average size and stability of *AccSets*, we collect the *AccSets* for all global objects in the nine programs, using multiple runs with different inputs. We then examine the cumulative distribution of the *AccSet* sizes and measure the similarity of *AccSets* across multiple runs with different inputs. We have also conducted similar statistical analyses for heap objects and the results are similar.

Figure 1 shows the cumulative distributions of the *AccSet* sizes for the three SPEC2000 benchmarks and six real applications. For the SPEC2000 benchmarks, 96% of the global objects in *vpr* have *AccSet* sizes less than 3, 90% of the global objects in *parser* have *AccSet* sizes less than 5, and 80% of the global objects in *gzip* have *AccSet* sizes less than 9. For the six real applications, around 85-100% of the global objects have *AccSet* sizes less than 10. In other words, the average *AccSet* size is small, and therefore *AccSets* can be used to detect outlier accesses with reasonable confidence.

To measure the stability of *AccSets* across multiple runs with different inputs, we introduce a metric called *Similarity*. For a given data object *OBJ* and *n* runs, the similarity for this object across the *n* runs is defined as

$$Similarity(OBJ) = \frac{|\cap (S_1, S_2, \dots, S_n)|}{|\cup (S_1, S_2, \dots, S_n)|}$$

where S_i is the *AccSet* of *OBJ* in run i . The similarity of an object is the size of the intersection of its *AccSets* across different runs divided by the size of the union of its *AccSets* in all the runs. It measures the fraction of common instructions in the total possible instructions that access this object. If the *AccSet* for an object is very stable, the similarity metric is close to one. If it is very unstable, the similarity metric is close to zero.

Figure 2 shows the cumulative distributions of the *AccSet* similarity for different runs. The figure shows that most objects have a similarity close to one, which indicates that most *AccSets* are stable across different runs. In the SPEC2000 benchmarks, 96-100% of the global objects' *AccSets* have similarity values greater than 0.97. For the six real applications shown in Figures 2(b) and 2(c),

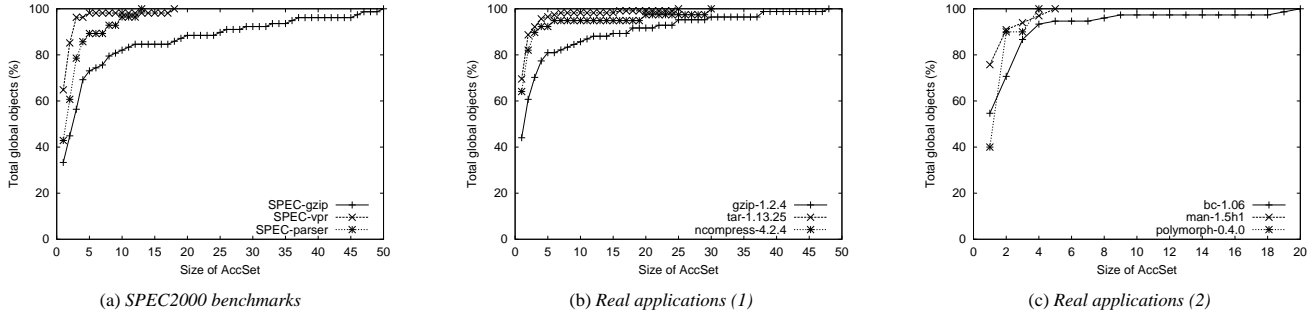


Figure 1. Cumulative distribution of AccSet size for three SPEC2000 benchmarks and six real applications. Each cumulative distribution curve gives the percentage of global data objects whose AccSet sizes are smaller than or equal to a given size. A high percentage for a small size means that most objects have small AccSets sizes. Note that the SPEC-gzip and gzip-1.2.4 applications are different.

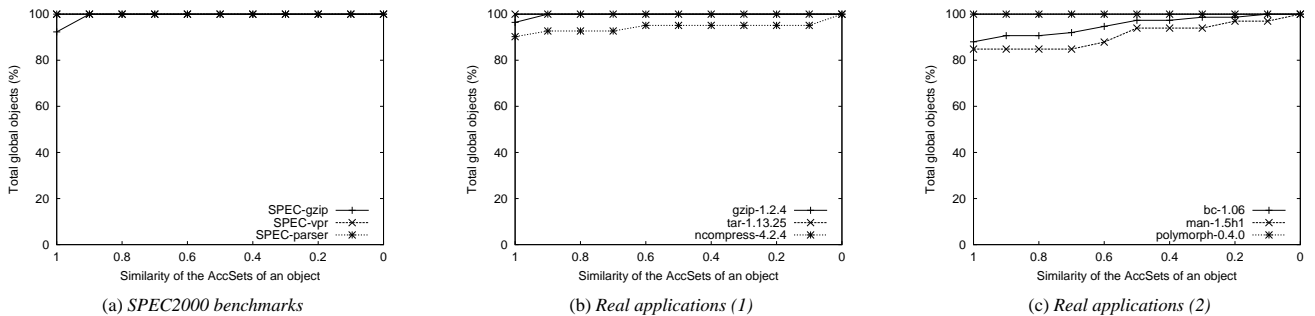


Figure 2. Cumulative distribution of AccSet similarity for three SPEC2000 benchmarks and six real applications. Each cumulative distribution curve shows the percentage of global data objects whose AccSets have a similarity greater than or equal to a given value. A high percentage at a value close to 1 indicates that most objects’ AccSets are similar across different runs. Note that the x-axis starts at 1 and goes to 0.

around 84-100% of the global objects’ AccSets have similarity values greater than 0.97. These results show that AccSets are quite stable across multiple runs with different inputs.

Since our infrastructure for recording AccSets cannot support long-running server programs (our iWatcher simulator does not run an OS), we do not have results on similarity analysis for different execution periods of a very long run. However, we expect that the results would be similar because most long-running server programs perform similar computation in different periods. For example, the Apache web server continuously services requests.

Further validation of our observations on PC-based invariants is provided by the data in Section 6.

4. Design of AccMon

Based on the above observation, a violation of a PC-based invariant usually indicates a potential bug in the program. For example, if a memory location is accessed by an instruction which has never accessed this location during normal execution, it is likely that this access is “illegal”, resulting from a memory-related bug. In this section, we design a tool to automatically detect these cases. We call this tool *AccMon*.

4.1. Overview

AccMon uses some architectural support as well as some compiler and run-time software infrastructure. The main functionality of each of the components of AccMon is shown in Table 1.

