ARCHITECTURAL SUPPORT FOR SOFTWARE DEBUGGING

BY

PIN ZHOU

B.E., Tsinghua University, 1997
M.E., Tsinghua University, 1999

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Abstract

As Moore's law has been continuously improving the microprocessor's speed, performance is no longer the only focus. Software robustness has become one of the increasingly important issues. However, recent impressive advances in computer architecture have not led to significant improvement in software robustness. Since software robustness is mainly affected by software bugs, the focuses of this research are to provide efficient and simple architectural support to improve dynamic monitoring for detecting memory-related bugs, and to propose a new bug detection method and an incremental consistency check framework that both leverage the proposed architectural support.

In this dissertation, we propose the *Intelligent Watcher (iWatcher)*, a novel architectural scheme to monitor dynamic execution *automatically*, *flexibly* and with *minimal overhead*. iWatcher associates program-specified monitoring functions with memory locations. When any such location is accessed, the monitoring function is automatically triggered with low overhead. To further reduce overhead and support rollback, iWatcher can optionally leverage Thread-Level Speculation (TLS). The experimental results with seven buggy applications (with various bugs) show that iWatcher detects all the bugs evaluated in our experiments with only a 0.1-179% execution overhead.

We also propose a new statistics-based method, called *program counter (PC)-based invariance*, to detect memory-related bugs *on the fly*, and a simple architectural extension, called the*Check Look-aside Buffer (CLB)*, that takes advantage of the bloom filter and the temporal object access locality to reduce the monitoring overhead in iWatcher. The PC-based invariance idea captures the invariant of the set of PCs that normally access a given key variable, and detects accesses by outlier instructions that are often caused by memory corruption, buffer overflow, stack smashing or other memory-related bugs. we build 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. AccMon leverages the iWatcher framework with the CLB extension to monitor accesses to key variables. 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. AccMon also has low overhead (0.24-2.88 times), which is an order of magnitude lower than Purify.

We also use the binary instrumentation tool PIN to build a pure software implementation of PC-based invariant detection called *AccMon-S*. AccMon-S does not require hardware support, but has much higher execution overheads (10.4-57.8 times), so it can only be used for in-house bug detection instead of bug detection during production runs. Besides detecting all ten bugs tested in AccMon, AccMon-S also detected two real bugs in two large real-word server applications, Apache and Squid, with few false alarms (0-4).

We also present an incremental checking framework, called iChecker, that leverages iWatcher to provide an iChecker library for efficient, incremental, run-time consistency checks of mutable data structures in C programs. The basic idea of iChecker is to perform a consistency check with a local check (on the parts that need to be checked due to the modifications since the last consistency check) instead of with a global check. The evaluation using four case studies shows that iChecker reduces the checking overhead by 1.1–155 times (23.3 on average) over global checks for large data structures. The required code modifications for iChecker are 25–108 lines (including the global checkers), which are 10–56 lines more than the modifications for traditional global checks.

Acknowledgments

First and foremost, I would like to thank my advisor, Prof. Yuanyuan Zhou, who has spent a considerable amount of time and effort to advise, encourage and inspire me during the four years I have worked with her. She has taught me how to do research step by step, from presenting an idea to writing the entire paper, from solving a specific problem to finding an important problem. The things I have learned from her not only benefited me during my Ph.D. period, but also will be invaluable in my future career. Yuanyuan has helped me identify my strengths and unleash my potentials. Her trust and encouragement has helped drive me to achieve my goals. Without her guidance, patience, and inspiration, this dissertation would not have been possible.

I also owe particular gratitude to my committee, Prof. Josep Torrellas, Prof. Marc Snir, and Prof. Darko Marinov. I have been very fortunate to work with Josep Torrellas and Darko Marinov in a few projects and have their close guidance on my Ph.D research. They have shared their ideas and experiences with me, provided advice on writing and presenting, and more importantly broadened my knowledge in different research areas. I thank Marc Snir for his careful reading and useful comments on this dissertation, and valuable discussions on my research.

I enjoyed working with many outstanding graduate students in our Opera group. I am grateful to all the Opera members for their intellectual discussions on various kinds of topics, creating a enjoyable and helpful group atmosphere, and making my Ph.D life more pleasant. I thank Shan Lu, Feng Qin, Zhenmin Li, Vivek Pandey, Anand Raghuraman and Jagadeesan Sundaresan, who have collaborated with me and made our projects successful. I thank Joseph Tucek for helping me with my writing.

I would like to thank many other faculty members at UIUC, in particular Prof. Sarita Adve for many insightful comments and discussions, and Prof. Jeniffer Hou for recommending me to my advisor. I should also thank my fellow graduate students in other groups, to name just a few, Ruchira Sasanka, Jose Renau and Milos Prvulovic for their technical discussions and help.

I wish to thank many friends for their support and encouragement. An incomplete list of them is Yuan Sun, Min Xu, Xue Liu, Wenbo He, Hui Ding, Xia Li, Marvel Ma, Zhifeng Chen, Qingbo Zhu, Weihang Jiang, Lin Tan, Shan Lu, Feng Qin, Zhenmin Li, Min Wu, Qixin Wang, Yi Cui and Yuan Xue. They have made my life much more wonderful and joyful.

Last but not least, I want to thank my family. Words are just not enough to express my thanks to them for their unconditional support and true love. I would like to thank my dear Mom and Dad for teaching me to be a good person which is far more important than any professional achievement and for providing me the best education they could. My husband, Wei, is the best companion that I have ever had in the world. He is not only a wonderful husband, but also a good friend, classmate, and collaborator. I have learned and been inspired a lot from all kinds of technical discussions with him. His endless support and encouragement has given me the power to conquer the difficulties I have encountered. My daughter, Maggie, is the most precious gift from heaven who has brought me the most happiness and has occupied the most important place in my heart. I would also like to thank my parents-in-law, who have always been understanding and very supportive, and thank my brother, Kun, who has taken full responsibility of taking care of my parents since I came to US. Without them, I would not have been able to go as far as I have.

To my parents, Pirong and Zibi

To my husband, Wei

To my daughter, Maggie

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software bugs significantly affect system reliability, availability and security. Software debugging often relies on inserting run-time software checks. In many cases, however, program execution typically slows down significantly, often by 10-100 times, and it is hard to find the root cause of a bug. Moreover, most dynamic checkers cannot find those hard-to-find program-specific bugs.

Recent impressive performance improvements in computer architecture have not led to significant gains in the case of software debugging. While recent work on architectural support for software debugging provides a good foundation, it is still far from providing a complete solution.

This dissertation work takes another step toward the goal of improving software debugging using architectural support. It proposes a novel architectural framework *iWatcher* for dynamic monitoring. It proposes a new statistics-based bug detection method, and builds an automatic detection tool *AccMon* for memory-related bugs using this method and leveraging the iWatcher framework. Finally, it also proposes an incremental checking framework *iChecker* for data structure consistency check in C programs, again leveraging the iWatcher framework.

1.1 Motivation

As Moore's law has been continuously improving the microprocessor's speed, performance is no longer the only focus. Instead, other issues, such as software robustness, hardware reliability and low power/energy efficient design, etc., are becoming increasingly important.

Software robustness problems are mainly caused by software bugs, because bugs significantly affect system reliability, availability and security. The increasing software complexity causes more software bugs and also makes them harder to detect. Despite costly efforts to improve software-development methodologies, software bugs in deployed programs continue to thrive, often accounting for up to 40% of computer system failures [MS00]. Software bugs can crash systems, making services unavailable or, in the form of “silent” bugs, corrupt information or generate wrong outputs. According to NIST [Nat02], software bugs cost the U.S. economy an estimated $59.5 billion annually, or 0.6% of the GDP!

There are different types of software bugs, such as memory-related bugs, concurrency bugs, etc. Among them, memory-related bugs are the most security critical and the most common ones. Memory-related bugs are caused by improper handling of memory objects and also major causes of security problems. Attackers exploit memory bugs to execute malicious code on otherwise safe computers, steal confidential information, or deplete the service by crashing or overloading it. Based on CERT advisories, in 2001-2004, more than 50% of security vulnerabilities are caused by memory-related bugs. Therefore, detecting memory bugs is very important for finding the security vulnerabilities, particularly in programs written in an unsafe language such as C/C++, and it is also the focus of this work.

Memory-related bugs can be further classified into: (1) Buffer overflow: Illegal access beyond the buffer boundary. (2) Stack smashing: Illegally overwrite the function return address. (3) Memory leak: Dynamically allocated memory have no reference to it, hence can never be freed. (4) Uninitialized read: Read memory data before it is initialized. The reading result is illegal. (5) Double free: One memory location freed twice. (6) Some program-specific (semantic) bugs: Bugs that are inconsistent with the original design and the programmers’ intention, and also caused by improper handling of memory object, such as wrong assignment. As we will see later, the bug in linux-simple benchmark in chapter 4 is an example of such a bug, caused by copy-pasting.

1.1.1 Existing Software Debugging Techniques

Current debugging techniques consist largely of interactive debuggers, static checking, and dynamic monitoring, all of which have significant limitations. Interactive debuggers, such as gdb and the debug tool embedded in Microsoft Visual Studio, are widely used by most programmers to find bugs. Since programmers know about the program, they can rely on the interactive debugger to find program-specific bugs by hand. However, since running programs in a debugger is very slow, and the programmer needs to manually examine the execution, interactive debugging is very time-consuming, and requires large human effort and experience. Moreover, it is very hard to reproduce a bug, because the bug may occur only after hours or even days of execution, only with a particular combination of user input and/or hardware configurations, or only with a particular interleaving of timing-related events.

Static checking performs checks statically. Examples of static approach include explicit model checking [MPC+02, SD95] and program analysis [CLL+02, EA03, HCXE02]. Most static tools require significant programmer involvement to write specifications or annotate programs. In addition, most static tools are limited by aliasing problems and other compile-time limitations. This is especially the case for programs written in unsafe languages such as C or C++, the predominant programming languages in industry. As a result, many bugs often remain in programs even after aggressive static checking.

Dynamic monitoring monitors the execution and checks for rule or invariant violations. It can be classified into two categories: programmer-specified monitoring and automatic monitoring. Assertion and data structure consistency checks fall into programmer-specified monitoring, because they require programmers to provide checks. In this work, consistency of a data structure means that the states of the data structure satisfy certain properties during the entire program execution except within some operations that intentionally violate the properties while evolving the data structure from one consistent state to another. A traditional consistency check usually needs to traverse the entire data structure to determine that the consistency properties hold, and is very expensive.

Many dynamic monitors belong to the automatic monitoring category, including Purify [HJ92], Valgrind [NS03], Intel Thread Checker [KAI], DIDUCE [HL02], Eraser [SBN+97], CCured [CHM+03, NMW02], and other tools [ABS94, CPM+98, LYHR01, PF97, PF95]. The strength of dynamic approach is that the analysis is based on actual execution paths and accurate values of variables and aliasing information.

Unfortunately, most dynamic checkers suffer from one or more of the following three limitations. First, inefficiency: they are often computationally expensive. One major reason is their large instrumentation cost [HL02, SBN$^+$97], since almost all existing dynamic checkers insert instrumentation in the code for checking rule or invariant violations. Another reason is that dynamic checkers that attempt to check accesses to certain locations may end up instrumenting more places than necessary due to lack of accurate information at instrumentation time for languages like C/C++. As a result, some dynamic checkers slow down a program by 6-30 times [HL02, SBN$^+$97], which makes such tools undesirable for production runs. Moreover, some timing-sensitive bugs may never occur with these slowdowns.

Second, inaccuracy: most dynamic checkers rely on compilers or pre-processing tools to insert instrumentation for checking accesses to certain locations, and, therefore, are limited by imperfect variable disambiguation, especially for C/C++. Consequently, some accesses to a monitored location may be missed by the instrumentation tool. Because of this reason, some bugs are caught much later than when they actually occur, which makes it hard to find the root cause of the bug.

Third, limited bug coverage: many dynamic checkers can detect only those bugs that violate some basic programming rules such as "an array pointer cannot move out-of-bounds", and fail to detect other bugs that are specific to the monitored software. For example, a forget-to-change bug (see the linux-simple benchmark in Chapter 4) caused by copy-pasting in the latest version of Linux can result in an incorrect pointer assignment to the wrong location. This bug does not violate any programming-based rules and thereby cannot be detected by most existing dynamic checkers.

This work focuses on addressing the above three limitations of dynamic monitoring.

1.1.2 Architectural Support for Software Debugging

As micro-architectural innovations have significantly improved performance, interest has recently risen in the architecture community to use transistors to improve software debugging. However, the current state of the art is very primitive, largely limited to watchpoints [Int04, SPA92] and event or branch trace buffers [Int04, Spr02].

Watchpoints, such as those supported by Intel's x86 [Int04] and Sun's SPARC [SPA92], trigger an exception every time that a programmer-specified memory location is accessed. While they are a good starting point, they have several limitations. First, they do not provide *low-overhead* checks that can be on *all the time* in a production run. This is because they trigger the exception mechanism, which has very high overhead and disrupts the execution of the application. Second, current architectures only support a handful of watchpoints (four in Intel x86).

Besides watchpoints, branch or event trace buffers [Int04, Spr02] can also potentially be used for debugging purposes, such as providing more program state information in a crash. However, they do not provide highly-processed information that could truly boost debuggability.

Recently, there have been some research proposals for micro-architectural support for software debugging. For example, Prvulovic and Torrellas proposed ReEnact [PT03], 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" [XBH03], which enables off-line deterministic replay and can be used for postmortem analysis of a bug. BugNet [NPC05] proposed by Narayanasamy et al. further reduces the amount of data recorded for deterministic replaying of an execution. Tuck et al. uses hardware and binary rewriting to protect pointers from buffer overflow [TCV04]. While recent work is promising and provides a good foundation, they provide relatively limited functionality (i.e., handle only certain types of bugs or provide only a trace) and are also relatively expensive, which indicates architectural support for software debugging is still far from providing a complete solution. This dissertation work takes another step toward the goal of improving software debugging using architectural support.

Novel architectural support would provide several benefits for improving software debugging over software-only solutions: (1) *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. (2) *Accuracy*: Architectural support can avoid pointer aliasing problems and accurately capture all desired accesses to monitored memory objects. (3) *Generality*: 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.

1.2 My Dissertation Work

My thesis is that using architectural support and novel techniques can improve the three main aspects of dynamic monitoring: efficiency, accuracy, and coverage.

This dissertation work provides novel, general and simple architectural support for dynamic monitoring, proposes a new bug detection method to catch those hard-to-catch program-specific memory bugs, and proposes an incremental consistency check framework which both leverage the proposed architectural support to reduce overhead. I made the following contributions in my work:

1. This work proposes the *Intelligent Watcher (iWatcher)*, an efficient and flexible architectural scheme to monitor dynamic execution *automatically*, *flexibly* and with *minimal overhead*. iWatcher associates program-specified monitoring functions with memory locations. When any such location is accessed, the monitoring function is automatically triggered with low overhead. To further reduce overhead and support rollback, iWatcher can optionally leverage Thread-Level Speculation (TLS). The experimental results with seven buggy applications (with various bugs) show that iWatcher detects all the bugs evaluated in our experiments with only a 0.1%-179% execution overhead. Overall, iWatcher's reasonably small overhead and ability to monitor many memory locations enable it to be used during both in-house testing and production runs.

2. This work proposes a new statistics-based method, called *program counter (PC)-based invariant*, to detect memory-related bugs *on the fly*, and a simple architectural extension, called

the *Check Look-aside Buffer (CLB)*, that uses a Bloom filter [Blo70] and takes advantage of the good temporal locality that exists in object accesses to reduce the monitoring overhead in iWatcher. The PC-based invariant idea captures the invariant of the set of PCs that normally access a given key variable, and detects accesses by outlier instructions that are often caused by memory corruption, buffer overflow, stack smashing or other memory-related bugs. It also builds 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. AccMon leverages the iWatcher framework with the CLB extension to monitor accesses to key variables. 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. AccMon also has low overhead (0.24-2.88 times), which is an order of magnitude lower than Purify [HJ92].

3. This work also uses the binary instrumentation tool PIN [LCM$^+$05] to build a pure software implementation of PC-based invariant detection called *AccMon-S*. AccMon-S does not require hardware support, but has much higher execution overheads (10.39-57.83 times), so it can only be used for in-house bug detection instead of bug detection during production runs. Besides detecting all ten bugs tested in AccMon, AccMon-S also detected two real bugs in two large real-word server applications, Apache and Squid, with few false alarms (0-4).

4. This work presents an incremental checking framework, called iChecker, that leverages iWatcher to provide a library for efficient, incremental, run-time consistency checks of mutable data structures in C programs. The idea of iChecker is to perform a consistency check with a local check (on the parts that need to be checked due to the modifications since the last consistency check) instead of with a global check. The programmer only needs to indicate the data structure to be checked and its associated local check function, and call a few library

calls in limited places. It is iChecker's responsibility to figure out on which portions of the data structure to perform this local check function. The evaluation using four case studies shows that iChecker reduces the checking overhead by 1.1–155 times (23.3 on average) over global checks for large data structures with 0.3–17.9 times (11.2 on average) space overhead. The required code modifications for iChecker are 25–108 lines (including the global checkers), which are 10–56 lines more than the modifications for traditional global checks and only account for 0.1%–21% of the original code.

1.3 Outline

The remainder of this dissertation is organized as follows. Chapter 2 discusses background and related work. Chapter 3 proposes iWatcher for dynamic monitoring. Chapter 4 presents the PC-based invariant detection idea, and the automatic tools AccMon (hardware support) and AccMon-S (pure software) for detecting memory-related bugs. Chapter 5 presents iChecker, the incremental check framework for mutable data structure consistency. Chapter 6 summarizes my current work and outlines the future research plans.

The materials in some chapters have been published as journal and conference papers. Some materials in Chapter 3 have been presented in [ZQL$^+$04] and [ZZQ$^+$05]. The materials in Chapter 4 have been presented in [ZLF$^+$04].

Chapter 2

Background and Related Work

This chapter discusses background and previous work related to our work.

2.1 Background

2.1.1 Dynamic Execution Monitoring

Many methods have been proposed for dynamic code monitoring. The most commonly used ones are assertions, dynamic checkers, and watchpoints.

Assertions Assertions are inserted by programmers to perform sanity checks at certain places. If the condition specified in an assertion is false, the program aborts. Assertions are one of the most commonly used methods for debugging. However, they can add significant overhead to program execution. Moreover, it is often hard to identify all the places where assertions should be placed.

Dynamic Checkers Dynamic checkers are automated tools that detect common bugs at run time. For example, DIDUCE [HL02] automatically infers likely program invariants, and uses them to detect program bugs. Purify [HJ92] and Valgrind [NS03] monitor memory accesses to detect memory leaks and some simple instances of memory corruption, such as freeing a buffer twice or reading an uninitialized memory location. StackGuard [CPM$^+$98] can detect some buffer overflow bugs, which have been a major cause of security attacks. Eraser [SBN$^+$97] can detect data races by dynamically tracking the set of locks held during program execution. These tools usually use compilers or code-rewriting tools such as ATOM [SE94], EEL [LS95] and Dyninst [BH00] to instrument programs with checks.

While this approach is promising, dynamic checkers often suffer from the following limitations: (1) aliasing problems, especially in C or C++ programs, (2) high run-time overhead, (3) hard-coded bug detection functionality, (4) language specificity, and (5) difficulty to work with low-level code.

Hardware-Assisted Watchpoints Hardware-assisted watchpoints [Int01, KH92, SPA92] use simple hardware support to watch a user-selected memory location. When a watched location is accessed by the program, an exception is handled by an interactive debugger such as gdb. Then, the state of the process can be examined by programmers using the debugger. The hardware support is provided through a few special debug registers. Watchpoints are designed to be used in an interactive debugger. For non-interactive execution monitoring, they are both inflexible and inefficient. They do not provide a way to associate an automatic check to the access of a watched location. Moreover, they require an expensive exception when a watched location is accessed. Finally, most architectures only support a few watchpoints (four in Intel's x86).

2.1.2 Classifying Dynamic Monitoring Methods

We classify the dynamic monitoring methods into two categories:

- *Code-Controlled Monitoring (CCM).* Monitoring is performed only at special points in the program. Assertions and most dynamic checkers belong to CCM because they only check at assertions or instrumentation points.

- *Location-Controlled Monitoring (LCM).* Monitoring is associated directly with memory locations and therefore all accesses to such memory locations are monitored. Hardware-assisted watchpoints and iWatcher belong to this category.

If we want to monitor the accesses to particular memory locations, which is a common technique used for bug detection, LCM has two advantages over CCM: (1) LCM monitors *all* accesses to a watched memory location using all possible variable names or pointers, whereas CCM may miss some accesses because of pointer aliasing; (2) LCM monitors only those memory instructions that *truly* access a watched memory location, whereas CCM may need to instrument at many unnecessary points due to the lack of accurate information at instrumentation time. Therefore, LCM can be used to detect both invariant violations and illegal accesses to a memory location, whereas it may be difficult and too expensive for CCM to check for illegal accesses. The main advantage of CCM is that it does not require hardware support while LCM typically needs it. In the case that we want to perform monitoring at specific execution points, CCM will be enough.

2.1.3 Invariant-Based Bug Detection

Similar to previous invariant-based bug detection work such as DAIKON [ECGN99, ECGN00] and DIDUCE [HL02], 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.

2.1.4 Thread-Level Speculation (TLS)

TLS is an architectural technique for speculative parallelization of sequential programs [CMT00, SBV95, SCZM00, THA+99]. TLS support can be built on a multithreaded architecture, such as simultaneous multithreading (SMT) or chip multiprocessor (CMP) machines. With TLS, the execution of a sequential program is divided into a sequence of *microthreads* (also called tasks, slices, or epochs). These microthreads are then executed speculatively in parallel, while special hardware detects violations of the program's sequential semantics. A violation results in squashing the incorrectly executed microthreads and re-executing them. To enable squash and re-execution, the memory state of each speculative microthread is typically buffered in caches or special buffers. When a microthread finishes its execution and becomes safe, it can commit. Committing a microthread implies merging its state with the safe memory. To guarantee sequential semantics, microthreads commit in order.

iWatcher can leverage TLS to reduce monitoring overhead and to support rollback and re-execution of a buggy code region [PT03]. For our design, we assume an SMT machine, and that the speculative memory state is buffered in caches. However, our iWatcher design can be easily ported to other TLS architectures.

If we use TLS in iWatcher, each cache line is tagged with the ID of the microthread to which the line belongs. Moreover, for each speculative microthread, the processor contains a copy of the initial state of the architectural registers. This copy is generated when the speculative microthread is spawned and is freed when the microthread commits. It is used in case the microthread needs to be rolled back.

The TLS mechanisms for in-cache state buffering and rollback can be reused to support incremental rollback and re-execution of the buggy code [PT03]. To do this, the basic TLS is modified slightly by postponing the commit time of a successful microthread. In the basic TLS, a microthread can commit when it completes and all its predecessors have committed. We say that such a microthread is *ready*. To support the rollback of buggy code, a ready microthread commits only in one of two cases: when we need space in the cache and when the number of uncommitted microthreads exceeds a certain threshold. With this support, a ready but uncommitted microthread can still be asked to rollback. This feature can be used to support one of the iWatcher modes (Section 3.3.5).

2.2 Related Work

2.2.1 Dynamic Monitoring

Many tools have been proposed for dynamic execution monitoring. Well-known examples include Purify [HJ92], Intel thread checker [KAI], Eraser [SBN$^+$97], StackGuard [CPM$^+$98], DIDUCE [HL02], Valgrind [NS03], CCured [CHM$^+$03, NMW02], and many others [ABS94, LYHR01, PF97, PF95]. StackGuard only detects attacks against stack return addresses — not general memory-related bugs. Eraser and Intel thread checker target multithreaded programming, and detect data races in multithreaded programs. SafeC [ABS94] presents a pointer and array access checking technique. By using a novel safe pointer structure and adding monitor instructions, most spatial and temporal access errors can be detected. It has the limitations of the programming-rule-based approach as mentioned below.

Most of these tools rely on instrumentation to perform dynamic checks. Consequently, to check all possible accesses to a given location, they typically need to instrument more than necessary. Moreover, most dynamic checkers impose significant run-time overhead. Our work innovates with general, efficient and flexible location-controlled monitoring capability.

As discussed in Chapter 4, 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.

Schnarr and Larus have proposed using unused processor cycles to reduce overhead for *code-controlled* monitoring [SL96]. Our work differs from theirs in that iWatcher provides convenient, flexible architectural support to perform *location-controlled* monitoring, and uses TLS to hide monitoring overheads.

Oplinger and Lam have used TLS to hide the overhead for dynamic monitoring [OL02]. iWatcher also exploits the benefits of recently proposed TLS architecture. However, in iWatcher, the thread spawning is automatically done by the hardware, whereas their study uses compilers to insert the thread-spawning into the programs.

2.2.2 Data-Consistency Checking

A closely related work to ours is that of Demsky et al. [DR03, DR05]. They propose a specification-based approach that automatically translates a programmer-provided specification of consistency properties into a global consistency checks (and repair code) for C/C++ programs. Our work differs in that we require the programmer to provide a local checking function written in C instead of a specification written in a different language. Moreover, our framework automatically supports incremental checks to reduce overhead and their work still performs expensive global checks.

Previous studies on consistency check also appeared in other domains, for example database systems [CFPT94, UD90], file systems, and operating systems [GJKW97, MA87]. These works provide application-specific solutions, whereas ours provides a general framework that can be used for almost any C programs.

2.2.3 Incremental Computation

Incremental computation computes the new output (for a new input) *incrementally* by reusing parts of the old computation (for an old input), instead of recomputing the entire output from the scratch. Incremental computation works well when a small change in input implies a small change in output, and there is only a small change in input. A widely used approach for incremental computation is based on the dependency graph of the computation [DRT81, YS88, ABH02]. The previous work proposed some foundational techniques but applied them in the context of attribute grammars [DRT81], specialized new languages [YS88] (that, for example, have no recursion or loops) or functional languages [ABH02].

Another approach for incremental computation is to generate the incremental code from the original code [LT95, ZL98]. Liu and Teitelbaum proposed a systematic approach for deriving incremental programs [LT95], and Zhang and Lin made the derivation semi-automated [ZL98].

The crucial difference between the previous work and our work is that our work applies to the C language that has mutable data structures and performs the computation with statements that mutate the state. In comparison, functional languages have mostly immutable structures and perform the computation by evaluating the expressions. To the best of our knowledge, there has been no work for incremental computation of imperative, C-like languages.

2.2.4 Other Related Work

Our work is related to previous work on fine-grain access control [S^+94, WCA02]. For example, Mondrian Memory Protection (MMP) [WCA02] provides access control at word granularity using a "protection look-aside buffer" (PLB) to record protection information. MMP can potentially be used to implement location-controlled monitoring. However, like hardware-assisted watchpoints, it needs to raise an exception and, therefore can add significant overhead.

Our work is also related to some of the classic work on capability-based architectures [Fab74, Lev84], protection-enhanced architectures [KCE92], hardware support for security [FS01, L^+00, SCG$^+$03, XKPI02], TLS [CMT00, SBV95, SCZM00, THA$^+$99], and hardware support for instruction-level profiling [DHW$^+$97].

Besides iWatcher and dynamic instrumentation tool PIN [LCM$^+$05], PC-based invariant detection can also be implemented by using other software-based instrumentation tools such as ATOM [SE94] or Dyninst [HMC94], hardware watchpoints [Int01, Joh82, SPA92, Wah92], or other tools [DHW$^+$97]. However, we expect that these tools would result in significant overheads, similar to the overheads of our software implementation AccMon-S. In addition, it is possible to use special hardware [WCA02] that provides fine-grain access control to monitor memory accesses in AccMon. We use iWatcher for the reasons given in Section 3.4.

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 [CCJA98]. 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 [BZ93]. 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 [SDB$^+$03], to identify cache misses early in the pipeline [PLL02], and to filter cache-coherence traffic in snoopy bus-based SMP systems to reduce energy consumption [MMFC01].

Our work is also related to computation reuse [SS97, CmWH99, dCFF00, HL03] and memoization [Pug88, HLY00, MFH95, ABH03], which exploit computation redundancy by reusing previously computed values. Our work differs from these works in that we focus on incrementally checking data structure consistency for mutable data structures, whereas the above work focuses on simply reusing the previously computed values.

Chapter 3

iWatcher: Architectural Support for Dynamic Monitoring

3.1 Overview

Recent impressive advances in microprocessor performance have failed to deliver significant gains in ease of software debugging. This is a major shortcoming of the state of the art, given that software bugs have major implications on computer system availability, reliability and security. Specifically, software bugs account for as much as 40% of computer system failures [MS00], more than 50% of security vulnerabilities in 2001-2004 based on CERT advisories, and cost the U.S. economy $59.5 billion annually, or 0.6% of the GDP [Nat02]!

As we discussed in section 1.1, code debugging is largely done using software techniques: interactive debugger, static checking and dynamic monitoring. They all have significant limitations. This chapter focuses on addressing the first two limitations of dynamic monitoring: inefficiency (large run-time overhead) and inaccuracy, due to using instrumentation. The inaccuracy will cause that some bugs are caught much later than when they actually occur, which makes it hard to find the root cause of the bug. The following C code gives a simple example.

```
int x, *p;
           /* assume invariant: x == 1 */
...
p = foo(); /* a bug: p points to x incorrectly */
*p = 5;    /* line A: unintended corruption of x */
...
```

```
InvariantCheck(x == 1);  /* line B */
z = Array[x];
...
```

While x is corrupted in line A, the bug is not detected until the invariant check at line B. Due to the difficulty of performing perfect pointer disambiguation, it may be hard for a dynamic checker to know that it needs to insert an invariant check right after line A.

To assist software debugging, several processor architectures such as Intel x86 and Sun SPARC provide support for watchpoints to monitor several programmer-specified memory locations [Int01, KH92, SPA92, WLG93]. When a watched memory location is accessed, the hardware triggers an exception that is handled by the debugger. It is then up to the programmer to manually check the program state. While watchpoints are a good starting point, they have several limitations. First, they do not support *low-overhead* checks on variable values *automatically*. Since exceptions are expensive, it would be very inefficient to use them for dynamic bug detection during production runs. Second, most architectures only support a handful of watchpoints (four in Intel x86). Therefore, it is hard to use watchpoints for dynamic monitoring in production runs, which requires efficiency and watching many memory locations.

As micro-architectural innovations have significantly improved performance, interest has recently risen in the architecture community to use transistors to improve software debugging. Several works [PT03, XBH03, NPC05, TCV04] have been conducted along this direction. While recent work is promising and provides a good foundation, it is still far from providing a complete solution.

This chapter introduces the *Intelligent Watcher (iWatcher)*, a novel architectural framework to monitor dynamic execution *automatically*, *flexibly* and with *minimal overhead*. iWatcher associates program-specified monitoring functions with memory locations. When any such location is accessed, the monitoring function is automatically triggered with low overhead. To further reduce overhead and support rollback, iWatcher can optionally leverage Thread-Level Speculation (TLS). The main advantages of iWatcher are:

- It monitors *all* accesses to the watched memory locations. Consequently, it catches hard-to-find bugs such as updates through aliased pointers and stack-smashing attacks commonly exploited by viruses.

- It has low overhead because it (i) only monitors memory instructions that *truly* access the watched memory locations, and (ii) uses minimal-overhead hardware-supported triggering of monitoring functions.

- It is general and flexible in that it can support a wide range of checks, including program-specific checks. Therefore, iWatcher can be used to build detection tools covering variety of bugs. Moreover, iWatcher is language independent, cross-module and cross-developer.

- It can *optionally* leverage TLS to hide monitoring overhead and provide rollback support. Specifically, with TLS, a monitoring function is executed in parallel with the rest of the program, and the program can be rolled back if a bug is found.

In contrast, due to aliasing problems, it is very hard for software-only dynamic checkers to monitor all accesses to the watched memory locations and only those.

We evaluate iWatcher using seven buggy applications with various real and injected bugs including accessing freed locations, memory leaks, buffer overflow, value-invariant violations, and smashed stacks. iWatcher detects all the bugs evaluated in our experiments with only a 0.1-179% execution overhead. Overall, iWatcher's reasonably small overhead and ability to monitor many memory locations enable it to be used in both in-house testing and production runs. In contrast, a well-known open-source bug detector called Valgrind induces orders of magnitude more overhead, and can only detect a subset of the bugs. Moreover, even with 20% of the dynamic loads monitored in a program, iWatcher only adds 72-182% overhead. We also show that TLS is effective at reducing overheads for programs with substantial monitoring. Finally, supporting four contexts in an SMT is enough to achieve the best performance in our experiments.

The remainder of this chapter is organized as follows. Sections 3.2, 3.3, and 3.4 describe iWatcher's functionality, architectural design, and advantages, respectively. Section 3.5 describes how to use iWatcher to detect various bugs. Sections 3.6 and 3.7 present the evaluation methodology and experimental results. Section 3.8 summarizes this chapter.

3.2 iWatcher Functionality

iWatcher provides high-flexibility and low-overhead dynamic execution monitoring. It associates program-specified monitoring functions with memory locations. When any such location is accessed, the monitoring function associated with it is automatically triggered and executed.

iWatcher provides two system calls to turn on and off monitoring on a memory location, namely *iWatcherOn* and *iWatcherOff*. These calls can be inserted in programs either automatically by an instrumentation tool or manually by programmers. The following is the *iWatcherOn* interface:

```
iWatcherOn(MemAddr, Length, WatchFlag, ReactMode,
   MonitorFunc, Param1, Param2, ... ParamN)
/* MemAddr: starting address of the memory region*/
/* Length: length of the memory region */
/* WatchFlag: types of accesses to be monitored */
/* ReactMode: reaction mode */
/* MonitorFunc: monitoring function */
/* Param1...ParamN: parameters of MonitorFunc */
```

If a program makes such a call, iWatcher associates monitoring function $MonitorFunc()$ with a memory region of $Length$ bytes starting at $MemAddr$. The $WatchFlag$ specifies what types of accesses to this memory region should be monitored. Its value can be “READONLY”, “WRITEONLY”, or “READWRITE”, in which case the monitoring function is triggered on a read access, write access or both, respectively.

At a *triggering access* (an access to a monitored memory location), the hardware automatically initiates the monitoring function associated with this memory location. The architecture passes the values of $Param1$ through $ParamN$ to the monitoring function. In addition, it also passes information about the triggering access, including the program counter, the type of access (load or store; word, half-word, or byte access), reaction mode, and the memory location being accessed. It is the monitoring function's responsibility to perform the check.

A monitoring function can have side effects and can read and write variables without any restrictions. To avoid recursive triggering of monitoring functions, no memory access performed inside a monitoring function can trigger another monitoring function.

From the programmers' point of view, the execution of a monitoring function follows sequential semantics, just like a very lightweight exception handler (Section 3.3 describes why monitoring in iWatcher is very lightweight). The semantic order is: the triggering access, the monitoring function, and the rest of the program after the triggering access.

Upon successful completion of a monitoring function, the program continues normally. If the monitoring function fails (returns FALSE), different actions are taken depending on the $ReactMode$ parameter specified in iWatcherOn(). iWatcher supports three modes: $ReportMode$, $BreakMode$ and $RollbackMode$:

- $ReportMode$: The outcome of the check is reported and the program continues. This mode can be used for profiling and error reporting without interfering with the execution of the program.

- $BreakMode$: The program pauses at the state right after the triggering access and control is passed to an exception handler. Users can potentially attach an interactive debugger, which can be used to find more information.

- $RollbackMode$: The program rolls back to the most recent checkpoint, typically much before the triggering access. This mode can be used to support replay of a code section to analyze an occurring bug [PT03], or to support transaction-based programming [OL02].

A program can associate multiple monitoring functions with the same location. In this case, upon an access to the watched location, all monitoring functions are executed following sequential semantics according to their setup order.

When a program is no longer interested in monitoring a memory region, it turns off the monitoring using

```
iWatcherOff(MemAddr, Length, WatchFlag, MonitorFunc)
/* MemAddr: starting address of the watched region*/
/* Length: length of the watched region */
/* WatchFlag: types of accesses to be unmonitored */
/* MonitorFunc: the monitoring function */
```

After this operation, the $MonitorFunc$ associated with this memory region of $Length$ bytes starting at $MemAddr$ and $WatchFlag$ is deleted from the system. Other monitoring functions associated with this region are still in effect.

Besides using the iWatcherOff() call to turn off monitoring for a specified memory region, the program can also use a $MonitorFlag$ global switch that enables or disables monitoring on all watched locations. This switch is useful when monitoring overhead is a concern. When the switch is disabled, no location is watched and the overhead imposed is negligible.

Note that iWatcher only provides a very flexible mechanism for dynamic execution monitoring. It is not iWatcher's responsibility to ensure that a monitoring function is written correctly, just like an *assert(condition)* call cannot guarantee that the condition in the code is correct. Programmers can use invariant-inferring tools such as DIDUCE [HL02] and DAIKON [ECGN00] to automatically insert iWatcherOn() and iWatcherOff() calls into programs.

With this support, we can rewrite the example of Section 3.1 using iWatcherOn() and iWatcherOff() operations. There is no need to insert the invariant check. iWatcherOn() is inserted at the very beginning of the program so that the system can continuously check x's value whenever and however the memory location is accessed. This way, the bug is caught at line A.

```
int x, *p;
           /* assume invariant: x = 1 */
iWatcherOn(&x, sizeof(int), READWRITE,
           BreakMode, &MonitorX, &x, 1);
...
p = foo(); /* a bug: p points to x incorrectly */
*p = 5;    /* line A: a triggering access */
...
z = Array[x]; /* line B: a triggering access */
...
iWatcherOff(&x, sizeof(int), READWRITE, &MonitorX);

bool MonitorX(int *x, int value){
  return  (*x == value);
}
```

3.3 Architectural Design of iWatcher

To implement the functionality described above, there are at least four challenges: (1) How to monitor a location? (2) How to detect a triggering access? (3) How to trigger a monitoring function? (4) How to support the three reaction modes? In this section, we first give an overview of the implementation and then show how it addresses these challenges.

3.3.1 Overview of the Implementation

iWatcher is implemented using a combination of hardware and software. Logically, it has four main parts. First, to detect triggering accesses on small monitored memory regions, we tag cache

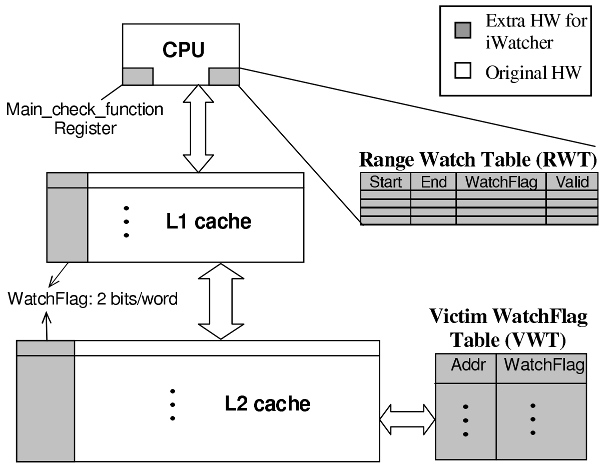


Figure 3.1: iWatcher hardware architecture.

lines in both L1 and L2 caches with WatchFlags; to detect triggering accesses on large monitored memory regions, we use a small *Range Watch Table (RWT)*. Second, the hardware triggers monitoring functions on the fly and provides a special *Main_check_function* register to store the common entry point for all monitoring functions. Third, we use software to manage the associations between watched locations and monitoring functions. Finally, we *optionally* leverage TLS to reduce overheads.

Figure 3.1 gives an overview of the iWatcher hardware. Each L1 and L2 cache line is augmented with WatchFlags. They identify words belonging to small monitored memory regions. There are two WatchFlag bits per word in the line: a read-monitoring one and a write-monitoring one. If the read (write)-monitoring bit is set for a word, all loads (stores) to this word automatically trigger the corresponding monitoring function. The processor also has a Main_check_function register that holds the address of the *Main_check_function()*, which is the common entry point to all program-specified monitoring functions. In addition, iWatcher also has a *Victim WatchFlag Table (VWT)*, which stores the WatchFlags for watched lines of small regions that have at some point been displaced from L2.

To detect accesses to large (multiple pages) monitored memory regions, iWatcher uses a set of registers organized in the RWT. Each RWT entry stores the start and end virtual addresses of a large region being monitored, plus two bits of WatchFlags and one valid bit. We will see that the RWT is used to prevent large monitored regions from overflowing the L2 cache and the VWT. The lines of these regions are not brought into the cache on an iWatcherOn() call. Moreover, the WatchFlags of these lines do not need to be set in the L1 or L2 cache unless the lines are also included in a small monitored region. When the RWT is full, additional large monitored regions are treated the same way as small regions.

The software component of iWatcher includes the iWatcherOn/Off() system calls, which set or remove associations of memory locations with monitoring functions. iWatcher uses a software table called *Check Table* to store detailed monitoring information for each watched memory location. The information stored includes MemAddr, Length, WatchFlag, ReactMode, MonitorFunc, and Parameters. Using software simplifies the hardware. An iWatcherOn/Off() call adds or removes the corresponding entry to or from the Check Table.

The iWatcher software also implements the Main_check_function() library call, whose starting address is stored in the Main_check_function register. When a triggering access occurs, the hardware sets the program counter to the address in this register. The Main_check_function() is responsible to call the program-specified monitoring function(s) associated with the accessed location. To do this, it needs to search the Check Table and find the corresponding function(s).

To reduce monitoring overhead, iWatcher can optionally leverage TLS to speculatively execute the main program in parallel with monitoring functions. Moreover, iWatcher can also leverage TLS to roll back the buggy code with low overhead, for subsequent replay.

While TLS was also used by Oplinger and Lam to hide overheads [OL02], iWatcher uses a different TLS spawning mechanism. Specifically, iWatcher uses dynamic hardware spawning, which requires no code instrumentation. Oplinger and Lam, instead, insert thread-spawning instructions in the program statically. In general, their approach is less efficient and may hurt some conventional compiler optimizations. Many of the new issues that appear with dynamic hardware spawning are discussed in Sections 3.3.3 and 3.3.4.

3.3.2 Watching a Range of Addresses

When a program calls iWatcherOn() for a memory region equal or larger than *LargeRegion*, iWatcher tries to allocate an RWT entry for this region. If there is already an entry for this region in the RWT, iWatcherOn() sets the entry's WatchFlags to the logical OR of its old value and the WatchFlag argument of the call. If, instead, the region to be monitored is smaller than *LargeRegion*, iWatcher loads the watched memory lines into the L2 cache (if they are not already in L2). We do not explicitly load the lines into L1 to avoid unnecessarily polluting L1. As a line is loaded from memory, iWatcher accesses the VWT to read-in the old WatchFlags, if they exist there. Then, it sets the WatchFlag bits in the L2 line to be the logical OR of the WatchFlag argument of the call and the old WatchFlags. If the line is already present in L2 and possibly L1, iWatcher simply sets the WatchFlag bits in the line to the logical OR of the WatchFlag argument and the current WatchFlag. In all cases, iWatcherOn() also adds the monitoring function to the Check Table.