AccMon uses iWatcher to catch all memory accesses to monitored memory objects and trigger a monitoring function at such accesses [40]. The monitoring function will check if the PC used to access the object is in the object’s AccSet. If the TLS option of iWatcher is enabled, the main program is speculatively executed in parallel while the monitoring function runs, to reduce overhead.

| Component | Main Functionality | |
|-----------------|---|--|
| Architecture | iWatcher | Catch accesses to monitored objects, invoke monitoring functions to check if a PC belongs to the AccSet of an object, and execute the main program in parallel with monitoring functions |
| | CLB | Filter most accesses that do not violate PC-based invariants |
| Compiler | Insert iWatcherOn/Off to monitor key memory objects, and provide hints to reduce overheads and false alarms | |
| Run-time system | Extract invariants, detect violations and rank errors | |

Table 1. Functionality of the components of AccMon.

To further reduce monitoring overhead, we propose the Check Look-aside Buffer (CLB). The CLB is a hardware cache that, for most recently-accessed monitored objects, filters out the accesses that do not violate the PC-based invariant. To do that, the CLB keeps the AccSets for several recently-accessed monitored objects. The memory address and PC of each load and store are checked against the contents of the CLB. If the memory address is found and the PC is part of the AccSet of the address, the monitoring function is not executed. If, instead, the memory address is found but the PC is not part of its AccSet, an access that violates the PC-based invariant has been found. Finally, if the memory address is

not found in the CLB and *iWatcher* indicates that this access is to a monitored object (i.e., a triggering access), the monitoring function is executed to check if the access violates the PC-based invariant. In addition, the run-time system inserts this address and its *AccSet* into the CLB. If necessary, the *AccSet* of a memory object in the CLB can be dynamically augmented with a new PC (See Section 4.2 for details).

The CLB resides in the processor. Figure 3 shows how it interacts with the different pipeline stages and the *iWatcher* trigger bit. More details are given in Section 4.2.

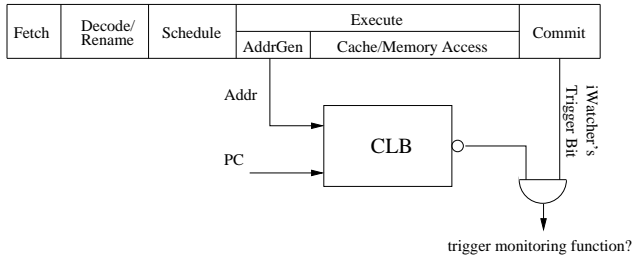


Figure 3. Interaction of the CLB with the processor pipeline and the *iWatcher* trigger bit.

We modify the Cetus compiler [21] to select memory objects to be monitored and to provide hints to reduce the number of false alarms and the run-time overhead. In our current implementation, we monitor global data objects, heap objects, and a few key stack objects, such as the stack locations that store return addresses. The compiler uses *iWatcherOn* to request *iWatcher* to monitor an object, and *iWatcherOff* to stop doing it. While the monitoring is on, *iWatcher* will automatically catch accesses to monitored objects.

The compiler also provides hints to reduce overheads and false alarms. For example, the compiler passes information to the run-time system regarding what instructions use pointers or access arrays. These instructions are more likely to induce bugs if their PCs are detected as outliers. The compiler can also temporarily disable system-wide monitoring using *DisableMonitoring()* in certain functions that do not have pointers or array accesses.

Note that although our current implementation uses a compiler to insert *iWatcherOn/Off()* into the source code, *AccMon* can also leverage a binary-instrumentation tool to avoid recompilation if source code is unavailable. However, source level instrumentation can provide some advantages, such as the optimizations described above and in Section 4.4. Since most debugging is done in-house, recompilation may not be a major issue. In addition, since monitoring can be dynamically turned off for most production runs by the underlying *iWatcher* architecture, code can be shipped with *iWatcherOn/Off* instrumentation.

The run-time system executes the monitoring function that detects and checks invariants. There are two distinct phases: the training phase and the bug-detection phase. During the training phase, the monitoring function dynamically builds *AccSets* for the monitored objects. In addition, it also tracks the number of occurrences of each PC in an *AccSet*. This information will be used later, in the bug-detection phase, to determine the confidence level for an outlier PC. During the bug-detection phase, the monitoring function checks each triggering access that does not hit in the CLB, to see if it is an outlier. In addition, the monitoring function dynamically

adjusts the confidence level as execution progresses. Section 4.3 describes the basic algorithms in more detail.

At the end of the bug-detection phase, *AccMon* produces an error report with a ranked list of detected violations. The violations are sorted by their confidence levels as computed by *AccMon*. Programmers can go through the list to check for potential bugs. Programmers can also mark certain errors as false alarms, and add the newly-observed PCs that cause false alarms into *AccSets*, so that *AccMon* can learn from its mistakes to reduce the number of false alarms in future runs.

4.2. CLB with a Bloom Filter

The main purpose of the CLB is to reduce overheads by filtering most of the valid accesses to monitored objects. Such valid accesses do not need to trigger the monitoring function. By filtering most of the valid accesses, *AccMon* can significantly reduce the number of times the monitoring function is executed. Since the overhead for the bug-detection phase is more important than the overhead for the training phase, the CLB is only used for the bug-detection phase in our current prototype of *AccMon*.

Designing the CLB is challenging. A major constraint is that the CLB needs to be very fast. Indeed, as shown in Figure 3, the CLB is tightly coupled with the processor pipeline. Moreover, it is accessed by every load and store instruction. In a wide-issue processor, the CLB is accessed very often and has little time to make a decision. Consequently, it cannot be built as a large associative table.

In addition, the CLB ideally needs to keep a lot of information. Since *AccMon* monitors every global data object, heap object and stack return address, there can be many monitored objects. For example, we have up to 10,000 such objects in our experiments. Suppose that, on average, each *AccSet* contains 10 PCs, where each PC is 4 bytes. In this case, an *AccSet* requires at least 48 bytes, since it needs 8 bytes to record the memory object’s start and end address. Therefore, maintaining all *AccSets* would require a 480,000-byte CLB. Such information would need to be organized in a two-level manner: A memory address would first index the table and find the matching CLB entry; then, the PC would be used to index the *AccSet* of the address to find if the PC was there.

Clearly, keeping all this information in a fast CLB is impractical. Moreover, it is unclear how to handle *AccSets* that contain more than 10 PCs.

To address these challenges, *AccMon* uses two strategies to make the CLB hardware practical: the first one is to use a Bloom filter to avoid storing all the PCs of an *AccSet* in each entry; the second one is to treat the CLB as a cache, which maintains only the *AccSets* of recently-accessed monitored objects.