When a program calls iWatcherOff(), iWatcher removes the corresponding monitoring function entry from the Check Table. Moreover, if the monitored region is large and there is a corresponding RWT entry, iWatcherOff() updates this RWT entry's WatchFlags. The new value of the WatchFlags is computed from the remaining monitoring functions associated with this memory region, according to the information in the Check Table. If there is no remaining monitoring function for this range, the RWT entry is invalidated. If, instead, the memory region is small, iWatcher finds all the lines of the region that are currently cached and updates their WatchFlags based on the remaining monitoring functions. iWatcher also updates (and, if appropriately removes) any corresponding VWT entries.

Caches and VWT are addressed by the physical addresses of watched memory regions. If there is no paging by the OS, the mapping between physical and virtual addresses is fixed for the whole program execution. In our prototype implementation, we assume that watched memory locations are pinned by the OS, so that the page mappings of a watched region do not change until the monitoring for this region is disabled using iWatcherOff().

Note that the purpose of using RWT for large regions is to reduce L2 pollution and VWT space consumption: lines from this region will only be cached when referenced (not during iWatcherOn()) and, since they will not set their WatchFlags in the cache, they will not use space in the VWT on cache eviction.

It is possible that iWatcherOn()/iWatcherOff() access some memory locations sometimes as part of a large region and sometimes as a small region. In this case, the iWatcherOn() or iWatcherOff() software handlers, as they add or remove entries to or from the Check Table, are responsible for ensuring the consistency between RWT entries and L2/VWT WatchFlags.

3.3.3 Detecting Triggering Accesses

iWatcher needs to identify those loads and stores that should trigger monitoring functions. A load or store is a triggering access if the accessed location is inside any large monitored region recorded in the RWT, or the WatchFlags of the accessed line in L1/L2 are set.

In practice, the process of detecting a triggering access is complicated by the fact that modern out-of-order processors introduce access reordering and pipelining. To help in this process, iWatcher augments each reorder buffer (ROB) entry with a *Trigger* bit, and each load-store queue entry with 2 bits that store WatchFlag information.

To keep the hardware reasonably simple, the execution of a monitoring function should only occur when a triggering load or store reaches the head of the ROB. At that point, the values of the architectural registers that need to be passed to the monitoring function are readily available. In addition, the memory system is consistent, as it contains the effect of all preceding stores. Moreover, there is no danger of mispredicted branches or exceptions, which could require the cancellation of an early-triggered monitoring function.

For a load or store, when the TLB is looked up early in the pipeline, the hardware also checks the RWT for a match. This introduces negligible visible delay. If there is a match, the access is a triggering one. If there is no match, the WatchFlags in the caches will be examined to determine if it is a triggering access.

A load typically accesses the memory system before reaching the head of the ROB. It is at that time that a triggering load will detect the set WatchFlags in the cache. Consequently, in our design, as a load reads the data from the cache into the load queue, it also reads the WatchFlag bits into the special storage provided in the load queue entry. In addition, if the RWT or the WatchFlag bits indicate that the load is a triggering one, the Trigger bit associated with the load's ROB entry is set. When the load (or any instruction) finally reaches the head of the ROB and is about to retire, the hardware checks the Trigger bit. If it is set, the hardware triggers the corresponding monitoring function.

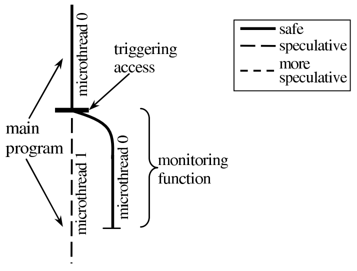
Stores present a special difficulty. A store is not sent to the memory system until it reaches the head of the ROB. At that point, it is retired immediately, but it may still cause a cache miss, in which case it may take a long time to actually complete. In iWatcher, this would mean that, for stores that do not hit in the RWT, the processor may have to wait a long time to know whether it is a triggering access. During that time, no subsequent instruction could be retired, as the processor may have to trigger a monitoring function. To reduce this delay as much as possible, we change the microarchitecture so that, as soon as a store address is resolved early in the ROB, a prefetch is issued to the memory system. Such prefetch brings the data into the cache, and the WatchFlag bits are read into the special storage in the store queue entry. If the RWT or the WatchFlag bits in the caches indicate that the store is a triggering one, the Trigger bit in the ROB entry is also set. With this support, the processor is much less likely to have to wait when the store reaches the head of the ROB. While issuing this prefetch may have implications for the memory consistency model supported in a multiprocessor environment, we consider the topic to be beyond the scope of this chapter.

Note that bringing the WatchFlag information into the load-store queue entries enables correct operation for loads that get their data directly from the load-store queue. For example, if a store in the load-store queue has the read-monitoring WatchFlag bit set, then a load that reads from it will set its own Trigger bit.

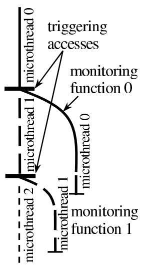
3.3.4 Executing Monitoring Functions

When a triggering load or store is retired, the architectural registers and the program counter are automatically saved, and execution is redirected to the address in the Main_check_function register. After the monitoring function completes, execution resumes from the saved program counter.

As an optimization, we can leverage the TLS mechanism. Specifically, when a triggering load or store is retired, the iWatcher hardware automatically spawns a new microthread (denoted as microthread 1 in Figure 3.2(a)) to speculatively execute the rest of the program after the triggering access, while the current microthread (denoted as microthread 0 in Figure 3.2(a)) executes the monitoring function non-speculatively. To provide sequential semantics (the remainder of the program is semantically after the monitoring function), data dependencies are tracked by TLS and any violation of sequential semantics results in the squash of the speculative microthread (microthread 1).



(a) *Executing a monitoring function.*



(b) *Triggering a monitoring function from a speculative microthread.*

Figure 3.2: Examples of monitoring function execution with TLS support.

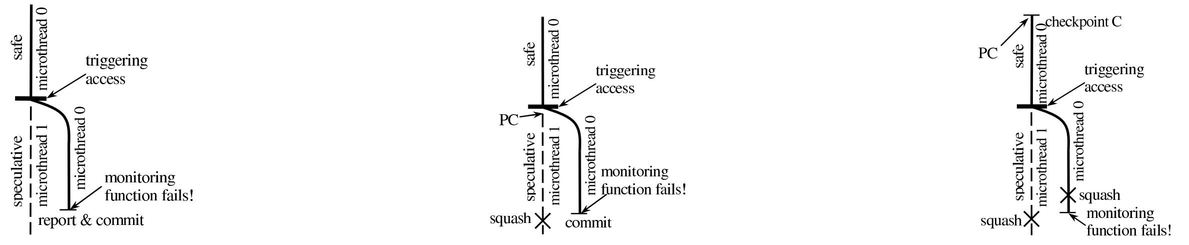
Microthread 0 executes the monitoring function by starting from the address stored in the Main_check_function register. It is the responsibility of the Main_check_function() to find the monitoring function(s) associated with the triggering access and call all such function(s) one after another. Note that, although semantically, a monitoring function appears to programmers like a user-specified exception handler, the overhead of triggering a monitoring function is tiny with our hardware support. Indeed, while triggering an exception handler typically needs OS involvement, triggering a monitoring function in iWatcher is done completely in hardware: the hardware automatically fetches the first instruction from the Main_check_function(). iWatcher can skip the OS because monitoring functions are not related to any resource management in the system and, in addition, do not need to be executed in privileged mode. Moreover, the monitoring functions for a program are in the same address space as the monitored program. Therefore, a "bad" program cannot use iWatcher to mess up other programs.

Microthread 1 speculatively executes the continuation of the monitoring function, i.e., the remainder of the program after the triggering access. To avoid the overhead of flushing the pipeline, iWatcher dynamically changes the microthread ID of all the instructions currently in the pipeline from 0 to 1. Note that, it is possible that some un-retired load instructions after the triggering access may have already accessed the data in the cache and, as per TLS, already updated the microthread ID in the cache line to be 0. Since the microthread ID on these cache lines should now be 1, the hardware re-touches the cache lines that were read by these un-retired loads, correctly setting their microthread IDs to 1. There is no such problem for stores because they only update the microthread IDs in the cache at retirement. In our experiments, these retouches account for a very tiny fraction of all accesses, and practically always hit the L1 cache. So the performance impact is negligible.

It is possible that a speculative microthread issues a triggering access, as shown on Figure 3.2(b). In this case, a more speculative microthread (microthread 2) is spawned to execute the rest of the program, while the speculative microthread (microthread 1) enters the Main_check_function. Since microthread 2 is semantically after microthread 1, a violation of sequential semantics will result in the squash of microthread 2. In addition, if microthread 1 is squashed, microthread 2 is squashed as well. Finally, if microthread 1 completes while speculative, iWatcher does not commit it; it can only be committed after microthread 1 becomes safe.

Note that, in a CMP-based iWatcher, microthreads should be allocated for cache affinity. In our Figure 3.2(a) example, speculative microthread 1 should be kept on the same CPU as the original program, while microthread 0 should be moved to a different CPU. This is because microthread 1 continues to execute the program and is likely to reuse cache state.

3.3.5 Different Reaction Modes



(a) *ReportMode.* (b) *BreakMode.* (c) *RollbackMode.*

Figure 3.3: Different reaction modes supported by iWatcher with the TLS optimization.

If a monitoring function fails, iWatcher takes different actions depending on the corresponding ReactMode. ReactMode can be *ReportMode*, *BreakMode*, and *RollbackMode*.

In *ReportMode*, the outcome of the check is reported and the program continues. This mode is used for profiling and error reporting without interfering with the execution of the program.

In *BreakMode*, the program pauses at the state right after the triggering access, and control passes to an exception handler. Users can attach an interactive debugger, which can be used to find more information.

Finally, in *RollbackMode*, the program rolls back to a previous checkpoint, typically much earlier than the triggering access. This mode can be used to support the replay of a code section to analyze a bug, or to support transaction-based programming.

Figure 3.3 illustrates the three supported reaction modes with the TLS optimization. *Report-Mode* is the simplest one. iWatcher treats it the same way as if the monitoring function had succeeded: microthread 0 commits and microthread 1 becomes safe. If the reaction mode is $BreakMode$, iWatcher commits microthread 0 but squashes microthread 1. The program state and the program counter (PC) of microthread 1 are restored to the state it had immediately after the triggering access (Figure 3.3(b)). The cache updates of microthread 1 are discarded. At this point, programmers can use an interactive debugger to analyze the bug.

If the reaction mode is $RollbackMode$, iWatcher squashes microthread 1 and also rolls back microthread 0 to a previous checkpoint (the checkpoint at PC in Figure 3.3(c)). iWatcher can use the support similar to ReEnact [PT03] to provide this reaction mode.

3.3.6 Other Issues

Displacements and Cache Misses When a watched line of small regions is about to be displaced from the L2 cache, its WatchFlags are saved in the VWT. The VWT is a small set-associative buffer. If the VWT needs to take an entry while full, it selects a victim entry to be evicted, and delivers an exception. The OS then turns on page protections for the pages that correspond to the WatchFlags to be evicted from the VWT. Future accesses to these pages will trigger page protection faults, which will enable the OS to insert their WatchFlags back into the VWT. However, in our experiments, we find that a 1024-entry VWT is never full. The reason is that the VWT only keeps the WatchFlags for watched lines of small regions that have at some point been displaced from L2.

On an L2 cache miss, as the line is read from memory, the VWT is checked for an address match. If there is a match, the WatchFlags for the line are copied to the destination location in the cache. We do not remove the WatchFlags from the VWT because the memory access may be speculative and be eventually undone. If there is no match, the WatchFlags for the loaded line are set to the default "un-watched" value. Note that this VWT lookup is performed in parallel with the memory read and, therefore, introduces negligible visible delay.

If TLS is used, speculative lines cannot be displaced from the L2. If space is needed in a cache set that only holds speculative lines, a speculative microthread is squashed to make room. More details can be found in [PT03].

Check Table Implementation The Check Table is a software table. Our current implementation uses one entry for each watched region. The entries are sorted by start address. To speed-up Check Table lookup, we exploit memory access locality to reduce the number of table entries accessed during one search. A table entry contains all arguments of the iWatcherOn() call. If there are multiple monitoring functions associated with the same location, they are linked together. Since the Check Table is a pure software data structure, it is easy to change its implementation. For example, a possible implementation could be to organize it as a hash table. It can be hashed with the virtual address of the watched location.

3.4 Advantages of iWatcher

Based on the previous discussion, we can list the advantages of iWatcher. One of them is that it provides location-controlled monitoring. Therefore, *all* accesses to a watched memory location are monitored, including "disguised" accesses due to dangling pointers or wrong pointer manipulations.

Another advantage of iWatcher is its low overhead. iWatcher only monitors memory operations that truly access a watched memory location. Moreover, iWatcher uses hardware to trigger monitoring functions with minimal overhead. Finally, iWatcher optionally uses TLS to execute monitoring functions in parallel with the rest of the program, effectively hiding most of the monitoring overhead.

iWatcher is flexible and extensible. Programmers or automatic instrumentation tools can add monitoring functions. iWatcher is convenient even for manual instrumentation because programmers do not need to instrument every possible access to a watched memory location. Instead, they only need to insert an iWatcherOn() call for a location when they are interested in monitoring this location and an iWatcherOff() call when the monitoring is no longer needed. In between, all possible accesses to this location are automatically monitored. In addition, iWatcher supports three reaction modes, giving flexibility to the system.

iWatcher is cross-module and cross-developer. A watched location inserted by one module or one developer is automatically honored by all modules and all developers whenever the watched location is accessed.

iWatcher is language independent since it is supported directly in hardware. Programs written in any language, including C/C++, Java or other languages can use iWatcher. For the same reason, iWatcher can also support dynamic monitoring of low-level system software, including the

Feature	Assertions	Hardware Watchpoints	Software-Only Dynamic Checkers		iWatcher
			Valgrind	Generic	
Hardware support	No	Simple	No	No	Modest. Optionally, TLS
Type of checks	Code-controlled	Location-controlled	Code-controlled	Code-controlled	Location-controlled
Reaction modes	Abort	Interrupt	Report or break	Report or break	Report, break or rollback
Programmer's effort	High	High	Low	Low	Moderate or low (automatic instrumentation)
Language dependent	No	No	No	Typically yes	No
Flexibility	Very flexible. Program specific	Inflexible. Support only a few watchpoints. Rely on programmers or debuggers for checks	Moderately flexible. Currently, it only supports limited types of checks	Moderately flexible	Very flexible. Program specific
Cross-module and cross-developer	No	Yes	Yes	No	Yes
Completeness	Hard to make sure all accesses are checked	Detects all accesses	Detects all accesses	May miss some accesses due to aliasing problems	Detects all accesses
Overhead	High	Low	Very high	Typically high	Low

Table 3.1: Comparison of iWatcher to three other approaches. Completeness refers to whether an approach monitors all accesses to a watched memory location by construction. Examples of software-only dynamic checkers include Purify, DIDUCE, Eraser, etc.

operating system.

iWatcher can be used to detect illegal accesses to a memory location. For example, it can be used for security checks to prevent illegal accesses to some secured memory locations. In our experiments, we have used iWatcher to protect the return address in a program stack to detect stack-smashing attacks [CPM$^+$98, FS01, One96, XKPI02].

Table 3.1 summarizes the differences between iWatcher and the three related approaches discussed in Section 2.1.

3.5 Bug Detection

To detect bugs, programmers and some software debugging tools like DIDUCE [HL02] need to use specific monitoring functions. In this section, to demonstrate how to write monitoring functions for iWatcher, we describe some monitoring functions that are used in our experiments to detect various bugs, including buffer overflow, memory leaks, stack smashing, accessing freed memory, and invariant violations.

(1) Detecting Buffer Overflow (BO_check): To detect buffer overflow for both dynamic buffers and static arrays, some paddings are added at the two ends of each buffer. The padding areas are then monitored by iWatcher. The monitoring function simply reports any accesses to these padding areas as bugs. It may miss some bugs if the out-of-bound access does not hit the padding areas. When a dynamic buffer is deallocated, its paddings are also freed and the corresponding monitoring is turned off.

(2) Detecting Memory Leaks (ML_check): Memory leak bugs are tackled by monitoring all accesses to heap objects. Each heap access triggers the monitoring function, which updates the time-stamp associated with the accessed object. The monitoring is turned off when an object is deallocated. Periodically, the time-stamps are checked. The heap objects that have not been accessed for a long time are likely to be memory leaks. Those objects are then ranked based on their time-stamps.

(3) Detecting Accesses to Freed Locations (FREE_check): All unallocated memory space in the heap is monitored using iWatcher. Any access to this space triggers the monitoring function that reports it as a bug. When a memory region is allocated, the monitoring for this memory region is turned off. Of course, the monitor may miss bugs in some cases, such as a dangling pointer points to a reallocated location.

(4) Detecting Various Memory Bugs (COMBO_check): This is used to catch all the above three types of bugs. The monitoring function is combination of the above three functions.

(5) Detecting Stack Smashing (STACK_check): To catch stack smashing bugs that are commonly exploited by viruses to launch security attacks, iWatcher monitors every stack location that stores return addresses. More specifically, after entering a function, iWatcherOn() is called on the return address location, and before the function returns, iWatcherOff() is called to turn off the monitoring to this location. The monitoring function, if triggered, simply reports any access as a bug.

(6) Detecting Invariant Violations (IV_check): To detect an invariant violation, the specific variable needs to be monitored. The monitoring function checks if the variable value satisfies the program-specific invariant.

The first five are general checks. The monitoring can be fully automated using a tool to insert the monitors into any programs. The last check is a program-specific monitoring, requiring specific knowledge about the program semantics.

Monitoring dynamic objects is done by wrapping the memory allocation and deallocation functions to insert iWatcherOn() and iWatcherOff() calls. For monitoring static arrays, the paddings and iWatcherOn/Off() calls are added manually now, although they can be done automatically with compiler support. For STACK_check, we use the compiler support to identify the stack location that stores the return address and add the iWatcherOn/Off() calls. The iWatcherOn/Off() calls for IV_check are inserted manually now, but can be automated using tools such as DAIKON and DIDUCE.

3.6 Evaluation Methodology

3.6.1 Simulated Architecture

To evaluate iWatcher, we have built an execution-driven simulator that models a workstation with a 4-context SMT processor augmented with TLS support and iWatcher functionality. The experiments are conducted on this default architecture unless specifically mentioned. The parameters of the architecture are shown in Table 3.2. We model the overhead of spawning a monitoring-function microthread as 5 cycles of processor stall visible to the main-program thread. The reaction mode used in all experiments is ReportMode, so that all programs can run to completion.

CPU frequency	2.4GHz	ROB size	360
Fetch width	16	I-window size	160
Issue width	8	Int FUs	6
Retire width	12	Mem FUs	4
Ld/st queue entries	32/thread	FP FUs	4
Spawn overhead	5 cycles	Reaction mode	ReportMode
L1 cache	32KB, 4-way, 32B/line, 3 cycles latency		
L2 cache	1MB, 8-way, 32B/line, 10 cycles latency		
VWT	1024 entries, 8-way, 2B/entry		
LargeRegion	64Kbytes		
RWT	4 entries, 32bits for the start and end addresses		
Memory	200 cycles latency		

Table 3.2: Parameters of the simulated architecture. Latencies are given as unloaded round-trips from the processor.

To isolate the benefits of TLS, we evaluate the same architecture without TLS support. In this case, on a triggering access, the processor first executes the monitoring function, and then proceeds to execute the rest of the program. For the evaluation without TLS support (with or without iWatcher support), the single microthread running is given a 64-entry load-store queue.

To study TLS scalability, we vary the number of contexts from 2 to 8 in the SMT machine, using the same number of shared resources as listed in Table 3.2. The number of contexts limits the maximum number of concurrently running microthreads.

3.6.2 Valgrind

In our evaluation, we compare the functionality and overhead of iWatcher to Valgrind [NS03], an open-source memory debugger for x86 programs. We choose Valgrind because it does not require modifying the tested applications and is publicly available. Valgrind is a binary-code 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. Because of this, it finds errors not only in the user code but also in all supporting dynamically-linked libraries. Valgrind takes control of a program before it starts. The program is then run on a synthetic x86 CPU, and every memory access is checked. All detected errors are reported.

Valgrind provides an option to enable or disable memory leak detection. We also enhance Valgrind to enable or disable variable uninitialization checks and invalid memory access checks (checks for buffer overflow and invalid accesses to freed memory locations).

In our experiments, we run Valgrind 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 iWatcher runs on a simulator, we cannot compare the absolute execution time of iWatcher with that of Valgrind. Instead, we compare their relative execution overheads over runs without monitoring.

3.6.3 Tested Applications

We have conducted two sets of experiments. The first one uses seven applications with both *injected* and *real* bugs to evaluate the functionality and overheads of iWatcher for software debugging. The second one systematically evaluates the overheads of iWatcher and the effect of architecture resources when monitoring applications without bugs.

The applications used in our first set of experiments contain various bugs, including memory leaks, accesses to freed locations, buffer overflow, stack-smashing attacks, and value invariant violations. These applications are: *bc-1.06* (an arbitrary precision calculator language), *gzip-1.2.4* (GNU zip, a popular compression utility provided by the GNU project), *polymorph-0.4.0* (a tool to convert Windows style file names to something more portable for UNIX systems), *ncompress-4.2.4* (a compression and decompression utility that is compatible with the original UNIX compress utility), *tar-1.13.25* (a tool to create and manipulate tar archives), *cachelib* (a cache management library developed at the University of Illinois), and *gzip* (a SPECINT 2000 application running the Test input data set). Of these programs, bc-1.06, gzip-1.2.4, polymorph-0.4.0, ncompress-4.2.4, tar-1.13.25 and cachelib already had real bugs (the bugs come with the code and were introduced by the original programmers), while we injected some common bugs into gzip.

Table 3.3 shows the details of the bugs and monitoring functions, as described in Section 3.5.

Application	Type of Monitoring	Bug Class, Location, and Origin	Monitoring Function
gzip-STACK	general	stack smashing: "huft_free()". Injected	STACK_-check
gzip-FREE	general	accessing freed location: "huft_free()". Injected	FREE_-check
gzip-BO1	general	dynamic buffer overflow: "huft_build()". Injected	BO_check
gzip-ML	general	memory leak: "huft_free()". Injected	ML_check
gzip-COMBO	general	combination of the bugs in gzip-ML, gzip-FREE, and gzip-BO1. Injected	COMBO_-check
gzip-BO2	general	static array overflow: "huft_build()". Injected	BO_check
gzip-IV1	program specific	value invariant violation: "huft_build()". Injected	IV_check
gzip-IV2	program specific	value invariant violation: "inflate()". Injected	IV_check
cachelib	program specific	value invariant violation at option.c:line 90. Real	IV_check
bc-1.06	general	dynamic buffer overflow at storage.c:line 176 and util.c:line 557. Real	BO_check
ncompress-4.2.4	general	stack smashing at compress42.c:line 886. Real	STACK_-check
gzip-1.2.4	general	static array overflow at gzip.c:line 1009. Real	BO_check
polymorph-0.4.0	general	stack smashing at polymorph.c:line 193 and 200. Real	STACK_-check
tar-1.13.25	general	dynamic buffer overflow at pregargs.c:line 92. Real	BO_check

Table 3.3: Bugs and monitoring functions. The applications with name in bold are new relative to [Zhou et al. 2004].

We evaluate the case of single type of bugs: stack-smashing, accessing freed location, buffer overflow (dynamic buffer overflow and static array overflow), memory leak, and value-invariant violations. We also evaluate the case of a combination of bugs (memory leak, accessing a freed location, and dynamic buffer overflow). Table 3.3 shows the bug information and monitoring functions.

For fair comparison between Valgrind and iWatcher, in Valgrind we enable only the type of checks that are necessary to detect the bug(s) in the corresponding application. For example, for gzip-ML, we enable only the memory leak checks. Similarly, for gzip-FREE, gzip-BO1, bc-1.06 and tar-1.13.25, we enable only the invalid memory access checks. In all our experiments, variable uninitialization checks are always disabled.

Finally, our second set of experiments evaluates iWatcher overheads and the effect of architecture resources (microthread contexts) by monitoring memory accesses in two unmodified SPECINT 2000 applications running the Test input data set. These applications are gzip and parser. We first measure the overhead for iWatcher with a default 4-context SMT processor and without TLS, as we vary the percentage of *dynamic* loads monitored by iWatcher and the length of the monitoring function. Then, we measure the iWatcher overheads with different numbers (namely 2, 4, 6 and 8) of microthread contexts in the SMT processor.

3.7 Experimental Results

3.7.1 Overall Results

Table 3.4 compares the effectiveness and the overhead of Valgrind and iWatcher. For each of the buggy applications considered, the table shows whether the schemes detect the bug and, if so, the overhead they add to the program's execution time. Recall from Section 3.6 that Valgrind's times are measured on a real machine, while iWatcher's are simulated.

Application	Valgrind		iWatcher	
	Bug Detected?	Overhead (%)	Bug Detected?	Overhead (%)
gzip-STACK	No	-	Yes	80.0
gzip-FREE	Yes	1466	Yes	8.7
gzip-BO1	Yes	1514	Yes	10.4
gzip-ML	Yes	936	Yes	37.1
gzip-COMBO	Yes	1650	Yes	42.7
gzip-BO2	No	-	Yes	10.5
gzip-IV1	No	-	Yes	10.5
gzip-IV2	No	-	Yes	9.6
cachelib	No	-	Yes	3.8
bc-1.06	Yes	7367	Yes	178.9
ncompress-4.2.4	No	-	Yes	2.4
gzip-1.2.4	No	-	Yes	168.3
polymorph-0.4.0	No	-	Yes	0.1
tar-1.13.25	Yes	132	Yes	3.8

Table 3.4: Comparing the effectiveness and overhead of Valgrind and iWatcher.

Consider effectiveness first. Valgrind can detect accessing freed locations, dynamic buffer overflow, memory leak bugs, and the combination of them. iWatcher, instead, detects all the bugs considered. iWatcher's effectiveness is largely due to its flexibility to specialize the monitoring function, and its low-overhead that enables more sophisticated monitoring functionality.

The table also shows that iWatcher has a much lower overhead than Valgrind. For bugs that can be detected by both schemes, iWatcher only adds 4-179% overhead, a factor of 25-169 smaller than Valgrind. For example, in gzip-COMBO, where both iWatcher and Valgrind monitor every access to dynamically-allocated memory, iWatcher only adds 43% overhead, which is 39 times less than Valgrind. iWatcher's low overhead is the result of triggering monitoring functions only when the watched locations are actually accessed, and of using TLS to hide monitoring overheads. The difference in overhead between Valgrind and iWatcher is larger in gzip-FREE, where we are looking for a pointer that de-references a freed-up location. In this case, iWatcher only monitors freed memory buffers, and any triggering access uncovers the bug. As a result, iWatcher's overhead is 169 times smaller than Valgrind's. Similarly, for bc-1.06 and tar-1.13.25, the iWatcher's overheads are 41 and 35 times smaller than Valgrind's, respectively. Finally, our results with Valgrind are consistent with the numbers (12-48 times slowdown) reported in a previous study [NS03].

If we consider all the applications, we see that iWatcher's overhead ranges from 0.1% to 179%. This overhead comes from three effects. The first one is the contention of monitoring-function microthreads and the main program for processor resources (such as functional units or fetch bandwidth) and cache space. Such contention has a high impact when there are more microthreads that want to execute concurrently than hardware contexts in the SMT processor. In this case, the main-program microthread cannot run all the time. Instead, monitoring-function and main-program microthreads share the hardware contexts on a time-sharing basis.

Columns 2 and 3 of Table 3.5 show the fraction of time that there is more than one microthread running or more than four microthreads ready to run, respectively. These figures include the main-program microthread. These figures are closely related to the product of the number of triggering accesses per 1 million instructions (Column 4) times the average size of the monitoring function (Column 7). The larger the product, the bigger these figures. Note that having more than four

Application	% Time with Microthreads		# Triggering Accesses per 1M Instr.	# iWatcher-On/Off() per 1M Instr.	Size of iWatcherOn/-Off() (Cycles)	Size of Monitoring Func. (Cycles)	Max Monitored Memory Size at a Time (Bytes)	Total Monitored Memory Size (Bytes)	Max # of Monitored Objects at a Time
	> 1	> 4							
gzip-STACK	0.1	0.0	0.2	8988.1	20.6	22.4	40	19558568	10
gzip-FREE	0.1	0.0	0.4	0.4	1291.3	24.4	246880	246880	239
gzip-BO1	0.1	0.0	0.4	0.9	210.4	177.0	80	1944	20
gzip-ML	23.1	16.9	13008.9	0.4	582.6	47.4	6613600	6847616	111
gzip-COMBO	26.2	15.2	13009.6	0.4	1082.3	45.2	6847616	6847616	243
gzip-BO2	0.1	0.0	0.2	1.6	59.0	24.8	32	3520	8
gzip-IV1	0.1	0.0	0.7	0.2	40.5	21.7	4	528	1
gzip-IV2	0.1	0.0	1.1	0.1	83.0	23.0	4	4	1
cachelib	0.4	0.0	91.6	0.2	128.9	16.5	40	40	10
bc-1.06	0.1	0.0	4.8	2594.0	412.7	412.0	3272	4336	818
ncompress-4.2.4	1.1	1.0	321.7	160.8	162.5	151.5	4	8	1
gzip-1.2.4	0.9	0.9	371.4	4827.8	280.7	429.0	208	208	52
polymorph-0.4.0	0.1	0.0	0.7	0.3	204.0	127.6	8	20	2
tar-1.13.25	0.1	0.0	0.6	15.4	363.4	174.0	96	96	24

Table 3.5: Characterizing iWatcher execution.

microthreads running does not mean that the main-program microthread starves: the scheduler will attempt to share all the contexts among all microthreads fairly. From the table, we see that three applications use more than 1 microthread for more than 1% of the time. Of those, there are two that have more than 4 microthreads ready to run for a significant fraction of the time. Specifically, this fraction is 15.2% for gzip-COMBO and 16.9% for gzip-ML. Note that these applications have relatively high iWatcher overhead in Table 3.4.

A second source of overhead is the iWatcherOn/Off() calls. These calls consume processor cycles and, in addition, bring memory lines into L2, possibly polluting the cache. The overhead caused by iWatcherOn/Off() cannot be hidden by TLS. In practice, their effect is small due to the small number of calls, except in gzip-STACK, bc-1.06 and gzip-1.2.4. Indeed, Columns 5 and 6 of Table 3.5 show the number of iWatcherOn/Off() calls per 1 million instructions and the average size of an individual call. Except for gzip-STACK, bc-1.06 and gzip-1.2.4, the product of number of calls per 1M instructions times the size per call is tiny compared to the execution cycles taken by 1 million instructions. For these cases, it can be shown that, even if every line brought into L2 by iWatcherOn/Off() calls causes one additional miss, the overall effect on program execution time is very small.

For gzip-STACK, bc-1.06 and gzip-1.2.4, the number of iWatcherOn/Off() calls per 1M instructions is huge (8988, 2594 and 4828, respectively). These calls introduce a large overhead that cannot be hidden by TLS. Moreover, iWatcherOn/Off() calls partially cripple some conventional compiler optimizations such as register allocation. The result is worse code and additional overhead. Overall, while for most applications the iWatcherOn/Off() calls introduce negligible overhead, for gzip-STACK, bc-1.06 and gzip-1.2.4, they are responsible for most of the 80%, 179% and 168% overheads of iWatcher, respectively.

For the applications with STACK_check (gzip-STACK, ncompress-4.2.4, and polymorph-0.4.0), the dominant overhead is the iWatcherOn/Off() calls. Since iWatcherOn() is called before entering any functions and iWatcherOff() is called before returning from any functions, the frequency of iWatcherOn/Off() calls is correlated to the function call frequency. Therefore, so is the iWatcher overhead for STACK_check. For example, since gzip-STACK has much more frequent iWatcherOn/Off() calls than ncompress-4.2.4 and polymorph-0.4.0, it has much higher overhead.

Finally, there is a third, less important source of overhead in iWatcher, namely the spawning of monitoring-function microthreads. As indicated in Section 3.6, each spawn takes 5 cycles. Column 4 of Table 3.5 shows the number of triggering accesses per million instructions. Each of these accesses spawns a microthread. From the table, we see that this parameter varies a lot across applications. For most of these applications, the triggering frequency is very small. Moreover, for all applications, even if we had a higher spawn overhead, such as 10 or 20 cycles, the total overhead is still insignificant.

Overall, we conclude that the overhead of iWatcher can be high (37-179%) if the application needs to execute more concurrent microthreads than contexts provided by the SMT processor (gzip-ML and gzip-COMBO), or the application calls iWatcherOn/Off() very frequently (gzip-STACK, bc-1.06, and gzip-1.2.4). For the other applications analyzed, the overhead is small, ranging from 0.1% to 10.5%.

Finally, the last three columns of Table 3.5 show other parameters of iWatcher execution: average monitoring function size, maximum monitored memory size at a time, and total monitored memory size, respectively. We can see that 6 monitoring functions take less than 25 cycles, and there are 8 applications where monitoring functions take 45-429 cycles. In some cases such as gzip-ML and gzip-COMBO, these relatively expensive monitoring functions occur in applications with frequent triggering accesses. When this happens, the fraction of time with more than 4 microthreads is high, which results in high iWatcher overhead (Table 3.4).

The last two columns show that in some applications such as gzip-ML and gzip-COMBO, iWatcher needs to monitor many addresses. In this case, the Check Table will typically contain many entries. Note, however, that even in this case, the size of the monitoring function, which includes the Check Table lookup, is still modest. This is because our Check Table lookup algorithm is efficient for most applications evaluated in our experiments.

3.7.2 Benefits of TLS

As indicated in Section 3.6, our experiments are performed using ReportMode. In this reaction mode, TLS speeds-up execution by running monitoring-function microthreads in parallel with each other and with the main program. To evaluate the effect of not having TLS, we now repeat the experiments executing both monitoring-function and main-program code sequentially, instead of spawning microthreads to execute them in parallel.

Figure 3.4 compares the execution overheads of iWatcher and iWatcher without TLS for all the applications. The amount of monitoring overhead that can be hidden by TLS in a program is the product of Columns 4 and 7 in Table 3.5. For programs with substantial monitoring, TLS reduces the overheads. For example, in gzip-COMBO, the overhead of iWatcher without TLS is 61.4%, while it is only 42.7% with TLS. This is a 30% reduction. As monitoring functions perform more sophisticated tasks such as DIDUCE's invariant inference [HL02], the benefits of TLS will become more pronounced.

For programs with little monitoring, the product of Columns 4 and 7 in Table 3.5 is small. For these applications, TLS does not provide benefit, because there is not much overhead that can be hidden by TLS.

Overall, we recommend supporting TLS, as it reduces the overhead of iWatcher in some ap-

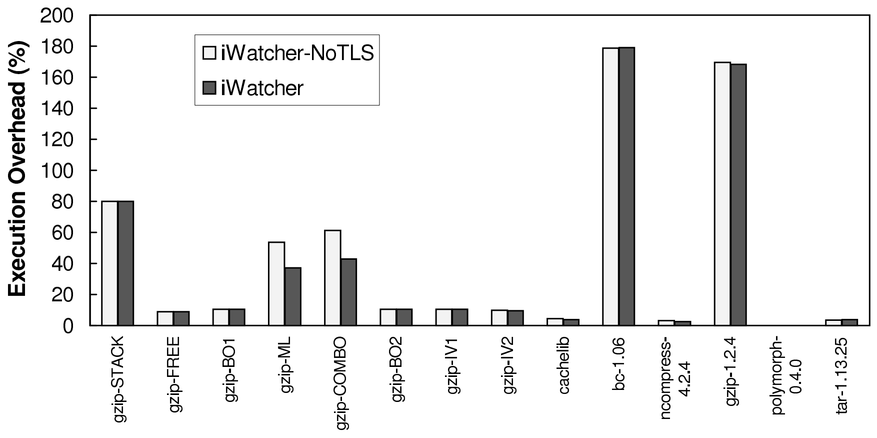


Figure 3.4: Comparing iWatcher and iWatcher without TLS.

plications. We also note that TLS can be instrumental in efficiently supporting RollbackMode (Section 3.3.5).

3.7.3 Sensitivity Study

To measure the sensitivity of iWatcher's overhead, we artificially vary the fraction of triggering accesses and the size of the monitoring functions. We perform the experiments on the bug-free gzip and parser applications.

In a first experiment, we trigger a monitoring function every Nth *dynamic* load in the program[1], where N varies from 2 to 10. The function walks an array, reading each value and comparing it to a constant for a total of 40 instructions. The resulting execution overheads for iWatcher (with the default 4-context SMT processor), and iWatcher without TLS are shown in Figure 3.5 (bar iWatcher-TLS4 and iWatcher-NoTLS, respectively). The figure shows that the overhead of iWatcher with TLS with frequent triggering accesses is tolerable. Specifically, the gzip overhead is 72% for 1 trigger out of 5 dynamic loads, and 194% for 1 trigger out of 2 loads. The parser overheads are a bit higher, namely 182% for 1 trigger out of 5 loads, and 409% for 1 trigger out of 2 loads. If iWatcher does not support TLS, however, the overheads go up: 273% for gzip and 593% for parser for 1 trigger out of 2 loads.

[1]For parser, we skip the program's initialization phase, which lasts about 280 million instructions, because its

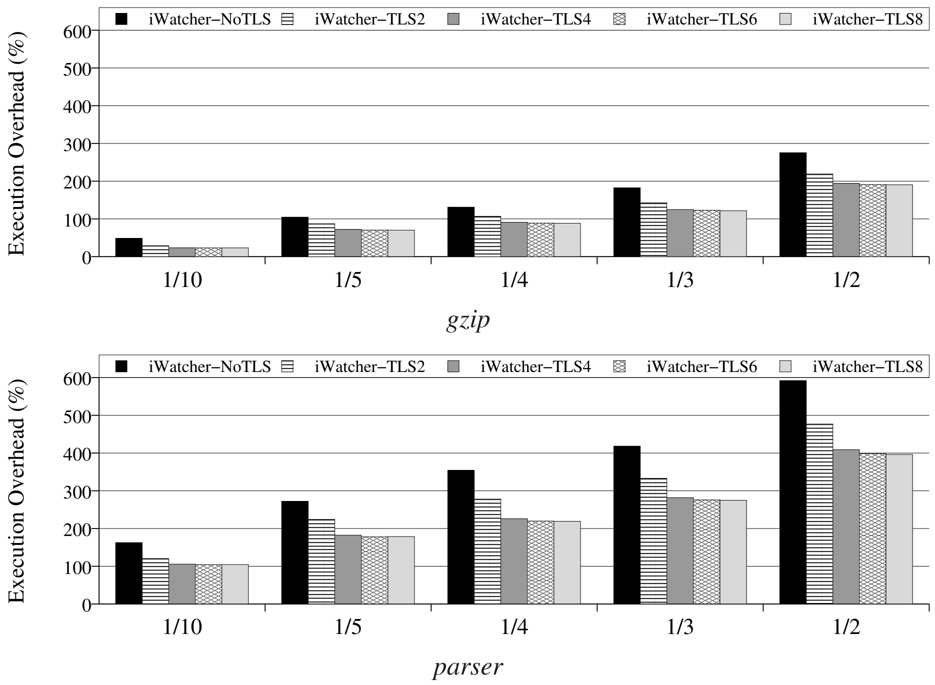


Figure 3.5: Varying the fraction of triggering loads.

In a second experiment, we vary the size of the monitoring function. We use the same function as before, except that we vary the number of instructions executed from 4 to 800. The function is triggered in 1 out of 10 dynamic loads. The resulting execution overheads are shown in Figure 3.6 (iWatcher-TLS4 and iWatcher-NoTLS). The figure again shows that iWatcher overheads with TLS are modest. For 200-instruction monitoring functions, the overhead is 65% for gzip and 169% for parser. In iWatcher without TLS, the overhead is 173% for gzip and 356% for parser. As we increase the monitoring function size, the absolute benefits of TLS increase, as TLS can hide more monitoring overhead.

3.7.4 Scalability Analysis

To evaluate the effect of architectural resources on iWatcher's overhead, we use different numbers (2, 4, 6 and 8) of microthread contexts for the two experiments performed in the sensitivity study (Section 3.7.3). Note that, in these experiments, the SMT processors with different numbers of

behavior is not representative of steady state.

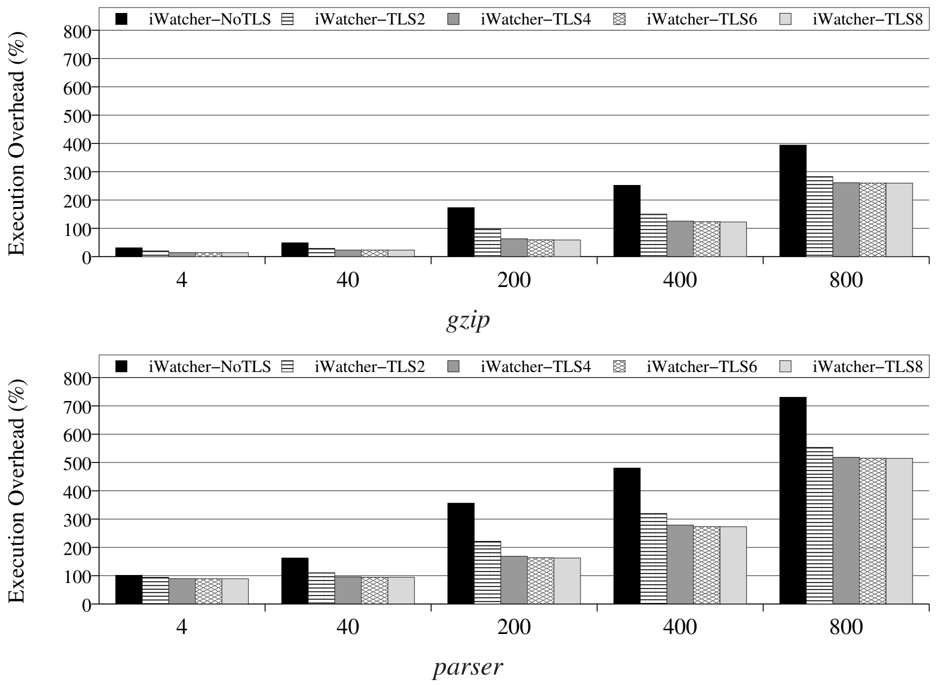


Figure 3.6: Varying the size of the monitoring function.

contexts have the same number of shared resources.

The first experiment varies the fraction of triggering accesses, as we did in the first experiment of Section 3.7.3. The second to fifth bars in Figure 3.5 show that the execution overheads for iWatcher with TLS on a 2/4/6/8-context SMT processor, respectively. The results show that using a 4-context SMT reduces iWatcher's overhead more than using a 2-context SMT. However, using a 6 or 8-context SMT shows little improvement over using a 4-context SMT. More specifically, when using a 4-context SMT instead of a 2-context SMT, the gzip overhead decreases by 17.3% for 1 trigger out of 5 dynamic loads, and by 11.4% for 1 trigger out of 2 loads. For parser, the overhead reduction using a 4-context SMT rather than a 2-context SMT is 18.6% for 1 trigger out of 5 loads, and 14.2% for 1 trigger out of 2 loads. However, the overheads with a 6 or 8-context SMT are almost the same as the overheads with a 4-context SMT in all triggering fractions for both gzip and parser.