We use a Bloom filter for the CLB because it can quickly test whether a PC belongs to the *AccSet* of the accessed object, and it uses only a few bytes to maintain a relatively large set. The Bloom filter was first proposed by Bloom [3] to support fast membership testing of a set. It uses multiple hash functions to map an element into a bit vector. For each member element, its corresponding bits in the vector are set to 1. To test whether an element is a member or not, its corresponding bits based on the hash functions are tested. If one of the bits is 0, the element does not belong to the set. Otherwise, the element may belong to the set. A Bloom filter never has false negatives, but it may introduce false positives due to hash

collisions. However, if the vector is long enough and enough bits are used for hashing, the probability of false positives is very low.

Figure 4 shows the implementation of the CLB. Similar to a TLB, the CLB is a fully-associative table with only a few entries (4 or 8 in our experiments). At each memory instruction, the memory address is used to index the CLB. Each CLB entry has 24 bytes, storing the start address, end address and the Bloom filter vector for a recently-accessed monitored memory object. The CLB uses 128 bits as the Bloom filter vector. At each memory instruction, 20 bits (bit 2 to bit 21, starting from the least significant bit) are taken from this instruction’s PC. The 20 bits are broken into 4 parts, with 5 bits each. Each part is used to directly index 32 bits in the Bloom filter vector of the corresponding CLB entry. This partial address indexing idea was also used in [30]. We use a direct index instead of a hash function to simplify the logic as much as possible.

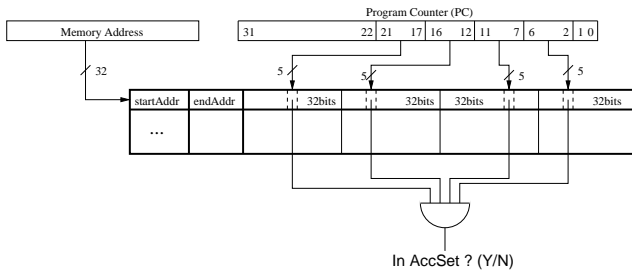


Figure 4. Implementation of the CLB using a Bloom filter.

If all indexed bits in the four parts have value 1, we conclude that this PC is in the AccSet. Therefore, this access is assumed to be valid and can be filtered even if it is recorded as a triggering access by iWatcher (Figure 3). Since we directly index bits 2-21 of a PC to four bits in the Bloom filter vector, the collision rate is almost zero, and so is the rate of false positives introduced by the CLB. A false positive occurs when an outlier PC is incorrectly flagged as part of the AccSet.

Treating the CLB as a cache exploits the good temporal locality of data accesses. Most programs have well-clustered memory accesses: an object such as an array or a structure tends to be accessed many times in a short period of time. If we keep recently accessed monitored objects in the CLB (with one entry per object), we only need a small table with a few entries to filter most valid accesses to monitored objects. As shown later in Table 6 in Section 6.2, the CLB hit ratios for most of the evaluated applications are very high, namely 80.1%-99.9% and 83.8%-99.9% for a 4-entry and an 8-entry CLB, respectively.

The CLB uses the least recently used (LRU) algorithm for replacement. After the CLB misses a triggering access, the AccMon run-time system inserts the accessed object’s AccSet into the CLB. If the CLB is full, the LRU entry in the CLB is replaced. This is controlled by the run-time system because CLB misses are handled by the AccMon monitoring function in the run-time system.

4.3. Basic Algorithms

The basic training and bug-detection algorithms, implemented mainly in AccMon’s run-time system, have three parts: (1) extracting invariants, (2) checking for violations of invariants, and (3) ranking results. All three parts need to access a core software data

structure called “PC-based invariants Table” (PCT), which maintains the AccSet for each monitored memory object. The PCT is maintained as a hash table and can be searched using a memory object’s name, as described in Section 4.4.1. Initially, the PCT is empty. Each PCT entry contains both an AccSet and an occurrence counter for each PC in the AccSet. This information is used to calculate confidence and rank results, as described later.

During the training phase with bug-free runs (or bug-free execution phases for long-running programs), AccMon builds the AccSet for each monitored object. At an access to an object *obj* by an instruction, AccMon first looks up *obj* in the PCT. If this *obj* is not in the PCT, it is inserted in it. In any case, the instruction’s PC is added to the *obj*’s AccSet if that PC is not already a member. The PC’s occurrence counter is also incremented. At the end of each training run, the PCT is saved on disk and is reloaded to memory at the beginning of the next training run. Since all triggering accesses made during the training phase need to go through the run-time system, the CLB is disabled during the training phase.

During the bug-detection phase, AccMon detects violations of PC-based invariants. In this phase, the CLB is enabled. When an object *obj* is accessed by a PC, the CLB is checked for *obj*. If the access is not filtered by the CLB (either because the CLB misses this *obj* or the corresponding Bloom filter indicates that this PC is not in *obj*’s AccSet) and the access is a triggering one, the AccMon monitoring function is triggered to determine if this is an outlier access. To do that, AccMon first checks the PCT to see if the PC is already in *obj*’s AccSet. If it is, then *obj* and its AccSet are inserted into the CLB. Otherwise, the AccMon monitoring function reports the access as a suspect and stores it in a table (the *Suspect Table*). Subsequent accesses by the same PC to the same object are not reported.

To reduce the programmers’ effort in analyzing the error report produced by AccMon, the errors are ranked based on their confidence values. A programmer only needs to check the top (e.g. 10) reported errors to find bugs. For an outlier access to object *obj*, its confidence value should depend on the number of observed accesses to *obj*, and *obj*’s AccSet size. If *obj* has been accessed only a few times, an outlier access to *obj* is less likely to be a bug. Instead, it is more likely to be a false alarm caused by insufficient training. Similarly, if *obj*’s AccSet is large, the possibility for this outlier to be a bug is also relatively low. Similar intuition is also shared by other work [10, 14].

Moreover, we also consider the historical behavior of the outlier instruction. If the instruction has been previously identified as an outlier for other memory objects, it is more likely to be a bug because this instruction may have corrupted many other objects.

Combining all these factors, the confidence value of an error is computed by using the formula:

$$Confidence = \frac{NumAccess_{total} \times (NumOccurrence_{pc} + 1)}{AccSetSize + 1}$$

where $NumAccess_{total}$ is the total number of times *obj* has been accessed, $NumOccurrence_{pc}$ is the number of times this outlier PC has been identified as an outlier for other objects as well, and $AccSetSize$ is *obj*’s AccSet size. While it is possible to further refine our ranking function, our results show that this ranking function is already very good.

4.4. Design Issues

4.4.1. Monitoring and Naming Objects

AccMon currently monitors all global data objects, all heap objects and key stack objects, e.g. stack locations used to store return addresses. To monitor heap objects, we intercept all memory allocation functions and insert instructions to call *iWatcherOn* immediately after a memory-allocation, and *iWatcherOff* immediately before a memory-free. For *realloc()*, *iWatcherOff* is called before it and *iWatcherOn* after it.