The second experiment varies the size of the monitoring function, as the second experiment in Section 3.7.3. The resulting execution overheads are shown in Figure 3.6, from the second to the fifth bars. The results again show that a 4-context SMT is enough to reduce the overheads to the minimum for almost all cases. There is no need to use more than 4 contexts for this experiment. For example, in the 200-instruction monitoring function case, the overhead reduction as we get from two to four contexts is 35.5% for gzip and 23.8% for parser. However, the overheads are pretty much the same from 4 contexts to 6 or 8 contexts.

3.8 Summary

This chapter has presented *iWatcher*, a novel architectural scheme for minimal-overhead location-controlled monitoring. iWatcher detects all accesses to a watched memory location, including those by aliased pointer dereferences. To reduce overhead, iWatcher optionally leverages Thread-Level Speculation (TLS). We have evaluated iWatcher on applications with various bugs. iWatcher detects all bugs evaluated in our experiments with only a 0.1-179% execution overhead. In contrast, a well-known open-source bug detector called Valgrind induces orders of magnitude more overhead, and can only detect a subset of the bugs. Moreover, even with 20% of the dynamic loads monitored in a program, iWatcher only adds 72-182% overhead. Finally, TLS is effective at reducing overheads for programs with substantial monitoring, and a 4-context SMT is enough to achieve the best performance in our experiments.

Chapter 4

AccMon: Automatically Detecting Memory-related Bugs

4.1 Overview

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 [HJ92], CCured [CHM$^+$03, NMW02], SafeC [ABS94] and Jones and Kelly's tool [JK97].

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 [LAZJ03] uses statistical analysis to find the difference between abnormal and normal runs for the purpose of providing more information for postmortem bug analysis. DAIKON [ECGN99, ECGN00] and DIDUCE [HL02] 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 bug detection over software-only solutions: (1) *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) *Generality*: 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.

This chapter proposes two innovative ideas in architectural support for software bug detection. First, we find many memory-related bugs, such as stack smashing, buffer overflow, memory corruption and some semantic bugs (e.g., wrong pointer assignment), share a common symptom that a key variable is accessed by an "illegal" instruction which usually do not access this variable in bug-free runs. We call such "illegal" instruction an outlier instruction. Based on this phenomena, we propose a new statistics-based method, called *program counter (PC)-based invariance*, to detect memory-related bugs *on the fly*. We also observe that in most programs, a given variable is typically accessed by only a few instructions, which can be used to identify the outlier instructions. We validate this observation using statistical analysis with nine applications (See Section 4.2). 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, and thus detect bugs. This is regardless of the values that these instructions assign to the variables.

Second, we propose a simple architectural extension, called the *Check Look-aside Buffer (CLB)*, that uses a Bloom filter [Blo70] to reduce the monitoring overhead in iWatcher. This extension takes advantage of the good temporal locality that exists in object 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. We also use the binary instrumentation tool PIN [LCM$^+$05] to build a pure software implementation of PC-based invariant detection tool called *AccMon-S*.

Our experimental results with nine buggy applications (with a total of twelve bugs) show that AccMon and AccMon-S can detect all ten bugs in the seven non-server applications 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 and AccMon-S catches a bug in the bc application that has never been reported*. Moreover, AccMon-S also detects the two bugs in the two server programs, apache and squid, with 0-4 false alarms (AccMon cannot run these servers due to the limitation of our simulator infrastructure). AccMon also has low overhead (0.24-2.88 times), which is an order of magnitude lower than Purify [HJ92]. 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 [HJ92] and CCured [CHM$^+$03, NMW02].

- 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 [ECGN99, ECGN00] and DIDUCE [HL02].

- 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.

This remainder of this chapter is organized as follows. Section 4.2 discusses the rationale for PC-based invariants. Section 4.3 presents the main idea and the details of our AccMon tool. Experimental methodology and results are presented in Section 4.5 and 4.6, respectively, followed by the summary in Section 4.7.

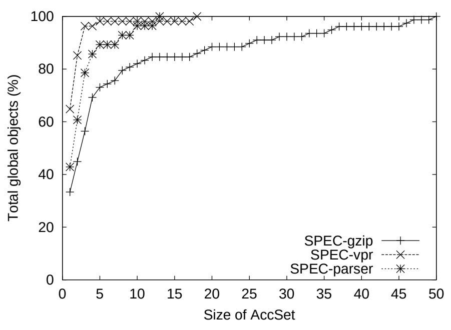
4.2 PC-Based Invariants

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.

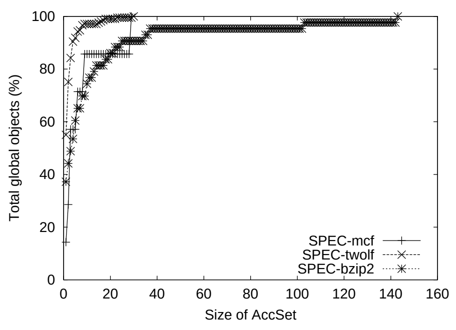
Intuitively, this characteristic makes sense. In most programs, a memory object is accessed at only a few places. For example, a linked list is usually accessed by the list manipulation functions. Also, from the programmers' point of view, it is very difficult to write or understand a program where a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

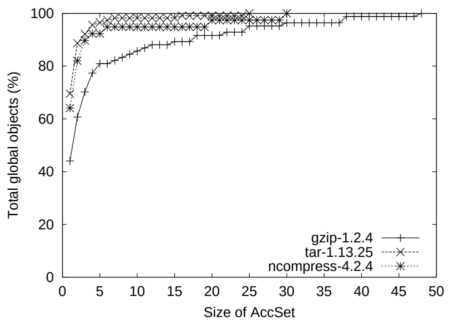
To validate this observation and understand the characteristics of AccSets, we have analyzed the behavior of twelve programs (six real applications used in our evaluation of AccMon and six SPEC2000 benchmarks). In particular, we examine the average size and stability of AccSets. If the average AccSet 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.



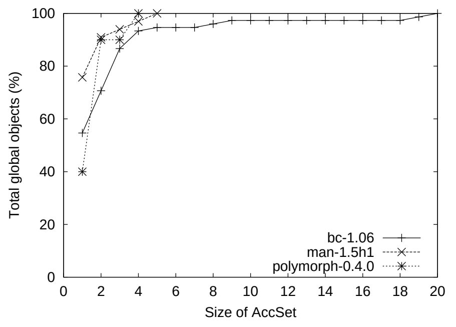
(a) *SPEC2000 benchmarks (1)*



(b) *SPEC2000 benchmarks (2)*



(c) *Real applications (1)*

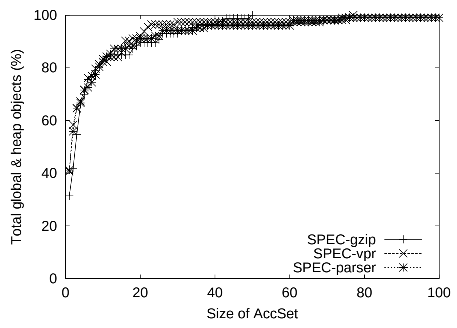


(d) *Real applications (2)*

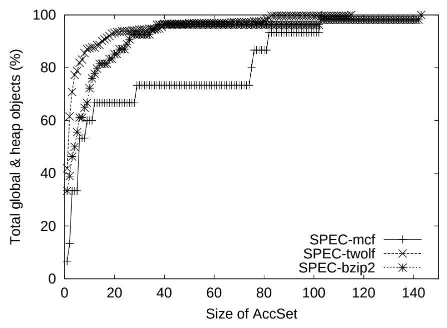
Figure 4.1: Cumulative distribution of global objects' AccSet size for six SPEC2000 benchmarks and six real applications. Each cumulative distribution curve gives the percentage of global data objects whose AccSet sizes are smaller than or equal to a given size. A high percentage for a small size means that most objects have small AccSets sizes. Note that the SPEC-gzip and gzip-1.2.4 applications are different.

To find the average size and stability of AccSets, we collect the AccSets for all global and heap objects (global and heap variables) in the twelve programs, using multiple runs with different inputs. We then examine the cumulative distribution of the AccSet sizes and measure the similarity of AccSets across 5 runs with different inputs.

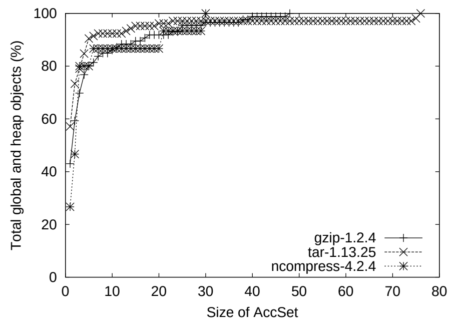
Figure 4.1 and Figure 4.2 show the cumulative distributions of the global objects' AccSet sizes and the global and heap objects' AccSet sizes, respectively, for the six SPEC2000 benchmarks and six real applications. Considering only global objects, for the six SPEC2000 benchmarks, 90%-96% of the global objects in vpr, parser and twolf have AccSet sizes less than 5, and 80%-85% of the global objects in mcf, gzip and bzip2 have AccSet sizes less than 15. For the six real applications, around 85-100% of the global objects have AccSet sizes less than 10. Looking at both global and heap objects together, for the six SPEC2000 benchmarks, 85%-90% of the global



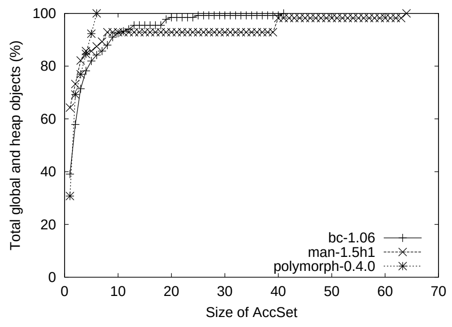
(a) *SPEC2000 benchmarks (1)*



(b) *SPEC2000 benchmarks (2)*



(c) *Real applications (1)*



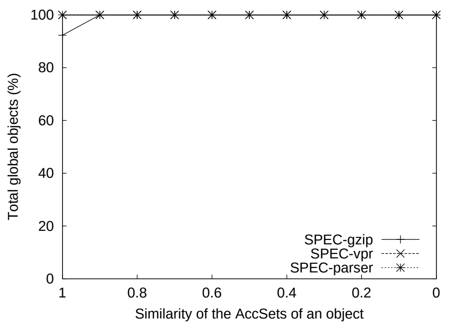
(d) *Real applications (2)*

Figure 4.2: Cumulative distribution of global and heap objects' AccSet size for six SPEC2000 benchmarks and six real applications. Each cumulative distribution curve gives the percentage of global and heap data objects whose AccSet sizes are smaller than or equal to a given size. are different.

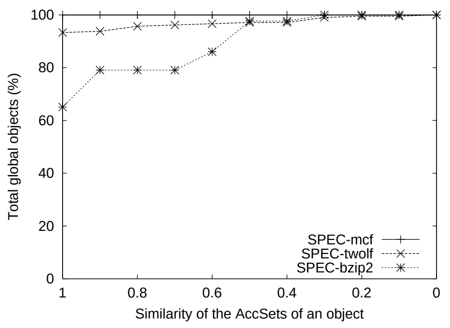
and heap objects in vpr, parser and twolf have AccSet sizes less than 15, and 80%-85% of the global and heap objects in gzip and bzip2 have AccSet sizes less than 15. The AccSet sizes for are larger, but still 67% of the global and heap objects have AccSet sizes less than 11. For the six real applications, around 85-100% of the global and heap objects have AccSet sizes less than 10. In other words, in general 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

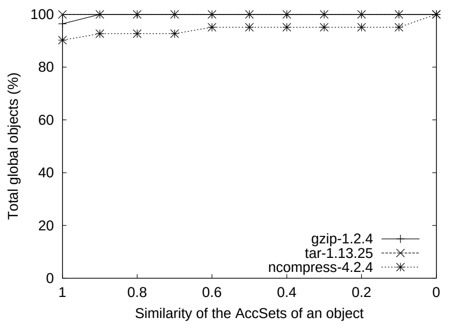
$$Similarity(OBJ) = \frac{|\cap(S_1, S_2, \ldots, S_n)|}{|\cup(S_1, S_2, \ldots, S_n)|}$$



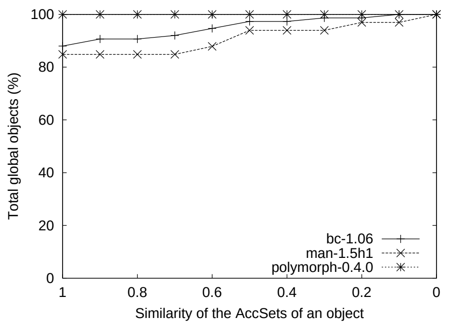
(a) *SPEC2000 benchmarks (1)*



(b) *SPEC2000 benchmarks (2)*



(c) *Real applications (1)*

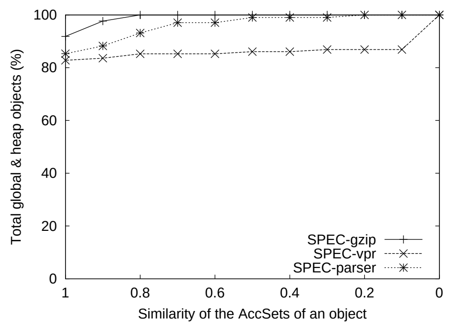


(d) *Real applications (2)*

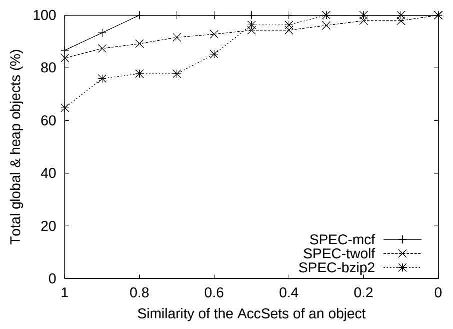
Figure 4.3: Cumulative distribution of global objects' AccSet similarity across 5 runs for six SPEC2000 benchmarks and six real applications. Each cumulative distribution curve shows the percentage of global data objects whose AccSets have a similarity greater than or equal to a given value. A high percentage at a value close to 1 indicates that most global objects' AccSets are similar across different runs. Note that the x-axis starts at 1 and goes to 0.

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.

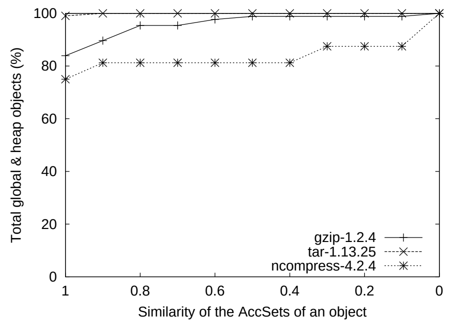
Figure 4.3 and Figure 4.4 show the cumulative distributions of the global objects' AccSet similarity and the global and heap objects' AccSet similarity for different runs. The figures show that in general most objects have a similarity close to one, which indicates that most AccSets are stable across different runs. For the six SPEC2000 benchmarks, 93-100% of the global objects' AccSets in five of them (except bzip2) have similarity values greater than 0.96. In bzip2, 79% of the global objects' AccSets have have similarity values greater than 0.9. 85-96% of the global and heap ob-



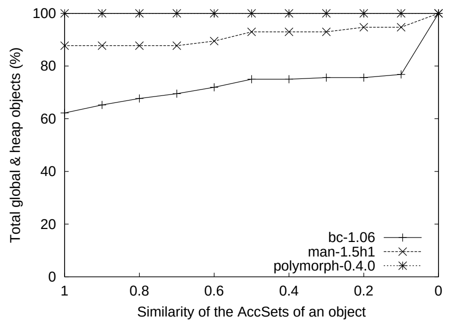
(a) *SPEC2000 benchmarks (1)*



(b) *SPEC2000 benchmarks (2)*



(c) *Real applications (1)*



(d) *Real applications (2)*

Figure 4.4: Cumulative distribution of global and heap objects' AccSet similarity across 5 runs for six SPEC2000 benchmarks and six real applications. Each cumulative distribution curve shows the percentage of global and heap data objects whose AccSets have a similarity greater than or equal to a given value.

jects' AccSet in gzip, parser, mcf and twolf have similarity values greater than 0.91, and 83% and 75% of the global and heap objects' AccSet in vpr and bzip2 have similarity values greater than 0.91, respectively. For the six real applications, as shown in Figures 4.3(c) and 4.3(d), around 84-100% of the global objects' AccSets have similarity values greater than 0.97. In Figures 4.4(c) and 4.4(d), 84-100% of the global and heap objects' AccSet have similarity values greater than 0.97 in gzip, tar and polymorph, and 70-88% of the global and heap objects' AccSet have similarity values greater than 0.7 in ncompress, man and bc. These results show that AccSets are quite stable across multiple runs with different inputs.

Further validation of our observations on PC-based invariants is provided by the data in Section 4.6.

4.3 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*.

4.3.1 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 4.1.

AccMon uses iWatcher to catch all memory accesses to monitored memory objects and trigger a monitoring function at such accesses [ZQL+04]. 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 4.1: 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 4.3.2 for details).

The CLB resides in the processor. Figure 4.5 shows how it interacts with the different pipeline stages and the iWatcher trigger bit. More details are given in Section 4.3.2.

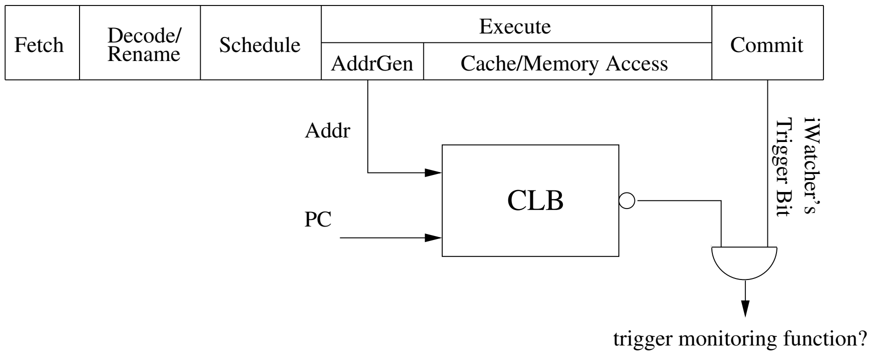


Figure 4.5: Interaction of the CLB with the processor pipeline and the iWatcher trigger bit.

We modify the Cetus compiler [LJE03] 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 4.3.4. 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 4.3.3 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.

4.3.2 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 4.5, 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 [Blo70] 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 4.6 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 [PLL02]. We use a direct index instead of a hash function to simplify the logic as much as possible.

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 4.5). 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 object 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

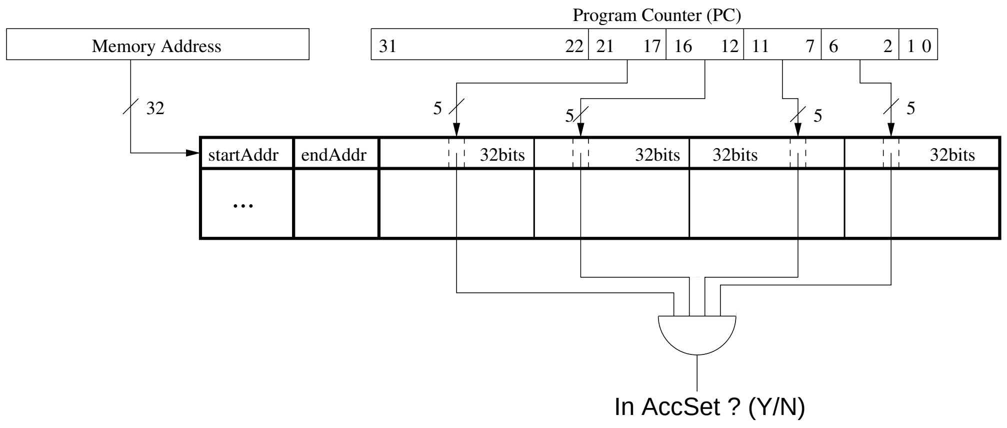


Figure 4.6: Implementation of the CLB using a Bloom filter.

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 4.8 in Section 4.6.2, 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.

4.3.3 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 4.3.4. 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 [ECC01, HL02].

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.

4.3.4 Design Issues

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. Note that, the monitoring scheme for heap objects is language and runtime dependent. Our scheme assumes that heap objects are explicitly allocated and freed and they are not moved during their lifetime. A different scheme is required for languages with garbage collection.

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 [BZ93, CCJA98, LW94] 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 [BZ93, CCJA98, LW94], 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.

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 [LJE03] 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.

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 [ZQL$^+$04]. 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.

4.4 Software Implementation – AccMon-S

To demonstrate the efficiency of the hardware implementation, and to evaluate the PC-based invariant detection idea with real server applications (our simulation of AccMon described in Section 4.5 does not support running server programs), we also implemented the PC invariants purely in software using the PIN binary instrumentation tool [LCM+05].

Like AccMon, the key task of AccMon-S is to collect and maintain PC invariants information, and use this information for bug detection. Specifically, for all global variables, all heap variables, and all stack locations storing the return addresses, AccMon-S learns the set of PCs accessing these variables during normal executions (at training phase), and then uses the PC sets to check for possible violations during the detection phase. The global variables are identified by using the symbol table, the heap variables are identified by wrapping memory allocation calls, and the stack return address locations are gotten using the PIN tool. In software, PC sets information is collected by PIN at every access to the interested locations (all accesses will be checked if they go to an interested location, and if yes, collect the PC), and maintained in the PCT table. This is training. For detection, the PCT table will be checked for possible violations. Again, every access will be intercepted, and if it goes to an interested location, checked for PC invariant violation.

4.5 Evaluation Methodology

4.5.1 Methodology Overview

We use cycle-accurate execution-driven simulations to model a workstation with iWatcher [ZQL+04] and AccMon functionality. The parameters of the architecture are shown in Table 4.2. The architecture includes a 4-context SMT processor with optional TLS support. The experiments for software implementation AccMon-S are conducted on real machines with a 2.4GHz Pentium 4 processor, 512KB L2 cache, 1GB of memory, and a 100Mbps Ethernet connection. For server applications, we run servers on one machine and clients on another.