We must name each memory object in the PCT. The primary constraint on the naming strategy is that the name of an object cannot change across different runs. For global data objects, their virtual memory addresses are used as their names. A global object's address is decided at compile time and will not change across different runs.

However, this simple naming strategy does not work for heap and stack objects because their virtual addresses can change across different runs. Instead, we use a call-chain naming strategy, which has been used in some previous work [2, 4, 20] for other purposes. When a heap object is allocated, it is named based on the current call-chain, i.e., the XOR-folding of the call-site address chain. As suggested in the literature [2, 4, 20], it is sufficient to use the last four call-sites in the call chain to distinguish heap/stack objects from one another. Although several heap objects may have the same call-chain, e.g. those allocated in a *for* loop, it is not important for our case since those objects are naturally similar and usually have similar AccSets.

4.4.2. Pruning False Alarms

It is possible that some corner cases caused by rarely touched paths end up being reported as violations of an invariant. These are false alarms. Too many false alarms make a debugging tool unusable.

To reduce false alarms, we use, in addition to confidence levels, simple heuristics. Specifically, by analyzing the behavior of buggy code, we have found that most invalid accesses in C/C++ occur in pointer dereferences and array accesses. The invariant violations caused by pointer or array accesses are more likely to be bugs, while violations caused by other accesses are more likely to be corner cases caused by rarely executed paths.

Based on the above observation, we use the Cetus compiler [21] to identify pointer-based dereferences and array accesses. The Cetus compiler generates a list of PCs that may be pointer-based dereferences or array accesses. Of course, the compiler has to be conservative, otherwise AccMon may miss some bugs. During the bug-detection phase, the AccMon monitoring function checks a suspect PC against this list. If the PC is not in the list, the suspect access is unlikely to be a bug. This optimization may cause some bugs to escape detection, but the probability is low based on our program behavior analysis.

4.4.3. Reducing Overhead

Overhead is another major issue for software debugging. We consider the two phases in which AccMon is used: the invariant training phase and the bug-detection phase. Since the training phase typically takes place in-house using successful regression test runs

before the software is released, or when a long-running server program has very light load (e.g. when it receives few requests), the overhead during this phase is less critical. In contrast, minimizing the overhead in the bug-detection phase is very important because such overhead may prevent some time-related bugs from occurring. In addition, it also affects the length of program execution that can be realistically monitored.

There are two ways to reduce overheads in AccMon: reducing the number of accesses monitored, and reducing the overhead of monitoring an access. The following three optimizations can be used by AccMon to reduce overheads. The first two belong to the first type and the third one belongs to the second type:

- *Monitor only store accesses.* Since corrupting writes are typically more harmful than illegal reads, it may be enough to monitor only store instructions. This can be achieved by setting the *WatchFlag* in the *iWatcherOn* call appropriately [40]. It is possible that this will lead to some bugs going undetected, but we feel that the probability is relatively low. In any case, users can disable or enable this optimization based on their overhead tolerance level.
- *Disable monitoring in certain functions.* If a function contains no pointer dereference or array access, we can turn off the monitoring of memory accesses. This optimization is performed using *EnableMonitoring()* and *DisableMonitoring()*. We have not implemented this optimization in AccMon yet.
- *Software optimization.* Besides using the CLB to filter out most valid accesses to monitored objects, AccMon software can also be optimized to reduce the overhead of the monitoring function. For example, in our current implementation, we use a hash table to manage the PCT.

5. Evaluation Methodology

5.1. Methodology Overview

We use cycle-accurate execution-driven simulations to model a workstation with *iWatcher* [40] and AccMon functionality. The parameters of the architecture are shown in Table 2. The architecture includes a 4-context SMT processor with optional TLS support.

| | | | |
|---------------------|--|--------------------|--------|
| CPU frequency | 2.4GHz | CLB entries | 4 or 8 |
| Thread contexts | 4 | ROB size | 360 |
| Fetch width | 16 | Instruction window | 160 |
| Issue width | 8 | Int FUs | 6 |
| Retire width | 12 | Ld/st FUs | 4 |
| Ld/st queue entries | 32/thr | FP FUs | 4 |
| L1 cache | 32K, 4-way, 32B/line, 3 cycles latency | | |
| L2 cache | 1M, 8-way, 32B/line, 10 cycles latency | | |
| Main memory | 200 cycles latency | | |

Table 2. Architecture modeled.

We compare AccMon to the Purify [15] and CCured [6, 28] (version 1.2.5) tools. Purify instruments the object code at link time and does not require source code changes. It can detect several types of memory-related bugs, including uninitialized reads, writing to freed memory and memory leaks. CCured is a hybrid static and dynamic bug detection tool. It first attempts to enforce a strong type system in C programs via static analysis. The portions of the program that

| Application | Lines of Code | Bug Type | Bug Location | Corrupted Location | Bug Description |
|-----------------|---------------|------------------------|-----------------------------------|---------------------------------|--|
| ncompress-4.2.4 | 1922 | Real-Reported | compress42.c: line 886 | Stack | Input file name longer than 1024 bytes corrupts stack return address |
| linux-simple | 256 | Extracted | based on memory.c:116 | Semantic Bug (No Corruption) | Wrong pointer assignment caused by copy-paste |
| polymorph-0.4.0 | 716 | Real-Reported | polymorph.c: lines 193 and 200 | Stack | Input file name longer than 2048 bytes corrupts stack return address |
| gzip-1.2.4 | 8163 | Real-Reported | gzip.c: line 1009 | Data/BSS | Input file name longer than 1024 bytes overflows a global variable |
| tar-1.13.25 | 27137 | Real-Reported | prepargs.c: line 92 | Heap | Unexpected loop bounds causes heap object overflow |
| man-1.5h1 | 4675 | Real-Reported | man.c: line 998 | Data/BSS | Wrong bounds checking causes static object corrupted |
| bc-1.06 | 17042 | Real-Reported | storage.c: line 176 | Heap | Misuse of bound variable corrupts heap objects |
| | | Real-Unreported | util.c:line 577 | Heap | Overwrite the heap object bounds |
| | | bc-lib: Injected | - | Data/BSS | Data corrupted inside a third-party library |
| | | bc-free: Injected | - | Heap | Access a freed object that may be allocated for other data |

Table 3. Applications and bugs analyzed. “Real-Reported” means that the bug was introduced by the original programmers and has been reported and fixed. “Real-Unreported” means that the bug was introduced by the original programmers but has never been reported before. “Injected” means that the bug was injected by us. “Extracted” means that the bug was extracted from a real program.

cannot be guaranteed by the CCured type system are instrumented with run-time checks to monitor the safety of the execution.