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 4.2: Architecture modeled.

We compare AccMon and AccMon-S to the Purify [HJ92] and CCured [CHM$^+$03, NMW02] (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 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 six applications to run with CCured. For application *tar*, *apache* and *squid*, we are unable to run it with CCured despite great manual effort. Therefore, we estimate the behavior based on CCured's functionality, but we cannot predict the overhead. In contrast, AccMon and AccMon-S do not require any manual modification of an application's source code.

We run Purify and CCured on the same real machine as that used for AccMon-S. Since AccMon runs on a simulator, we cannot compare the absolute execution time of AccMon with that of AccMon-S, Purify and CCured. Instead, we compare their execution overheads relative to runs without any monitoring.

Since existing value-based invariant detection tools such as DIDUCE [HL02] do not work with C/C++ programs, we cannot quantitatively compare AccMon and AccMon-S 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 [ECGN99, ECGN00, HL02] describing these tools.

4.5.2 Evaluated Applications

We have conducted two sets of experiments. The first set uses buggy applications to evaluate the functionality and overheads of AccMon and AccMon-S for software debugging. Note that, the two real buggy server applications *apache* and *squid* are only used to evaluate AccMon-S, since our simulator does not support running them so far. The second set further evaluates the overheads of AccMon with bug-free SPEC benchmarks.

For the first set of experiments, we selected nine buggy programs that exhibit a broad spectrum of memory-related bugs. Table 4.3 gives the details about these applications and their bug characteristics. Two of them are real server applications, namely apache and squid. Some of the non-server applications, such as tar-1.13.25 and bc-1.06, are relatively large, with more than 17K lines of code. Note that, we use the server applications to only evaluate AccMon-S, since our AccMon simulator does not support server programs.

The eight 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 eight programs are: *gzip*, *man*, *polymorph*, *ncompress*, *tar*, *bc*, *apache* and *squid*. *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. *apache* is a commonly used web server. *squid* is a web cache and proxy server.

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	Wrong pointer assignment caused by copy-paste
polymorph -0.4.0	716	Real-Reported	polymorph.c: lines 193&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
apache -1.3.27	283K	Real-Reported	mod_alias.c: line 311	Stack	AliasMatch expression in config file with more than 10 captures corrupts stack return address
squid -2.3.s5	93K	Real-Reported	ftp.c: lines 1024&1027	Heap	Mis-calculation of the request length due to special chars causes heap object overflow

Table 4.3: 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.

To demonstrate the unique bug-detection strengths of PC invariants, 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 a recent 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 [CYC$^+$01, LLMZ04]. Since we cannot run Linux in our simulator, we built a simple benchmark (linux-simple) to measure the effectiveness of PC invariants 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 and AccMon-S can be used in any run (normal or abnormal) to detect potential bugs. To demonstrate the capability of AccMon and AccMon-S to detect a bug, we need to use abnormal runs, as do other run-time bug detection studies [CHM$^+$03, ECGN99, ECGN00, HL02, NMW02]. To do that, we use bug-exhibiting inputs to generate these abnormal runs. But this does not mean that AccMon and AccMon-S need bug-exhibiting inputs to function.

The second set of experiments evaluates AccMon overheads using six bug-free SPEC2000 applications running the Test input data set, namely gzip, parse, vpr, mcf, twolf and bzip2.

4.6 Experimental Results

4.6.1 Overall Results

For the seven non-server buggy applications, AccMon and AccMon-S detect all ten bugs, and found one previously unreported (to the best of our knowledge) bug. Table 4.4 compares the effectiveness and the overhead of AccMon, AccMon-S, Purify, CCured, and value-based invariant detection tools. For the two server applications, AccMon-S detects the two test bugs. Table 4.5 compares AccMon-S, Purify, CCured, and value-based invariant detection tools on server pro-

Application	**AccMon**		**AccMon-S**	**Purify**		**CCured**		Value-Based Invariants
	Bug Detected?	Over-head	Over-head	Bug Detected?	Over-head	Bug Detected?	Over-head	Bug Detected?
ncompress-4.2.4	Yes	0.24X	10.39X	No	8.33X	Yes	0.17X	Difficult*
linux-simple	Yes	0.60X	57.83X	No	32.84X	No	5.50X	Difficult
polymorph-0.4.0	Yes	0.76X	42.65X	No	44.65X	Yes	0.50X	Difficult
gzip-1.2.4	Yes	0.94X	39.32X	Yes	42.45X	Yes	0.40X	Easy
tar-1.13.25	Yes	1.04X	35.42X	Yes	13.68X	NR(Yes)	NR	Difficult
man-1.5h1	Yes	1.50X	26.08X	Yes	4.83X	Yes	0.69X	Easy
bc-1.06	Bug1: Yes Bug2: Yes bc-lib: Yes bc-free: Yes	2.88X	52.36X	Yes Yes No Yes	46.11X	Yes Yes No No	1.35X	Depends Difficult Depends Difficult

Table 4.4: Overall results on non-server applications. 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 requirements; *Difficult in column 9 means that we could not find an effective way to detect the bug using value-based invariants.

Application	**AccMon-S**		**Purify**		**CCured**		Value-Based Invariants
	Bug Detected?	Over-head	Bug Detected?	Over-head	Bug Detected?	Over-head	Bug Detected?
apache-1.3.27	Yes	32.17X	No	37.50X	NR(Yes)	NR	Difficult*
squid-2.3.s5	Yes	43.22X	Yes	17.21X	NR(Yes)	NR	Difficult

Table 4.5: Overall results for AccMon-S on servers. 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 requirements; *Difficult in column 8 means that we could not find an effective way to detect the bug using value-based invariants.

grams.

The default setup for AccMon is a TLS-enabled iWatcher with an 8-entry CLB, and with only write accesses monitored. The results of AccMon are obtained using this default setup unless otherwise mentioned in Sections 4.6.2 and 4.6.3. 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). For

AccMon-S, we evaluate its effectiveness using the same training and detection inputs as those for AccMon, and it has the same debugging functionality as AccMon-S. Therefore, we only report its overhead here. The default setup for AccMon-S has no software CLB because a software CLB does not provide performance benefits, and monitors only write accesses.

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, AccMon and AccMon-S are from actual experiments since these tools work with all applications.

Application	Training Overhead	#Monitored Accesses	#Monitored Accesses after the CLB	Monitored Sizes (Bytes)	Max # of Monitored Objects	Ranking of the Bug	# False Alarms
ncompress-4.2.4	1.20X	158995	13	806180	60	1	1
linux-simple	1.64X	11769	5	3352	43	1	0
polymorph-0.4.0	0.99X	520	4	10472	53	1	8
gzip-1.2.4	3.06X	274594	44441	396641	190	1	0
tar-1.13.25	1.52X	29729	102	88142	432	2	2
man-1.5h1	2.83X	27909	921	187898	644	1	0
bc-1.06	3.98X	260813	84716	467005	454	1,2,3,4	0

Table 4.6: Detailed results for AccMon on non-server programs. 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.

Table 4.6 shows the detailed AccMon results on the non-server programs. The detailed AccMon-S results on these programs are very similar, except for AccMon-S's training overheads, which are very close to AccMon-S's detection overheads (shown in Table 4.4). Table 4.7 shows the detailed AccMon-S results for the server programs.

Functionality From Table 4.4, we see that AccMon and AccMon-S 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,

Application	Training Overhead	#Monitored Accesses	#Monitored Accesses after the CLB	Monitored Sizes (Bytes)	Max # of Monitored Objects	Ranking of the Bug	# False Alarms
apache-1.3.27	33.04X	50001	458	586190	189	1	0
squid-2.3.s5	43.86X	700014	31150	5243260	9049	1	4

Table 4.7: Detailed results for AccMon-S on server programs. The column on number of monitored accesses after the CLB is only for the bug-detection phase.

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 the 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 six applications to run with CCured (except tar-1.13.25). Of these six applications, CCured misses the bug in linux-simple, and the bc-free and bc-lib bugs in bc-1.06. The reasons for missing the three bugs are, respectively: 1) the bug in linux-simple does not violate any programming rule, 2) CCured uses garbage collection to manage memory allocation (explained in Section 4.5), and 3) CCured cannot monitor accesses by a third-party library whose source code is unavailable. For the bug in tar-1.13.25, we conservatively estimate that CCured would catch it.

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.

For the two servers (apache and squid), AccMon-S can catch the bugs (as shown in Table 4.5) by using PC-based invariants. AccMon could also detect them if the simulator supported running server programs, since it uses the same detection technique as AccMon-S.

Like all other bug detection tools, there are some bugs that AccMon and AccMon-S cannot detect, for example, memory leaks, because PC-based invariants can only be used to catch bugs causing memory corruption; memory leaks do not corrupt the memory.

Overhead Table 4.4 shows that AccMon has an acceptable overhead, which is significantly lower than Purify's and AccMon-S's. AccMon has an overhead of only 0.24-2.88 times, even though most applications monitor hundreds of KBytes data (Table 4.6). 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 [CHM$^+$03]), and orders of magnitude less than AccMon-S's overheads (10.39-57.83 times). For example, in ncompress-4.2.4, AccMon monitors a total of 0.8 MBytes of memory (Table 4.6) and almost 92.1% of dynamic memory accesses (not shown in the tables), but it adds only 24% overhead (Table 4.4). For server applications, AccMon-S introduces 32 times of overhead for apache and 43 times of overhead for squid (Table 4.5). We believe AccMon will have much less overheads on servers too.

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 4.6).

False Alarm Rate AccMon and AccMon-S have a very low false alarm rate, and the bugs are ranked high in the error reports. Table 4.6 shows that for non-server applications, there are no false alarms for four applications, and only 2-8 false alarms for two applications. Moreover, all bugs are ranked in the top 2 entries of the error reports. For server programs, there are no false alarms for apache and only 4 false alarms for squid, as shown in Table 4.7. Therefore, a programmer can easily identify real bugs.

4.6.2 Impact of the CLB

Figure 4.7 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 a 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 (bringing 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.

For AccMon-S, the software CLB has little impact on its overheads because of two reasons. First, The overheads of AccMon-S come from not only the monitored accesses, but also the non-monitored accesses, since it needs to look in the PCT table for every access to determine if the access is monitored or not. A CLB can only reduce the overhead for monitored accesses, and does not help with the non-monitored accesses, which are a large fraction of all accesses. Second, the software CLB lookup time is much larger than the negligible hardware CLB lookup time, and close to the PCT table lookup time. Therefore, it cannot save much even for monitored accesses.

Table 4.8 gives the 4-entry and 8-entry CLB hit ratios for the seven non-server applications in AccMon, and the CLB hit ratios for all nine applications in AccMon-S. As we can see, the CLB hit ratios in AccMon and AccMon-S are similar, and the slight differences are caused by different binaries and memory allocation layouts.

Figure 4.7 shows that the CLB reduces AccMon's overheads by a significant (28.9-80.6%) amount. For example, the overhead of AccMon with gzip is reduced by a factor of 3.17 from

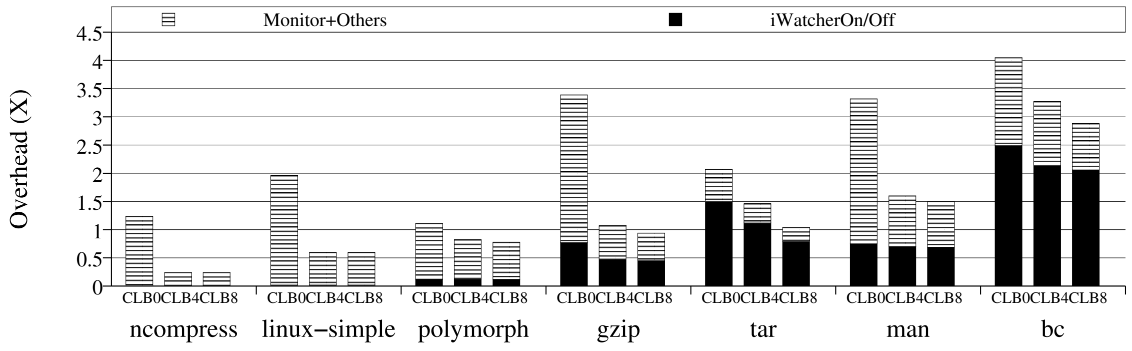


Figure 4.7: Overhead introduced by AccMon with and without the CLB.

	#entries	ncompress	linux-simple	polymorph	gzip	tar	man	bc	apache	squid
AccMon	4	99.9%	99.9%	99.2%	80.1%	51.5%	96.3%	43.2%	-	-
	8	99.9%	99.9%	99.2%	83.8%	99.7%	96.7%	67.5%	-	-
AccMon-S	4	99.9%	99.8%	98.6%	75.2%	57.4%	99.4%	44.9%	99.70%	91.98%
	8	99.9%	99.8%	98.8%	79.8%	98.7%	99.4%	64.9%	99.76%	94.97%

Table 4.8: CLB hit ratios for monitored accesses.

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 4.8. 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 4.7.

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.

For apache and squid, since the hit ratios are very high (>90%) for both 4-entry CLB and 8-entry CLB, we can predict that if we could run the two servers on AccMon, their monitoring overheads would be significantly reduced by using CLB.

4.6.3 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 4.3.4. Figure 4.8 compares the overheads of monitoring both read/write accesses (rw) and write only accesses (wo). Table 4.9 shows the number of monitored accesses before and after the CLB filtering process for both rw and wo.

In Figure 4.8, 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 4.9), resulting in a 61.9% reduction in overhead (Figure 4.8).

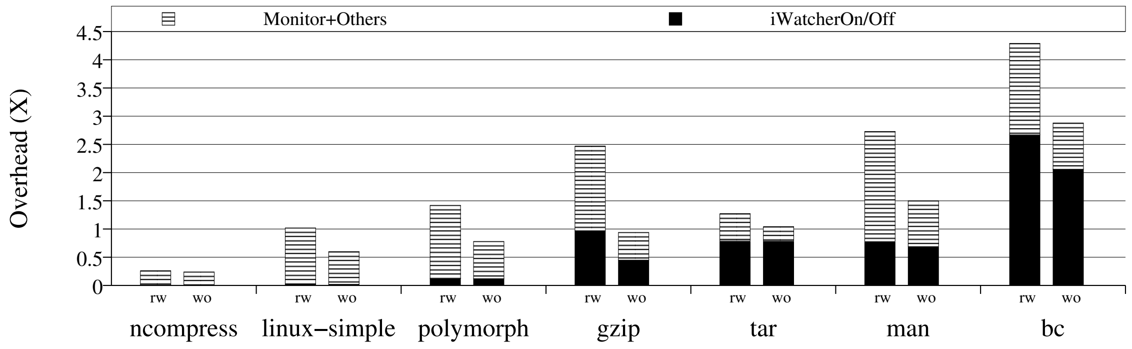


Figure 4.8: Overhead of monitoring different types of accesses.

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 4.9: 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 4.9). However, linux-simple and polymorph show a significant overhead reduction in Figure 4.8. The reason is that going from rw to wo causes a significant reduction of monitored accesses before the CLB for these applications (Table 4.9). 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 4.10, 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.

	ncompress	linux-simple	polymorph	gzip	tar	man	bc
Before Pruning	1	0	8	1	8	0	0
After Pruning	0	0	8	0	2	0	0

Table 4.10: Number of false alarms before and after pruning.

4.6.4 Overhead with SPEC Benchmarks

Overall Results To measure AccMon overheads on bug-free applications, we run six SPEC2000 benchmarks, namely gzip, parser, vpr, mcf, twolf and bzip2[1], with the Test input data set. The experiments use the default setup for AccMon: TLS-enabled, 8-entry CLB and only monitoring write accesses.

[1] For 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, vpr and bzip2, we only run them for 300 million instructions.

Table 4.11 shows the results for the six SPEC benchmarks. The overheads range from 1.29 to 4.14 times for all the six applications. The size of monitored memory is 6.5-117 MBytes. Recall that the overhead is broken down into iWatcherOn/Off overhead and monitoring plus other overhead. For all the six 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. As shown on Table 4.11, the number of monitored accesses per 1M instructions is large for all the six applications, ranging from 12k to 73k. Fortunately, most of these accesses are filtered by the CLB, as indicated by the high CLB hit ratios, 83.3%-99.9% for all these applications except twolf. This significantly reduces the monitoring overhead which, together with other overheads (described in Section 4.6.2), accounts for the non-iWatcherOn/Off component of the total overhead. For example, Table 4.11 shows that, with CLB, mcf and bzip2 have very few monitored accesses per 1M instructions, 0.12 and 0.18 respectively. Therefore, they suffer very small monitoring overheads, which are indicated by the very small monitoring+other overheads (overhead-iWatcherOn/Off overhead), 0.04X for mcf and 0.19X for bzip2.

Appli-cation	Overhead	iWatcherOn/Off Overhead	#Monitored Accesses per 1M Inst.	#Monitored Accesses per 1M Inst. after CLB	CLB Hit Ratios (%)	Monitored Sizes (Bytes)
gzip	1.29X	0.80X	73215.78	5698.34	92.2	13533869
parser	3.16X	2.13X	12442.43	77.48	99.4	10244523
vpr	1.73X	0.95X	45238.16	7561.99	83.3	6585702
mcf	2.19X	2.15X	39509.37	0.12	>99.9	117385440
twolf	4.14X	1.94X	46267.82	30400.77	34.3	7988353
bzip2	1.35X	1.16X	44320.44	0.18	>99.9	25429319

Table 4.11: AccMon behavior for SPEC applications.

Impact of the CLB Figure 4.9 shows AccMon's overheads with different CLB entries, no CLB (CLB0), 4-entry CLB (CLB4) and 8-entry CLB (CLB8), on the six SPEC2000 benchmarks. Table 4.12 gives the 4-entry and 8-entry CLB hit ratios for the six benchmarks.

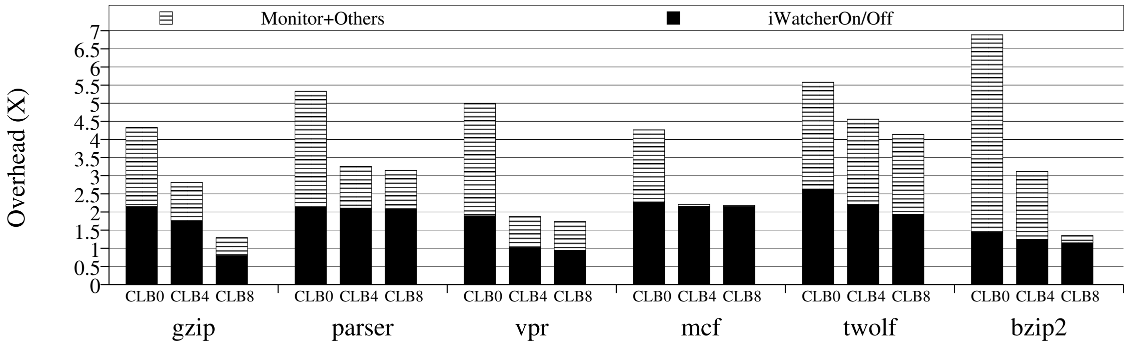


Figure 4.9: Overhead introduced by AccMon with and without the CLB.

#Entries	gzip	parser	vpr	mcf	twolf	bzip2
4	50.5%	95.7%	79.8%	99.5%	26.7%	61.2%
8	92.2%	99.4%	83.3%	>99.9%	34.3%	>99.9%

Table 4.12: CLB hit ratios for monitored accesses.

As shown in Figure 4.9, comparing with the baseline case (no CLB), the overheads are significantly reduced by 18.3-62.5% using 4-entry CLB and 25.8-80.4% using 8-entry CLB. For example, in gzip, 4-entry CLB filters 50.5% of the monitored accesses as shown in Table 4.12. As such, there is a 51.8% reduction from 2.18 times to 1.05 times in the monitor+other overhead, and the total AccMon's overhead is reduced from 4.33 times to 2.82 times. With a 8-entry CLB, the monitor+other overhead is further reduced by 77.5% from 2.18 times to 0.49 times, and the total overhead is reduced by a factor of 3.36 from 4.33 times to 1.29 times, because 92.2% (Table 4.12) monitored accesses are filtered.

Among the six benchmarks, gzip and bzip2 have large overhead reduction from a 4-entry CLB to a 8-entry CLB, because their CLB hit ratios greatly increase, from 50.5% to 92.2% for gzip and from 61.2% to 99.9% for bzip2 as shown in Table 4.12. The overheads of the other four benchmarks are only slightly reduced, due to the small CLB hit ratio improvement (0.4-7.6%).

4.7 Summary

This chapter made three contributions. First, it proposed the novel idea of PC-based invariants to detect memory-related bugs. Second, it efficiently implemented this idea using previously proposed iWatcher hardware framework and proposed the CLB, a new architectural extension to the iWatcher framework that significantly reduces the overhead of PC-based invariant debugging. The hardware implementation called AccMon leverages architectural, run-time system and compiler support. It detects ten 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.

It also used the binary instrumentation tool PIN [LCM$^+$05] to build a pure software implementation of PC-based invariant detection tool called *AccMon-S*. AccMon-S does not require hardware support, but has much higher execution overhead (10.4X-57.8X), so it can only be used for in-house bug detection instead of bug detection during production runs. Besides detecting all ten bugs tested in AccMon, AccMon-S also detected two real bugs in two large real-word server applications, Apache and Squid with 0-4 false alarms.

Since PC-based invariants detection is a statistics-based approach, it can catch bugs that do not violate any programming-based rules. For example, there are a few bugs in our experiments that are detected by PC invariants but are missed by other tested tools such as Purify [HJ92] and CCured [NMW02, CHM$^+$03].