Because CCured requires significant manual changes to an application’s source code to conform to its standard, we have not run all applications with CCured. We modified four applications to run with CCured. For the other applications, we estimate the behavior based on CCured’s functionality, but we cannot predict the overhead. In contrast, AccMon does not require any manual modification of an application’s source code.

We run Purify and CCured on a real machine with a 2.6 GHz Pentium 4 processor, 32-Kbyte L1 cache, 2-Mbyte L2 cache, and 1-Gbyte main memory. Since AccMon runs on a simulator, we cannot compare the absolute execution time of AccMon with that of Purify and CCured. Instead, we compare their execution overheads relative to the runs without any monitoring.

Since existing value-based invariant detection tools such as DIDUCE [14] do not work with C/C++ programs, we cannot quantitatively compare AccMon with DIDUCE. Instead, we carefully evaluated each application to see whether value-based invariants can easily be used to catch the bugs. To be as fair as possible, we even used tricks (such as assuming perfect pointer aliasing knowledge) beyond those envisioned in the papers [11, 12, 14] describing these tools.

5.2. Evaluated Applications

We have conducted two sets of experiments. The first one uses buggy applications to evaluate the functionality and overheads of AccMon for software debugging. The second one further evaluates the overheads of AccMon with bug-free SPEC benchmarks.

For the first set of experiments, we selected seven buggy programs that exhibit a broad spectrum of memory-related bugs. Table 3 gives the details about these applications and their bug characteristics. Some of these applications, such as tar-1.13.25 and bc-1.06, are relatively large, with more than 17,000 lines of code.

The six real buggy programs are from the open-source community. The bugs come with the code and were introduced by the original programmers (except the two injected bugs in bc-1.06). For some programs, we select an older version that had memory-related bugs. The six programs are: *gzip*, *man*, *polymorph*, *ncompress*, *tar*,

and *bc*. *gzip* (GNU zip) is a popular compression utility provided by the GNU project. *man* is a utility in the UNIX family to format and display online manual pages. *polymorph* is a tool to convert Windows’ style file names to something more portable for UNIX systems. *ncompress* is a compression and decompression utility that is compatible with the original UNIX compress utility. *tar* is a tool to create and manipulate tar archives. *bc* is an arbitrary precision numeric processing language.

To demonstrate the unique bug-detection strengths of AccMon, we inject two bugs in bc-1.06. The first, bc-lib, demonstrates the case where a memory object is corrupted by a third-party library whose source code is unavailable. Some programming-rule-based tools, such as CCured or other similar tools, cannot instrument the library to detect the bug. The second, bc-free, is a bug where a dangling pointer dereferences an object that is first freed and then reallocated. Since CCured uses garbage collection to manage memory allocation, this bug will not occur when the code is linked with CCured. Consequently, CCured is unable to detect this bug. However, when the program is not linked with CCured, the bug will re-occur.

We also construct an extracted version of a bug from the latest version of Linux (linux-2.6.6/arch/sparc64/prom/memory.c). This bug is caused by copy-paste and results in an incorrect pointer assignment. The wrong pointer assignment causes incorrect results in some cases. Such copy-paste bugs are common in Linux [5, 22]. Since we cannot run Linux in our simulator, we built a simple benchmark (linux-simple) to measure the effectiveness of AccMon on this type of bugs. Since this bug does not violate any programming rule, it is hard for tools such as CCured and Purify to detect it.

In our experiments, we do not use any specific knowledge about the bugs. Instead, we blindly monitor all global objects, heap objects and stack return addresses for all applications. AccMon can be used in any run (normal or abnormal) to detect potential bugs. To demonstrate AccMon’s capability to detect a bug, we need to use abnormal runs, as do other run-time bug detection studies [6, 11, 12, 14, 28]. To do that, we use bug-exhibiting inputs to generate these abnormal runs. But this does not mean that AccMon needs bug-exhibiting inputs to function.

| Application | AccMon | | Purify | | CCured | | Value-Based Invariants |
|-----------------|---------------|----------|---------------|----------|---------------|----------|------------------------|
| | Bug Detected? | Overhead | Bug Detected? | Overhead | Bug Detected? | Overhead | Bug Detected? |
| ncompress-4.2.4 | Yes | 0.24X | No | 8.33X | Yes | 0.17X | Difficult* |
| linux-simple | Yes | 0.60X | No | 32.84X | No | 5.50X | Difficult |
| polymorph-0.4.0 | Yes | 0.76X | No | 44.65X | Yes | 0.50X | Difficult |
| gzip-1.2.4 | Yes | 0.94X | Yes | 42.45X | Yes | 0.40X | Easy |
| tar-1.13.25 | Yes | 1.04X | Yes | 13.68X | NR(Yes) | NR | Difficult |
| man-1.5h1 | Yes | 1.50X | Yes | 4.83X | NR(Yes) | NR | Easy |
| bc-1.06 | Bug1: Yes | 2.88X | Yes | 46.11X | NR(Yes) | NR | Depends |
| | Bug2: Yes | | Yes | | NR(Yes) | | Difficult |
| | bc-lib: Yes | | No | | NR(No) | | Depends |
| | bc-free: Yes | | Yes | | NR(No) | | Difficult |

Table 4. Overall results. For bc, Bug1 is in storage.c and Bug2 is in util.c. For CCured, NR means that we have not modified the application’s source code to run with CCured; NR(Yes) means that we estimate that CCured should be able to detect the bug if the application were modified to conform to CCured’s requirement; NR(No) means that we estimate that CCured cannot detect the bug based on our knowledge about CCured. *Difficult in column 8 means that we could not find an effective way to detect the bug using value-based invariants.

| Application | Training Overhead | # Monitored Accesses | # Monitored Accesses after the CLB | Monitored Sizes (Bytes) | Ranking of the Bug | # False Alarms |
|-----------------|-------------------|----------------------|------------------------------------|-------------------------|--------------------|----------------|
| ncompress-4.2.4 | 1.20X | 158995 | 13 | 806180 | 1 | 0 |
| linux-simple | 1.64X | 11769 | 5 | 3352 | 1 | 0 |
| polymorph-0.4.0 | 0.99X | 520 | 4 | 10472 | 1 | 8 |
| gzip-1.2.4 | 3.06X | 274594 | 44441 | 396641 | 1 | 0 |
| tar-1.13.25 | 1.52X | 29729 | 102 | 88142 | 2 | 2 |
| man-1.5h1 | 2.83X | 1518 | 90 | 187898 | 1 | 0 |
| bc-1.06 | 3.98X | 260813 | 84716 | 467005 | 1,2,3,4 | 0 |

Table 5. Detailed results for AccMon. The column on number of monitored accesses after the CLB is only for the bug-detection phase. Note that there are four bugs detected for bc.