Chapter 5

iChecker: Incremental Data Structure Consistency Check

5.1 Overview

Consistency of a data structure means that the states of the data structure satisfy certain properties during the entire program execution except within some operations that intentionally violate the properties while evolving the data structure from one consistent state to another. For example, a sorted doubly-linked list is consistent if the list nodes have values in order and every list node is appropriately linked to its predecessor and successor. Such a list should remain consistent, for instance, after an insertion of a new node in the list. But, during the insertion, the list is not doubly-linked at some point.

Data-structure consistency is critical for many programs [DR03, DR05]. Unfortunately, software faults may corrupt the data and cause inconsistency, which can make the program enter an error state, generate wrong outputs, and even crash. For example, in the file-system case study (Section 5.4.5), a fault in updating the inode bitmap causes an inconsistency. Consequently, the program re-assigns an already used inode to a new file and loses the information of the original file that was stored in this inode.

One approach to ensuring data-structure consistency is to statically analyze the code that can potentially access the target data structure, for example using static analysis such as shape analysis [SRW02, MS01], theorem proving [Pau94, BW96], or a combination of both [ZLKR04]. Static analysis is highly automated and can be effectively used to verify several classes of consistency properties and code. However, the inherent limitations of static analysis make it either incomplete or unsound [FLL^{+}02]. Interactive theorem provers can be used, in principle, to prove arbitrary properties, but they require a high level of expertise from the users and a large amount of manual effort. Even a combination of static analysis and theorem proving has not been shown practical for verifying the consistency properties for arbitrarily complicated data structures, especially those in programs that are written in industrial programming languages such as C/C++.

Another, more widely used approach to ensuring data-structure consistency is to perform run-time checks at the appropriate program points, e.g., at the entrance and the exit of relevant functions. The checks are performed using assertions that either the programmers insert manually or a compiler inserts automatically [GJKW97, MA87]. Each assertion typically calls a checking function (checker). This checking code can be manually written by programmers (in most practical cases) or automatically generated from programmer-provided specifications [DR03, DR05].

The programmer would like to perform *frequent* run-time checks for the early detection of data structure inconsistency, because it can limit the amount of damage caused by the error propagation and reduce the time and effort needed for fault localization [DCRR04]. Unfortunately, the traditional consistency check usually needs to traverse the entire data structure to determine that the consistency properties hold. (A checker may find that some property does not hold after traversing only a part of the structure.) Such global checks are fairly expensive and can incur large overhead for large data structures with frequent checks, up to 416 times as reported in our experimental results. Such big overhead prevents global checks being frequently invoked. consequently, a data-structure corruption cannot be detected early enough.

A key observation about data structures is that a typical operation modifies only a small, localized part of the data structure. The effects of such small modifications on the consistency are mostly local. Thus, it is often unnecessary to traverse the entire data structure to check the consistency. If we start from a consistent state (e.g., after passing the last check) and the program modifies only a small part of the state, it is sufficient to check the consistency of only the affected part of the data structure. Consider, for example, a sorted doubly-linked list that was consistent and in which a new node is inserted between two existing nodes. To verify if the data structure is still consistent, we intuitively need to check only the new node and some existing node(s) whose pointers are modified; we do not need to traverse the entire list. We refer to a check on a partial structure (e.g., a node in a list, a subtree in a tree, or an element in an array) as a *local check*. In contrast, we refer to a traditional check of the entire structure as a *global check*.

It is conceivable that the incremental check can significantly reduce the time for checking data consistency compared with the global check, especially when the data structure is large, and there are only small modifications between two consecutive checks [DR03]. To perform incremental checking, an effective solution is to borrow ideas from incremental computation [DRT81, YS88, ABH02, RR93, LT95, ZL98]. It computes the new output *incrementally* by reusing parts of the old computation (instead of by recomputing the entire output from the scratch), when the new input and old input differ slightly. Unfortunately, most of previous works on incremental computation were done in the context of (pure) functional languages and did not consider mutable data structures as used in imperative languages. To the best of our knowledge, there is no previous work on incremental computation/checking of programs written in C, which is still one of the dominant programming languages in industry, especially for performance critical systems and server software. The main challenge for incremental checking of consistency in C programs is that any part of a data structure may be potentially mutated by any write, including the "sneaky" writes caused by dangling pointers, buffer overruns, or other memory corruptions.

This chapter presents our incremental checking framework called iChecker that leverages a simple, previously proposed hardware iWatcher [ZQL$^+$04] to provide an iChecker library for efficient, incremental, run-time consistency checks of mutable data structures in C programs. The basic idea of iChecker is to perform a consistency check with a local check (on the parts that need to be checked due to the modifications since the last consistency check) instead of with the global check. Besides invoking the consistency checks (as in traditional global check), the programmer only needs to indicate the data structure to be checked and its associated local check function, and call a few library calls in limited places. It is iChecker's responsibility to figure out on which part of the data structure to perform this local check function. To achieve this functionality completely in software would require monitoring almost every memory access except a few fraction that can be eliminated via sophisticated static program analysis.

More specifically, iChecker exploits the precise memory monitoring feature of iWatcher to automatically build a *dependency set* for each part of the data structure (e.g., a node) to record what other data this part's consistency depends on. During the execution, iChecker also leverages iWatcher to keep track of updates to data structures. At the check time, only those parts (of data structures) that are affected by the updates are checked using the local checker function provided by programmers.

We evaluate iChecker using four case studies: two micro-benchmarks (sorted doubly-linked list and binary search tree) and two larger applications (a simple file system and an interactive game). These case studies represent different types of structures and consistency checks, namely pointer-based and array-based data structures, iterative and recursive global checks, and read-only and read-write local checks.

Our experimental results show that iChecker requires only modest changes (*25–108* lines including the global checkers), which are 10–56 lines more than the modifications for traditional global checks and only account for 0.1%–21% of the original code. Since iChecker leverages the iWatcher support to efficiently catch the memory accesses to the data structure for both building dependency sets and tracking modifications, for large data structures the time overhead of using iChecker for incremental check is *1.1–155* times (*23.3* times on average) less than that of the global check with 0.3–17.9 times (11.2 on average) of space overhead, and the time overhead is estimated to be *2–7* times less than a software-only implementation for incremental check. Note that iChecker offers more improvement for the two larger applications in terms of the much smaller time and space overhead, as well as the much smaller percentage modifications over the original code, than for the two micro-benchmarks.

iChecker works for mutable data structures and thus applies to many C programs, including even the low level system code such as parts of an operating system, as demonstrated in our file system case study.

The main modification required by iChecker is that the programmer needs to provide the local checker based on the global checker. The programmer needs to ensure soundness of the local checkers with respect to the global checker, similarly as the programmer needs to ensure soundness of the global checker with respect to the desired properties. However, since the programmer's effort can bring significant benefits in some cases, like *155* times less overhead in the file system example, we expect the programmer to be willing to invest such effort selectively.

iChecker provides a strong demonstration case of leveraging new hardware innovations in performing software engineering tasks. This is important for two reasons. First, it can make a strong influence on the hardware designers to include more extensions in the near-future microprocessors to enhance software quality and programmers productivity. Second, when such new hardware is ready, iChecker can immediately take advantage of the hardware.

The remainder of this chapter is organized as follows. In Section 5.2, we use an example to show the iChecker interface. In Section 5.3, we describe the iChecker framework and its implementation. Then, we use four case studies to demonstrate the use of the iChecker framework in Section 5.4, followed by the summary in Section 5.5.

5.2 iChecker Interface and Example

We use a sorted doubly-linked list as a simple example to illustrate how the programmers can use the iChecker framework for incremental consistency checking. Figure 5.1 shows three instances of sorted doubly-linked lists. There are two consistency properties: the nodes should have values sorted in the ascending order, and every node should be appropriately linked to its predecessor and successor.

Figure 5.2 shows the code for basic operations of the list. The highlighted code represents modifications for incremental checking (as discussed below). Figure 5.2 also shows the function *isDoublyLinkedSortedList* that performs a global consistency check, traversing the entire list and checking that each node is properly linked and has a value in order. *checkListNode* is the local check function, which, for a given node, checks if its successor's value is not smaller than its own and its successor is pointing back to this node itself. Note that in our example the relationship between any pair of nodes are associated only with the first node to avoid redundant checks.

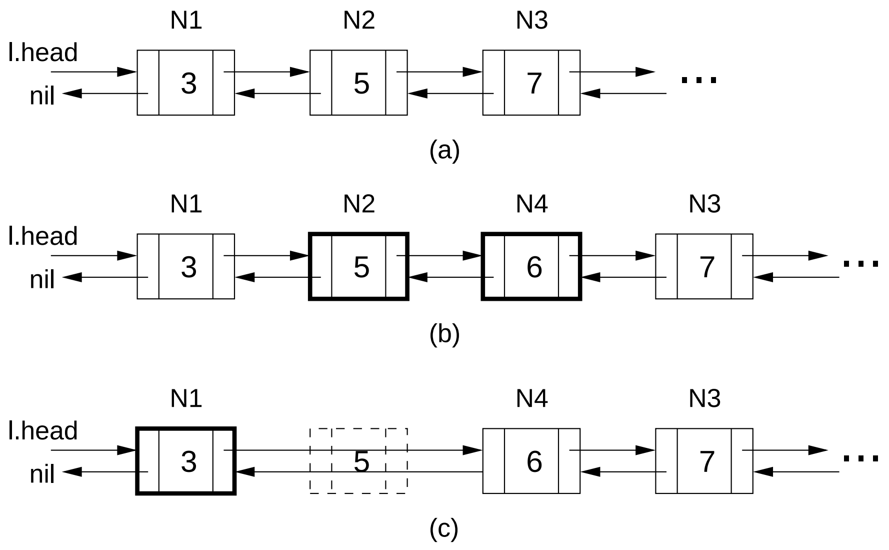


Figure 5.1: Sorted doubly-linked list: (a) the original list; (b) the list after insertion of N4; (c) the list after removal of N2.

5.2.1 Modifications and iChecker Interface

The modifications shown in Figure 5.2 use the iChecker interface shown in Figure 5.3. They can be inserted manually by the programmer or automatically by the compiler. We next explain each modification and the general interface (one data structure and four functions) associated with it:

ICheckStruct: This structure from our iChecker library keeps track of the data required for incremental checking. In the example, an instance of *ICheckStruct* is added to each list.

newICheckStruct: This function initializes the corresponding *ICheckStruct* that stores a function pointer for the global check *globalFunc*, an address to pass as a parameter to the global check *globalPara*, a function pointer for the local check used by incremental check *localFunc*, and the size of a data structure part that forms the basic unit for the incremental check *partSize*. In the example, *newICheckStruct* is called to initialize the above data structure when the list is initialized.

insertIntoICheckStruct: This function adds a data structure part to the new set of the $ICheckStruct$

```
typedef struct node {
  int val;
  struct node *next;
  struct node *prev;
} ListNode;

typedef struct list {
  ListNode *head;
  int size;
  ICheckStruct *icStruct;
} List;

List *newList() {
  List *l=(List *)malloc(sizeof(List));
  l->head=NULL;
  l->size=0;
  l->icStruct=
  newICheckStruct(&isDoublyLinkedSortedList,
               (void*)l, &checkListNode,
               sizeof(ListNode));
  return l;
}
```

```
void insertIntoList(List *l, int val) {
  ListNode *cur;
  /*insert the node *cur with cur->val==val*/
  ......
  insertIntoICheckStruct(l->icStruct, (void*)cur);
}

void removeFromList(List *l, int val) {
  ListNode *cur;
  /*remove the node *cur with cur->val==val*/
  ......
  deleteFromICheckStruct(l->icStruct, (void*)cur);
}

int main(int argc, char **argv) {
  List *l = newList();
  ......
  insertIntoList(l, 6);          /* line A */
  if (!genericICheck(l->icStruct)) ...;
  ......
  removeFromList(l, 5);      /* line B */
  if (!genericICheck(l->icStruct)) ...;
  ......
}
```

```
/* isDoublyLinkedSortedList is the global checker, checkListNode is the local checker which is also used by the global checker */
```

```
int isDoublyLinkedSortedList(void *args) {
  List *l=(List *)args;
  ListNode *cur=l->head;
  if (cur==NULL) return 1;
  if (cur->prev!=NULL) return 0;
  for (; cur!=NULL && cur->next!=NULL;
      cur=cur->next) {
    if (!checkListNode(cur)) return 0;
  }
  return 1;
}
```

```
int checkListNode(void *args) {
  ListNode *cur=(ListNode *)args;
  if (cur->next==NULL) return 1;
  if (cur->next->prev!=cur) return 0;
  if (cur->val>cur->next->val) return 0;
  return 1;
}
```

Figure 5.2: Code with the modifications for incremental check.

pointed by $icStruct$ for checking. In the example, *insertIntoICheckStruct* is called after inserting a node in the list.

deleteFromICheckStruct: This function removes a data structure part from the dependency sets, new set, and affected set (that it belongs to) of $icStruct$, indicating that there is no need to check this part later. In the example, *deleteFromICheckStruct* is called after removing a node from the list.

handleOrder: This function notifies the library to handle the special order of the local checks for the data structure corresponding to the $icStruct$. It is only invoked in special cases.

genericICheck: This function performs the incremental check for the data structure corresponding to the *ICheckStruct* pointer $icStruct$. The return value of -1 means that there are no modifications since the last check and thus nothing needs to be checked, 1 means success, and 0 means failure.

```
/* globalFunc: function pointer for global check */
/* globalPara: the parameter for global check */
/* localFunc: function pointer for local check */
/* partSize: the size of a data structure part */
/* which is the basic unit for incremental check */
/* return: the ICheckStruct for incremental check */
ICheckStruct *newICheckStruct(Function globalFunc,
void *globalPara, Function localFunc, int partSize);

/* icStruct: pointer to the associated ICheckStruct */
/* part: the address of the new part */
void insertIntoICheckStruct(ICheckStruct *icStruct,
                            void *part);

/* icStruct: pointer to the associated ICheckStruct */
/* part: the address of the removed part */
void deleteFromICheckStruct(ICheckStruct *icStruct,
                            void *part);

/* icStruct: pointer to the associated ICheckStruct */
void handleOrder(ICheckStruct *icStruct);

/* icStruct: pointer to the associated ICheckStruct */
/* return code: -1 no check, 1 pass, 0 failure */
int genericICheck(ICheckStruct *icStruct);
```

Figure 5.3: Functions in iChecker interface.

This function should be called when the check is desired.

Besides the above modifications, the programmer may also need to provide a local checking function. In this example, the local checker is already a function called by the global checker, so the programmer does not need to write anything additional. In more advanced examples discussed in Section 5.4, the programmer needs to provide a local checker.

5.2.2 Incremental Check Demonstration

Assume that executing the *main* function in the example produces the list shown in Figure 5.1(a) right before inserting the value 6. The list after the insertion is shown in Figure 5.1(b). Performing the incremental check after the insertion requires that only the nodes $N2$ and $N4$ (shown in bold) are checked using the local checker $checkListNode$. In contrast, the global check would traverse all the nodes. Further execution removes the value 5 from the list. The list after the removal is shown in Figure 5.1(c). After the deletion, iChecker performs the local check only on the node $N1$ (shown in bold), whereas the global check would once again traverse all the nodes.

5.3 iChecker Framework

This section first gives an overview of iChecker framework followed by the detailed iChecker library implementation. It next explains how we address the three most important challenges for incremental consistency checks. It finally illustrates the framework using the sorted, doubly-linked list example.

5.3.1 Overview

Incremental consistency checking of mutable data structures should check only the parts of the data structure whose consistency may have changed due to modifications performed since the last consistency check. To achieve this, we propose a mechanism that builds a *dependency set* for each part (e.g., a node in a list, a subtree in a tree, or an element in an array) of the data structure. The dependency set for a part is the set of data on which this part's consistency depends. For example, as explained in Section 5.2, the dependency set of a node in the sorted doubly-linked list would include its value, its *next* field, its successor's value, and the successor's *prev* field.

The incremental check is performed on two types of parts. First, all new parts added since the last check are checked using the local checker; during this checking, iChecker automatically builds a dependency set for each new part. Second, each *affected part*, i.e., an existing part whose dependency set has a memory location that changed value since the last check, is re-checked using the local checker. During each local check, the dependency set for the corresponding part is also updated as some data may not be correlated to this part any more and some other may become this part's new "neighbors".

To implement the above incremental check process, we need to address three challenges: 1) obtain a local checker, 2) track modifications since the last check, and 3) find affected parts of the data structure that need to be checked based on the modifications. Our goal is to design and implement a framework that can address these challenges and, in the mean time, also achieve efficiency, simplicity, and flexibility. To achieve this goal, our iChecker framework leverages iWatcher [ZQL$^+$04] and provides a run-time iChecker library. Table 5.1 shows the main functionality of the hardware and library.

Component	Main Functionality
Hardware (iWatcher)	1. Track the monitored writes and reads 2. Trigger the monitoring functions
Library (iChecker)	1. build and update dependency sets 2. Track modifications that can affect consistency 3. Maintain an internal data structure for incremental check 4. Invoke the local checker for each new or affected part

Table 5.1: Main functionality of the iChecker components.

Our framework uses iWatcher to automatically track the writes and reads to the checked data structure (monitored accesses); iWatcher also automatically triggers the monitoring functions for monitored accesses. Our library has two monitoring functions. The read monitoring function builds/updates dependency sets using monitored reads, and the write function determines affected parts based on monitored writes (modifications) and dependency sets. For building/updating the dependency set of a part, the library informs iWatcher to track the monitored reads only during the local check of this part. The library also maintains an internal data structure, which includes the dependency sets. Finally, the library invokes the local checker for each new and affected part.

Using hardware, we can *efficiently* track writes to and reads from the checked data structure. Hardware passes information about writes to the library so it can track the modifications performed since the last check. Hardware passes information about reads to the library so that it can build and update dependency sets.

Between *simplicity* and *flexibility*, there is a trade-off: if a framework requires less modifications to the application, it is simpler for programmers to use but less flexible in supporting different structures and consistency properties. Our design choice is to find a balance. To achieve simplicity, our framework provides the iChecker library that exposes simple interface (Section 5.2.1) to the application and hides lots of implementation details, including the interaction with iWatcher. To achieve flexibility, we ask the programmer to provide the local checker.

5.3.2 Library Implementation

The iChecker library consists of one data structure, *ICheckStruct*, and four functions, *newICheckStruct*, *insertIntoICheckStruct*, *deleteFromICheckStruct*, and *genericICheck*. Recall that Figure 5.3 shows the interface for these functions.

The library data structure *ICheckStruct* maintains the information necessary for incremental check. Each instance of checked application data structure is associated with an instance of the library data structure *ICheckStruct*. In addition to the information described in Section 5.2, the *ICheckStruct* structure also contains a table that stores dependency sets for each checked part, a new set of parts that have never been checked, and an affected set of old parts whose dependency sets have some modified locations.

The implementation of *newICheckStruct* and *insertIntoICheckStruct* are straightforward as described in Section 5.2. The *genericICheck* function performs the incremental consistency check procedure. The application calls this function when a check is required. The function does the following: 1) calls *iWatcherOn* for each element in the new set to notify iWatcher to monitor these elements, 2) performs local check on each element in the new set and the affected set, and 3) informs iWatcher to track the monitored reads only inside the local check. At monitored reads, iWatcher automatically triggers the read monitoring function (provided by the library but not exposed to the user) to build/update dependency sets. The local checker returns 0 or 1 depending on the consistency result, and -1 if there is no element in either the new set or the affected set.

5.3.3 Obtaining an Incremental Checker

We assume that the application provides the global checker. To handle different types of structures and consistency properties, we also require the application to provide the local checker, which is usually already provided to implement the global checker. Without any high-level semantic information about the checked data structure, it would be very difficult to automatically generate the local checker only based on the global checker.

5.3.4 Efficient Tracking of Modifications

Our framework dynamically tracks modifications to the checked data structure using iWatcher hardware [ZQL+04]. An alternative approach for tracking modifications would be using software instrumentation to intercept most stores to check if the address is a monitored location. Due to aliasing problem, the instrumentation cannot precisely determine which stores could access a monitored location. This approach would induce larger overhead and contradict our efficiency goal.

Recall that during the *genericICheck*, each new part is monitored because the library calls *iWatcherOn* for a new part. iWatcher can automatically catch the writes to the monitored locations.

5.3.5 Finding the Affected Parts

Our framework uses *dependency sets* to determine the affected parts that need to be checked during the next incremental check. Each dependency set has the format $l : \{l_1, l_2, \ldots, l_n\}$, where l is a memory location of a checked data-structure part and $l_1, l_2, \ldots, l_n$ are memory locations inside the checked data-structure. The dependency set states that the part at memory location l depends on the memory locations $l_1, l_2, \ldots, l_n$. If any of the memory locations $l_1, l_2, \ldots, l_n$ is modified, a local check needs to be performed for the part l. With the help from iWatcher, the iChecker library builds and updates dependency sets: the library functions build and update dependency sets based on the monitored reads that iWatcher tracks during the local check. The intuition for this is the following: if the execution of local checker for the part at l reads only the locations $l_1, l_2, \ldots, l_n$, then the check result for l depends on $l_1, l_2, \ldots, l_n$.

Based on the dependency sets and the modifications, the library can compute which parts should be incrementally checked at the next consistency check. Since modifications potentially affect the consistency of these parts, we call these parts affected parts. The library adds them to the affected set of the corresponding *ICheckStruct*. The parts in the affected set are incrementally checked when *genericICheck* is called. For example, based on the above dependency set, if location l_2 is modified, iChecker adds l to the affected set so it will check l later.