The second set of experiments evaluates AccMon overheads using three bug-free SPEC2000 applications running the Test input data set, namely gzip, parse and vpr.

6. Experimental Results

6.1. Overall Results

AccMon detects all ten bugs in the seven buggy applications, and found one previously unreported bug (to the best of our knowledge). Table 4 compares the effectiveness and the overhead of AccMon, Purify, CCured, and value-based invariant detection tools. Table 5 shows detailed AccMon results. The default setup for AccMon is a TLS-enabled iWatcher with an 8-entry CLB, and with only write accesses monitored. The results are obtained using this default setup unless otherwise mentioned in Sections 6.2 and 6.3. AccMon’s initialization time to bring the PCT into the cache is also included in AccMon’s overhead. The monitoring in iWatcher is always enabled throughout the entire execution of a tested program (i.e., *DisableMonitoring* is never called).

The evaluation is done in two ways: actual experiments and best-knowledge analysis. If a tool is available, and works with an application, we report the actual experimental results. But if the tool does not target C/C++ programs, or cannot work with an application, we use our best knowledge to estimate whether it can detect the bug or not. However, we cannot estimate its overhead. All results with Purify and AccMon are from actual experiments since these tools work with all applications.

AccMon’s Functionality. From Table 4, we see that AccMon can catch bugs that cannot be detected by other tools such as Purify, CCured and value-based invariant detection tools. While AccMon catches all tested bugs, Purify misses four bugs: ncompress-4.2.4, linux-simple, polymorph-0.4.0 and bc-lib. Purify misses the bugs

in ncompress-4.2.4 and polymorph-0.4.0 because it does not monitor stack accesses. Purify misses the bug in bc-lib because Purify cannot detect wrong pointer arithmetic that results in the corruption of a valid memory object instead of Purify’s “red-zone” (padding inserted by Purify). Purify fails to detect the bug in linux-simple because that bug does not violate any programming rule. Instead, it is just a simple incorrect pointer assignment.

We have modified four applications to run with CCured (ncompress-4.2.4, polymorph-0.4.0, gzip-1.2.4 and linux-simple). Of these four applications, CCured detects the bugs in three of them, but misses the bug in linux-simple because it does not violate any programming rule. For the other applications, we expect CCured to miss the bug in bc-free because CCured uses garbage collection to manage memory allocation (explained in Section 5). Since CCured cannot monitor accesses by a third-party library whose source code is unavailable, we believe that CCured would miss the bug in bc-lib as well. For the other four bugs, we conservatively estimate that CCured would catch them.

Value-based invariant detection tools would miss six of the ten tested bugs because these bugs do not violate any value-based invariant. To ensure a fair comparison, our evaluation with value-based invariant detection tools is very conservative. We even used techniques beyond those described in the previous value-based invariant papers, such as assuming perfect aliasing knowledge.

AccMon’s Overhead. Table 4 shows that AccMon has an acceptable overhead, which is significantly lower than Purify’s. AccMon has an overhead of only 0.24-2.88 times, even though most applications monitor hundreds of KBytes data (Table 5). This is an order of magnitude less than Purify, which has an overhead of 4.83-46.11 times (the Purify results match the numbers reported in [6]). For example, in ncompress-4.2.4, AccMon monitors a total of 0.8 MBytes of memory (Table 5) and almost 92.1% of dynamic memory ac-

cesses (not shown in the tables), but it adds only 24% overhead (Table 4).

For those applications that can run on CCured, AccMon’s overhead is similar to that of CCured. The only exception is linux-simple. CCured has performed very aggressive compiler-based optimizations to reduce the amount of dynamic checks. We believe that AccMon’s overhead can be further lowered with similar compiler-based optimizations. In addition, CCured requires non-trivial modifications to an application’s source code to run. This requirement may not be practical for some programs, especially large server programs.

CCured has a much higher overhead (5.5 times) than AccMon (0.60 times) for linux-simple. The reason is that this program has many accesses to array structures, which cause many dynamic checks to be inserted by CCured. In contrast, AccMon’s CLB hardware effectively filters out most of these memory accesses and leaves a small number of accesses (only 5) to be checked by the run-time system (See Table 5).

AccMon’s False Alarm Rate. AccMon has a very low false alarm rate, and the bugs are ranked high in the error reports. Table 5 shows that there are no false alarms for five applications, and only 2-8 false alarms for two applications. Moreover, all bugs are ranked in the top 2 entries of the error reports. Therefore, a programmer can easily identify real bugs.

6.2. Impact of the CLB

Figure 5 shows the impact of the CLB on AccMon’s overheads, and the sensitivity to the number of entries in the CLB. We compare the overheads in three cases: without CLB (CLB0), with a 4-entry CLB (CLB4) and with an 8-entry CLB (CLB8). The overhead is broken down into two parts: (1) the iWatcherOn/Off overhead (overhead for executing iWatcherOn/Off calls), and (2) the monitoring plus other overhead. Since we support TLS, it is hard to further separate the monitoring overhead from other overhead such as run-time system initialization (bring the PCT into the cache), the effect of instrumentation on compiler optimization, or the effect of resource competition. However, we expect that the monitoring overhead dominates the other overheads for most applications. Table 6 gives the 4-entry and 8-entry CLB hit ratios for all seven applications.

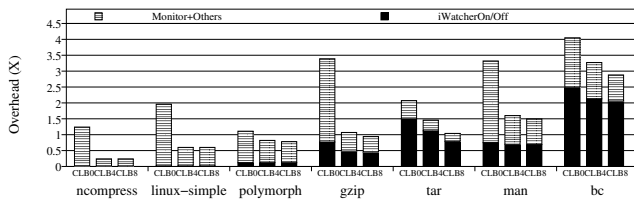


Figure 5. Overhead introduced by AccMon with and without the CLB.

| #Entries | ncompress | linux-simple | polymorph | gzip | tar | man | bc |
|----------|-----------|--------------|-----------|-------|-------|-------|-------|
| 4 | 99.9% | 99.9% | 99.2% | 80.1% | 51.5% | 93.7% | 43.2% |
| 8 | 99.9% | 99.9% | 99.2% | 83.8% | 99.7% | 94.1% | 67.5% |

Table 6. CLB hit ratios for monitored accesses.

Figure 5 shows that the CLB reduces AccMon’s overheads by a significant 28.9-80.6%. For example, the overhead of AccMon with

gzip is reduced by a factor of 3.17 from 3.39 times to 1.07 times with a 4-entry CLB. This is because the 4-entry CLB filters 80% of the triggering accesses in gzip, as indicated in Table 6. Only 20% of the triggering accesses are processed by the AccMon monitoring function. This effect is shown in the 77.5% reduction in the monitor+other overhead given in the breakdown of gzip in Figure 5.

Except in tar and bc, the overhead is reduced only slightly (0-12.1%) for most applications as we go from a 4-entry CLB to an 8-entry CLB. The reason is that the CLB hit ratios only increase slightly (0-3.7%) for these five applications. On the other hand, for tar and bc, an 8-entry CLB reduces the overheads by 28.2% and 11.9%, benefiting from the 48.2% and 24.3% improvement in the CLB hit ratios, respectively.

6.3. Impact of the Optimizations

Monitoring only Write Accesses. AccMon’s overhead is reduced significantly (7.7-61.9%) by monitoring only write accesses instead of all accesses. The rationale is discussed in section 4.4.3. Figure 6 compares the overheads of monitoring both read/write accesses (rw) and write only accesses (wo). Table 7 shows the number of monitored accesses before and after the CLB filtering process for both rw and wo.

In Figure 6, the reduction in overhead as we go from rw to wo comes from reducing the number of monitored accesses. For example, in gzip the number of monitored accesses after the CLB is reduced by 58.5% as we go rw to wo (Table 7), resulting in a 61.9% reduction in overhead (Figure 6).

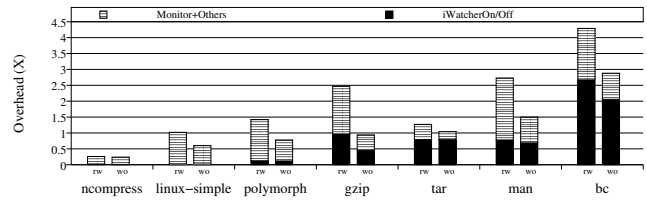


Figure 6. Overhead of monitoring different types of accesses.

| Application | rw (Before CLB) | wo (Before CLB) | rw (After CLB) | wo (After CLB) |
|--------------|-----------------|-----------------|----------------|----------------|
| ncompress | 334019 | 158995 | 27 | 13 |
| linux-simple | 178142 | 11769 | 5 | 5 |
| polymorph | 18658 | 520 | 5 | 4 |
| gzip | 1048300 | 274594 | 107079 | 44441 |
| tar | 107980 | 29729 | 188 | 102 |
| man | 3598 | 1518 | 737 | 90 |
| bc | 782901 | 260813 | 164371 | 84716 |

Table 7. Number of monitored accesses before and after CLB filtering for different types of accesses.

In ncompress, linux-simple, and polymorph, going from rw to wo induces a very small absolute decrease in the number of monitored accesses after the CLB (Columns 4 and 5 of Table 7). However, linux-simple and polymorph show a significant overhead reduction in Figure 6. The reason is that going from rw to wo causes a significant reduction of monitored accesses before the CLB for these applications (Table 7). Since the PCT of an application is generated based on all monitored accesses before the CLB, the size of the PCT is significantly reduced from rw to wo for these two applications. As a result, the overhead of bringing the PCT into the

cache (part of other overhead) is reduced significantly, resulting in a similar reduction in the total overhead.

Compiler-Based False Alarm Pruning. The compiler optimization that differentiates pointer/array accesses from other accesses is effective at pruning false alarms. As shown on Table 8, this optimization reduces the number of false alarms in tar-1.13.25 from 8 to 2. However, this optimization fails for polymorph-0.4.0, because the bug causes the program to enter an error handler that is never entered in normal execution, resulting in eight false alarms that are caused by the pointer/array accesses inside the handler.

| | ncom-press | linux-simple | poly-morph | gzip | tar | man | bc |
|----------------|------------|--------------|------------|------|-----|-----|----|
| Before Pruning | 1 | 0 | 8 | 1 | 8 | 0 | 0 |
| After Pruning | 0 | 0 | 8 | 0 | 2 | 0 | 0 |

Table 8. Number of false alarms before and after pruning.

6.4. Overhead with SPEC Benchmarks

To measure AccMon overheads on bug-free applications, we run three SPEC2000 benchmarks, namely gzip, parser and vpr¹, with the Test input data set. The experiments use the default setup for AccMon: TLS-enabled, 8-entry CLB and only monitoring write accesses.

Table 9 shows the results for the three applications. The overheads are 1.29, 3.16 and 1.73 times for gzip, parser and vpr respectively. The size of monitored memory is 6.5-13.5 MBytes. Recall that the overhead is broken down into iWatcherOn/Off overhead and monitoring plus other overhead. For all the three applications, the iWatcherOn/Off overhead is a substantial portion of the total overhead. The large iWatcherOn/Off overhead is mainly the result of watching the locations for return addresses. In this case, both iWatcherOn and iWatcherOff are invoked once per function call.

The monitoring overhead is related to the number of monitored accesses per 1M instructions after CLB filtering. The CLB hit ratios for all three applications are high: 92.2%, 99.4% and 83.3% for gzip, parser and vpr, respectively. Therefore, many accesses are filtered by the CLB, especially for parser. This significantly reduces the monitoring overhead which, together with other overheads (described in Section 6.2), accounts for the non-iWatcherOn/Off component of the total overhead.

| Application | Overhead | iWatcherOn/Off Overhead | # Monitored Accesses per 1M Inst. after CLB | Monitored Sizes (Bytes) |
|-------------|----------|-------------------------|---|-------------------------|
| gzip | 1.29X | 0.80X | 5699.1 | 13533869 |
| parser | 3.16X | 2.13X | 77.5 | 10244523 |
| vpr | 1.73X | 0.95X | 7563.0 | 6585702 |

Table 9. AccMon behavior for SPEC applications.

7. Related Work

Our work builds upon many previous studies on improving software robustness. Due to lack of space, we only briefly describe some closely related work that is not described in previous sections.

¹For parser, we fast forward the program’s initialization phase, which lasts for about 280 million instructions, because its behavior is not representative of steady state. To reduce simulation time, for both parser and vpr, we only run them for 300 million instructions.

Many tools have been proposed for dynamic execution monitoring. Well-known examples include Eraser [32], StackGuard [7], Valgrind [34] and others [1, 29], besides those discussed in previous sections. StackGuard only detects attacks against stack return addresses — not general memory-related bugs. Eraser targets multithreaded programming, and detects data races in lock-based multithreaded programs. Valgrind is a dynamic checker to detect general memory-related bugs such as memory leaks, memory corruption and buffer overflow. It simulates every single instruction of a program, so it incurs a significant 10-20 times overhead [40].

As discussed in Section 1, most dynamic bug detection methods can be classified into two types: programming-rule-based (PRB) and statistics-rule-based (SRB). These two are not competing techniques. Instead, they complement each other since both offer unique advantages that can be integrated to detect a wider range of bugs. Since both approaches focus on different types of rules, the types of bugs caught by them often differ. For example, a wrong pointer assignment bug caused by copy-paste does not violate any PRB rules, but may violate a SRB rule, such as a PC-based invariant. However, SRB usually requires inferring rules from normal runs, which may not always be possible. Therefore, PRB is more useful for catching relatively simple bugs that obviously violate programming rules, whereas SRB is more applicable to detecting those “silent” bugs that successfully pass through many regression tests before the software is released. These regression tests allow statistical rules to be extracted.

Our work is also related to research on static analysis and model checking [9, 10, 13, 26]. These methods usually require program annotations or specification of invariants, and are commonly limited by pointer aliasing and other compile-time limitations. MC checker [9, 10] extracts beliefs from system code and applies them to bug detection. This is an application of the invariants-based method to static analysis.

Besides iWatcher, AccMon can also use software-based instrumentation tools such as ATOM [36] or Dyninst [16], hardware watchpoints [17, 18, 35, 37], or other tools [8]. However, we expect that these tools would result in significant overheads. In addition, it is possible to use special hardware [38] that provides fine-grain access control to monitor memory accesses in AccMon. We use iWatcher for the reasons given in Section 1 and 2.2.

Our work is also related to address profiling techniques for performance optimization. Calder et al proposed a data placement strategy based on temporal relations by profiling memory accesses [4]. Barrett et al used address profiling to predict the life time of heap variables and then used this information to reduce the memory page fault rate [2]. In our work, we monitor memory accesses to detect software bugs.

There are several works that use Bloom filters in hardware. They use a Bloom filter to minimize load/store queue (LSQ) searches [33], to identify cache misses early in the pipeline [30], and to filter cache-coherence traffic in snoopy bus-based SMP systems to reduce energy consumption [25].

8. Conclusions and Future Work

This paper made two contributions. First, it proposed the novel idea of PC-based invariants to detect memory-related bugs. Second, it proposed the CLB, a new architectural extension to the iWatcher

framework that significantly reduces the overhead of PC-based invariant debugging. We demonstrated our ideas with a debugging tool called AccMon. AccMon leverages architectural, run-time system and compiler support. It detects all tested bugs with few false alarms (0 for five applications and 2-8 for two applications) and low overheads (0.24-2.88 times). The latter is an order of magnitude smaller than Purify. Since AccMon is a statistics-based approach, it can catch bugs that do not violate any programming-based rules. For example, there are 3-4 bugs in our experiments that are detected by AccMon but are missed by other tested tools such as Purify [15] and CCured [28, 6].

There are several possible directions for future work. First, due to the limitations of our simulation infrastructure, we cannot run tests with long-running server programs. Consequently, we are in the process of extending the simulation infrastructure to support Linux and long-running server programs. Second, we will combine AccMon with value-based invariants [11, 12, 14] to build a more comprehensive tool. Third, it is possible to use a software instrumentation tool to monitor memory accesses, detect PC-based invariants and then check for violations. However, we expect this approach to have a much higher overhead because, due to aliasing problems, we may have to monitor more memory accesses than the current approach. Finally, while AccMon uses the invariant of a set of PCs that access an object, it may be also helpful to capture the invariant of a set of objects accessed by a PC. Note that this method may not always work: in a loop that processes objects one-by-one from a linked list, a given PC is accessing many objects, whereas a given object is accessed by only a few PCs.

9. Acknowledgments

We thank the anonymous reviewers for useful feedback, and the UIUC PROBE and Opera groups for useful discussions.

References

- [1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 290–301, 1994.
- [2] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 187–196, 1993.
- [3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
- [4] B. Calder, K. Chandra, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, 1998.
- [5] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *Symposium on Operating Systems Principles*, pages 73–88, 2001.
- [6] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 232–244, 2003.
- [7] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, Jan 1998.
- [8] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Wehl, and G. Z. Chrysos. Profileme: Hardware support for instruction-level profiling on out-of-order processors. In *International Symposium on Microarchitecture*, pages 292–302, 1997.
- [9] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, Oct 2000.
- [10] D. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 57–72. ACM Press, 2001.
- [11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, 1999.
- [12] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *International Conference on Software Engineering*, 2000.
- [13] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, 2002.
- [14] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, May 2002.
- [15] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Winter Technical Conference*, 1992.
- [16] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Scalable High Performance Computing Conference*, 1994.
- [17] Intel Co. The IA-32 intel architecture software developer’s manual, volume 2: Instruction set reference. Intel.
- [18] M. S. Johnson. Some requirements for architectural support of software debugging. In *Proceedings of the first International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-I)*, 1982.
- [19] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging*, 1997.
- [20] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, 1994.
- [21] S.-I. Lee, T. A. Johnson, and R. Eigenmann. Cetus - an extensible compiler infrastructure for source-to-source transformation. In *Proceedings of the 16th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC '2003)*, 2003.
- [22] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation*, 2004.
- [23] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, 2003.
- [24] E. Marcus and H. Stern. Blueprints for high availability. John Wiley and Sons, 2000.
- [25] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. Jetty: Filtering snoops for reduced energy consumption in SMP servers. In *Proceedings of the Seventh International Symposium on High Performance Computer Architecture (HPCA-7)*, 2001.
- [26] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Dec. 2002.
- [27] National Institute of Standards and Technology (NIST), Department of Commerce. Software errors cost U.S. economy \$59.5 billion annually. NIST News Release 2002-10, 2002.
- [28] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2002.
- [29] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software-Practice & Experience*, 27(1), 1997.
- [30] J.-K. Peir, S.-C. Lai, and S.-L. Lu. Bloom filtering cache misses for accurate data speculation and prefetching. In *Proceedings of the 16th Annual ACM International Conference on Supercomputing (ICS)*, 2002.
- [31] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, June 2003.
- [32] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
- [33] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable hardware memory disambiguation for high ILP processors. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO-36)*, Dec 2003.
- [34] J. Seward. Valgrind, an open-source memory debugger for x86-GNU/Linux. available at URL <http://developer.kde.org/~sewardj/>.
- [35] SPARC International. *The SPARC architecture manual: Version 8*. Prentice-Hall, 1992.
- [36] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI)*, June 1994.
- [37] R. Wahbe. Efficient data breakpoints. In *Proceedings of the fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, 1992.
- [38] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, Oct 2002.
- [39] M. Xu, R. Bodik, and M. D. Hill. A “flight data recorder” for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, 2003.
- [40] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA)*, 2004.