5.3.6 Analysis on the Sorted Doubly-Linked List

We next provide a detailed analysis of incremental checking for the sorted doubly-linked list example in Section 5.2. Figure 5.1(a) shows the list l right before the execution of line A in Figure 5.2. Assume that iChecker has already built the dependency sets for all nodes in Figure 5.1(a) and instructed iWatcher to monitor all these nodes. The dependency sets are

$\&N1 : \{\&(N1.next), \&(N2.prev), \&N1, \&(N1.val), \&(N2.val)\}$ $\&N2 : \{\&(N2.next), \&(N3.prev), \&N2, \&$
...

We illustrate how iChecker tracks modifications and builds the dependency sets using the node $N4$ as an example part. Line A inserts the node $N4$, and the list becomes as shown in Figure 5.1(b). This insertion modifies the monitored locations $\&(N2.next)$ and $\&(N3.prev)$; iChecker detects this because iWatcher automatically tracks all accesses to monitored locations. Based on the modifications and the above dependency sets, iChecker adds $\&N2$ to the affected set.

At the end of the insertion function, *insertIntoICheckStruct* adds $\&N4$ to the new set. The $genericICheck$ after line A then performs the local checks on $\&N4$ (which is in the new set) and $\&N2$ (which is in the affected set). The execution of the check *checkListNode* (from Figure 5.2) on $\&N4$ reads locations $\&(N4.next)$, $\&(N4.next \rightarrow prev)$ ($\&(N3.prev)$), $\&N4$, $\&(N4.val)$, and $\&(N4.next \rightarrow val)$ ($\&(N3.val)$). Therefore, the library builds the dependency set for node $N4$

$$\&N4 : \{\&(N4.next), \&(N3.prev), \&N4, \&(N4.val), \&(N3.val)\}$$

Similarly, the library updates the dependency set for node $N2$

$$\&N2 : \{\&(N2.next), \&(N4.prev), \&N2, \&(N2.val), \&(N4.val)\}$$

After the incremental check, $genericICheck$ makes the new set and the affected set empty.

Line B removes the node $N2$, and the list becomes as shown in Figure 5.1(c). This deletion modifies $\&(N1.next)$, $\&(N4.prev)$, $\&(N2.next)$, and $\&(N2.prev)$, so iChecker adds $\&N1$ and $\&N2$ to the affected sets. However, $N2$ is removed from the list, and the dependency set for $N2$ is removed during the *deleteFromICheckStruct* call (at the end of the deletion). After line B, thus, $genericICheck$ only invokes the local check on $\&N1$ and updates its dependency set to

$$\&N1 : \{\&(N1.next), \&(N4.prev), \&N1, \&(N1.val), \&(N4.val)\}$$

Subject	Data structure type		Global checker type		Local checker type		Shadow	
	array-based	pointer-based	iterative	recursive	read-only	read-write	no	yes
list		X	X		X		X	
tree		X		X		X	X	
filesystem	X		X		X			X
Freeciv	X	X	X		X			X

Table 5.2: Subject classification. The data structures in Freeciv contain both array-based and pointer-based parts.

5.4 Case Studies

This section demonstrates the use of our iChecker framework in four case studies: a sorted doubly-linked list, a binary search tree, a simplified Linux file system, and an interactive game Freeciv. The first two studies are micro-benchmarks, and the last two were applications used in previous studies on detecting and repairing inconsistencies in data structures [DR03, DR05].

At the end, we also use a small sorted doubly-linked list and a small binary search tree to serve as negative examples for which using incremental checks imposes larger overhead than using global checks.

5.4.1 Subject Characteristics

We chose these four subjects for our case studies because they have very different data structure and consistency characteristics. Table 5.2 shows these characteristics. We categorize the subjects based on four aspects: data structure type, global checker type, local checker type, and check complexity (need shadow copy or not).

If the global checker is recursive, we add additional fields in the checked structure to record the output of recursive functions for incremental check (Section 5.4.4). Read-only means that the local checker only reads from the checked data structure, whereas read-write means that the local checker also writes to the checked data structure. For the latter, the order of local checks is critical for the correctness (Section 5.4.4). Shadow means that the checkers (both global and local) need to maintain a shadow copy of some parts of the checked data structure; a local checker with shadow structure needs the old value of a checked part for updating the shadow structure (Section 5.4.5).

5.4.2 Evaluation Methodology

Since our iChecker framework uses the iWatcher hardware, we implement the framework using the cycle-accurate execution-driven simulator that was also used in many previous studies [MRH+02, PT03, ZQL+04, ZLF+04]. The simulator simulates a 2.4 GHz machine with 2-level caches and 512 MB RAM. All cache misses are simulated as in real machines, thus the performance impact caused by all memory accesses is fully simulated. We implemented the iChecker library in C. All four subject programs are also in C.

For each case study, we will describe important consistency properties, provide global checkers and local checkers for these properties, and insert calls to perform consistency checks at the end of the function calls that modify the data structures.

Since our proposed incremental check mechanism can be also implemented without hardware support for catching monitored memory accesses, we also estimate the time overhead of potential software-only implementations. To catch accesses to the checked data structure using only software, the most straightforward way (called $SoftImp$) is to instrument all reads inside the local checker and all writes anywhere in the program. In contrast to iWatcher that looks up its *Check Table* (which maps monitored locations to their associated monitoring functions) only for accesses to monitored locations, a software-only implementation needs to look up its own table for every instrumented read or write instruction, often finding that the access is to a non-monitored location. Since the tables in two approaches would be similar, we use the average lookup time of iWatcher to approximate of a software-only implementation, specifically the time for those lookups that access a non-monitored location. Therefore, the estimated execution time of a software-only implementation is:

$$estimated\_time = iChecker\_time + avg\_lookup$$
$$* (\#instrumented\_accesses - \#monitored\_accesses)$$

We obtain all four parts of the formula from our simulations.

Although compilers can statically decide the addresses of some accesses (e.g. using pointer analysis) and eliminate some instrumentations in the software implementation, the fraction of reduction in instrumented memory accesses is usually not very large for programs like ours that use pointers to do fine-grain manipulation of data structures.

Another way to do local check in software is that the programmer invokes the local checker on the parts of the data structure that she knows may have been changed. For example, after inserting a node into the list, the programmer could check only the new node and its neighbors. Although this way does not require tracking of memory accesses, it cannot detect unintended modifications, e.g. through memory corruption. Therefore, we do not compare it with the global check and our incremental check.

Table 5.3 shows the time overheads (over the baseline, original program without checks) of global check, the time and space overheads (over the baseline) of incremental check using iChecker framework, as well as the estimated time overheads of SoftImp for software-only incremental check. The space overheads of global check are very small, and the space overheads of SoftImp should be similar to those of incremental check using iChecker. Table 5.4 shows the sizes of the original code as well as the modifications (the modifications for iChecker count the global checker) for all programs.

Application		global	iChecker		SoftImp
		time	time	space	time
Sorted doubly-linked list	n=2000	3.1X	2.8X	11.2X	18.1X
	n=4000	2.8X	1.3X	12.8X	6.5X
	n=8000	2.5X	0.5X	14.1X	2.6X
Binary search tree	n=2000	103.9X	35.8X	14.3X	257.0X
	n=4000	210.7X	41.6X	16.1X	253.1X
	n=8000	416.4X	50.1X	17.9X	249.2X
Filesystem		6.2X	0.04X	0.3X	0.1X
Freeciv		0.7X	0.1X	2.5X	0.2X

Table 5.3: Overheads of consistency checks for four case studies

Subject	Original code size (LOC)	Modification size (LOC)	
		global	iChecker
list	149	15	25
tree	172	18	36
filesystem	2871	37	93
Freeciv	88788	52	108

Table 5.4: Sizes of original code and modifications. The LOC refers to the original code and modifications for local and global checkers.

5.4.3 Sorted Doubly-Linked List

The sorted doubly-linked list is our running example from Section 5.2. The workload program first inserts n nodes with random values, then randomly removes 100 nodes. We perform the consistency check at the end of each insertion and deletion call.

Performance As shown in Table 5.3, iChecker can reduce the check time overhead by a factor of 1.1–5 for this program. The larger the n the higher the reduction because a larger data structure incurs a higher time overhead with global checks. iChecker's check time overhead is also 5-6 times smaller than the estimated overhead of a software-only implementation. The large space overheads in this benchmark are because the simple program only contains the list structure. As we can see, for more complicated programs like Filesystem and Freeciv, the space overheads are smaller.

5.4.4 Binary Search Tree

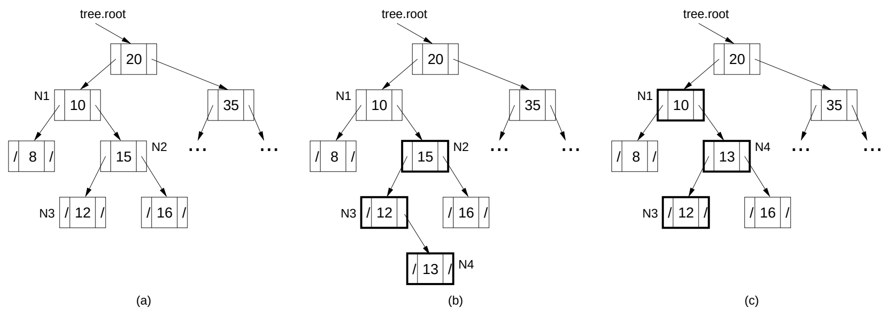


Figure 5.4: Binary search tree: (a) the original tree (b) the tree after insertion of N4; (c) the tree after removal of N2. The nodes in bold will be checked after the operation.

Example instance: Figure 5.5(a) shows instances of a binary search tree.

Consistency property: We require the tree to be ordered for binary search: for each node N, the maximum value in N's left subtree is not greater than the value in N, and the minimum value in N's right subtree is not less than the value in N. This property also implies that the nodes form a tree (not an arbitrary graph with cycles).

```
int isOk(void *args) {
  TreeNode *t=(TreeNode *)args;
  if (t==0) return 1;
  if (t->left !=0 && (!isOk(t->left) || max(t->left)>=t->val))
    return 0;
  if (t->right !=0 && (!isOk(t->right) || min(t->right)<=t->val))
    return 0;
  return 1;
}

int max(TreeNode * t) {
  if (t->right!=0)
    return max(t->right);
  return t->val;
}

int min(TreeNode * t) {
  if (t->left!=0)
    return min(t->left);
  return t->val;
}
```

(a)

```
int isOkICheck(void *args) {
  TreeNode *t=(TreeNode *)args;
  if (t==0) return 1;
  if (t->left !=0 && (!t->left->isOk) || t->left->max>=t->val))
    return 0;
  if (t->right !=0 && (!t->right->isOk || t->right->min<=t->val))
    return 0;

  t->max=maxI(t);
  t->min=minI(t);
  t->isOk=1;
  return 1;
}

int maxI(TreeNode * t) {
  if (t->right!=0)
    return t->right->max;
  return t->val;
}

int minI(TreeNode * t) {
  if (t->left!=0)
    return t->left->min;
  return t->val;
}
```

(b)

Figure 5.5: The checkers for binary search tree: (a) global checker and (b) local checker.

Global checker: Figure 5.5(a) shows the global checker. It is a recursive function and also uses the recursive *max* and *min* helper functions. The parameter for the global checker is the root of the tree. Intuitively, the local checker should perform the similar check but only on a subtree that was modified. However, if we simply use the global checker as the local checker (by just passing the root of a subtree as the parameter for local check), the dependency set for any tree node would include all its descendants because the recursive calls eventually access all its descendants. In particular, the dependency set for the root would include all tree nodes. Hence, if anything in the tree changes, the root would need to be incrementally checked, which is the same as the global check.

Local checker: The above problem can be solved by following this simple three-step guideline: 1) for each recursive function called in the global checker, add a corresponding field in the structure part; 2) update these fields in the local checker; and 3) replace function calls with field reads in recursive helper functions. In particular, for the tree, it means adding fields *isOk*, *max*, and *min* to the $TreeNode$ structure. Figure 5.5(b) shows the local checker that updates the fields in *isOkICheck* and reads them in the *maxI* and *minI* functions.

Check propagation: The local checker updates the max and min fields, which can trigger *check propagation*: checking a node N can trigger checking of the N's parent (due to the possible changes in its max or min values), which can then trigger checking of the N's grandparent, and so on until the root node. For example, Figure 5.4(b) shows the tree after inserting $N4$ to the tree. Suppose that the consistency has been checked right before the insertion. Then checking consistency right after the insertion requires that $N4$ be checked because it is a new node. Since $N3$ depends on its right child (that has changed from *NULL* to $\&N4$), $N3$ needs be checked, too. The check on $N3$ changes its max from 12 to 13, and thus its parent $N2$ also needs be checked (because $N2$ depends on $N3.max$). In this example, check propagation stops at $N2$. In general, it can proceed to the root, which means that the incremental check may need to check (only) one entire path in the tree, whereas the global check always needs to traverse the entire tree.

Check order: The writes in the local checker also raise the issue of the order of the local checks. If some node and its ancestor both need to be checked, ideally the checking should follow the bottom-up order, i.e., check the descendants before checking the ancestors, because checking ancestors needs max and min values that may be updated due to the check propagation. However, the current implementation of our framework simply follows the order of modifications for check order. If not used carefully, this order may produce wrong check results. For example, in the removal case of Figure 5.4(c), the modification order is $N1$, $N4$, $N3$, but the correct check order should be $N3$, $N4$, $N1$.

To guarantee the correct check result for the local checker that is read-write, the library does the following. After checking one data-structure part (one node for tree), the library records the check result. If some later checks propagate to this part (indicating that the order was suboptimal), the library checks the part again. The process repeats until fixed point, when there is no order violation. This process terminates even for cyclic dependencies when local checkers are correct. The application notifies the library to handle the order by calling the library function *handleOrder(icStruct)*. Note that the check propagation and check order issues do not arise in the list example, because the local checker only reads from the data structure, which is indeed the common case for checks.

Other code modifications: Similarly to the list example, the program for tree also needs to call *newICheckStruct* for initialization, *insertIntoICheckStruct* when inserting a new tree node, *delete-FromICheckStruct* when removing a node from the tree, and *genericICheck* for a consistency check.

Performance The workload program first inserts n nodes with random values, then randomly removes 100 nodes. We perform the consistency check at the end of each insertion and deletion. Table 5.3 shows that iChecker reduces the check time overhead by 2.9–8 times compared to global checks and 5–7 times compared to the software-only estimation. The large space overheads are due to the same reason as that for the linked list.

5.4.5 A Simplified Linux File System

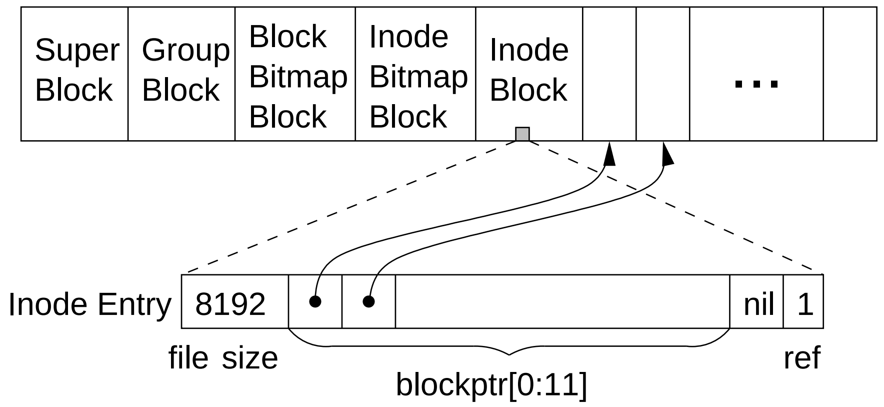


Figure 5.6: A simplified Linux file system application.

This subject program was implemented and used by Demsky et al. [DR03, DR05] in previous studies on detecting and repairing data-structure inconsistencies. It is a simplified version of the

Linux ext2 file system.

Example instance: As shown in Figure 5.6, the file system consists of an array of blocks. The first five blocks are used to keep the file system meta information. The $Inode$ block contains all inode entries, the $InodeBitmap$ block identifies the used and free inode entries, and the $BlockBitmap$ block identifies the used and free blocks. Each inode entry has a ref field. If an inode entry is used, it should have $ref > 0$; otherwise, it should have $ref = 0$. Each inode entry also has 12 block pointers $blockptr[0..11]$, and each pointer either points to a block used by this inode, or equals to NULL. Both inode bitmap and block bitmap are an array of chars.

Consistency properties: We check two properties, *inodeConsistency* that requires the status (used or free) of all inode entries to match the corresponding inode bitmap bits (1 or 0) and *blockConsistency* that requires the status of all blocks to match the corresponding block bitmap bits.

```
int checkInode(void *args) {
  /* ptr points to the file system block array */
  /* itb points to the inode block */
  struct block *ptr=(struct block *)args;
  struct InodeBlock *itb=(struct InodeBlock *) &ptr[itbptr];

  /* ib.inode is the inode bitmap */
  /* itb->entries[i] is the ith inode entry */
  for (int i=0; i<NUMINODES; i++) {
    if ((!(ib.inode[i/8]&(1<<(i%8))) && itb->entries[i].ref>0) ||
        ((ib.inode[i/8]&(1<<(i%8))) && itb->entries[i].ref<=0))
      return 0;
  }
  return 1;
}
```

(a)

```
int checkInodeICheck(void *args) {
  struct InodeBlock *itb=(struct InodeBlock *) &ptr[itbptr];
  /* iboffset points to the checked char in the inode bitmap */
  /* startEntry is 1st in the 8 entries corresponding to the */
  /* checked char */
  char *iboffset = (char *)args;
  int startEntry = (iboffset-(char *)(&ib))*8;

  for (int i=0; i<8; i++) {
    if ((!((*iboffset)&(1<<i)) && itb->entries[startEntry+i].ref>0)
        || (((*iboffset)&(1<<i)) & itb->entries[startEntry+i].ref<=0))
      return 0;
  }
  return 1;
}
```

(b)

Figure 5.7: The checkers for inodeConsistency in the simplified Linux file system: (a) global checker and (b) local checker.

Check $inodeConsistency$ *Global checker*: Figure 5.7(a) shows the global checker. Since the status of an inode entry is determined by its ref, the global checker checks if the ref in each inode entry matches the corresponding bit in the inode bitmap.

Local checker: A basic data-structure part for local checker is a char in the inode bitmap array. A char in the inode bitmap corresponds to 8 consecutive inode entries. Figure 5.7(b) shows the local checker. It only checks if the bits in the given char match the ref fields in the corresponding 8 inode entries.

Other modifications: As usual, we need to initialize $ICheckStruct$ with *newICheckStruct*. After the initialization, we also need to insert all chars from the inode bitmap array into the new set by using *insertIntoICheckStruct*, because all these array elements should be checked during the next/first incremental check. As for dynamic structures (e.g., list or tree), we call *genericICheck* for consistency checking.

We also need to monitor the ref fields of all inode entries. However, the ref field is not a part of incremental check, so we cannot use $insertIntoICheckStruct$ call to notify the library to call $iWatcherOn$ on these fields. Therefore, at the beginning, we need to explicitly call $iWatcherOn$ for all ref fields.

```
int checkBlock() {
  for (int i=0; i<NUMBLOCK/8; i++)
    if (bb.blocks[i]!=shadowbb.blocks[i])
      return 0;
  for (i=0; i<NUMBLOCK%8; i++)
    if ((bb.blocks[NUMBLOCK/8]&(1<<(i%8)))!=
(shadowbb.blocks[NUMBLOCK/8]&(1<<(i%8))))
      return 0;
  return 1;
}

int genBlockBitmap(void *args) {
  struct block *ptr=(struct block *)args;
  struct InodeBlock *itb=(struct InodeBlock *) &ptr[itbptr];

  /* shadowbb.blocks is the shadow block bitmap */
  /* bb.blocks is the file system block bitmap */
  /* itb->entries[i].Blockptr[j] is the jth block pointer of the ith inode */
  for (int i=0; i<NUMINODES; i++) {
    for (int j=0; j<12; j++) {
      if (itb->entries[i].Blockptr[j]!=NULL) {
        int k=itb->entries[i].Blockptr[j];
        if (shadowbb.blocks[k/8]&(1<<(k%8)))
          return 0;
        shadowbb.blocks[k/8]=shadowbb.blocks[k/8]|(1<<(k%8));
      }
    }
  }
  return 1;
}
```

(a)

```
int genBlockBitmapICheck(void *args) {
  int *bptr = (int *)args;
  struct InodeBlock *itb=(struct InodeBlock *) &ptr[itbptr];
  int index = 12*(bptr-(int *)(itb)/sizeof(Inode)
              +(bptr-(int *)(itb))%sizeof(Inode)-1;

  /* old array stores old values for all block pointers */
  /* old[index] keeps old value for the checked block pointer bptr */
  if (old[index]!=0) {
    // reset old bit to 0
    if (!(shadowbb.blocks[old[index]/8]&(1<<(old[index]%8))))
      return 0;
    shadowbb.blocks[old[index]]&=0xff^(1<<(old[index]%8));
  }

  if (*bptr!=0) {
    if (shadowbb.blocks[(*bptr)/8]&(1<<((*bptr)%8)))
      return 0;
    shadowbb.blocks[(*bptr)/8]|=1<<((*bptr)%8);

    old[index]=*bptr;
  }
  return 1;
}
```

(b)

Figure 5.8: The checkers for blockConsistency in the simplified Linux file system: (a) global checker and (b) local checker.

Check $blockConsistency$ *Global checker*: The consistency check for block is not as straight-forward as the check for inode because the status of a block is determined not by a field of the block structure, but by block pointers of other data: inode entries. One straightforward way to implement the global check function is: for every bit in the block bitmap, search every inode to see if any *blockptr* in the inode points to the corresponding block. If the results do not match (e.g. the bit in the bitmap is set to 1 but there is no inode's *blockptr* point to the corresponding block, or vice versa), the data structure is inconsistent. While this implementation is simple, it is very inefficient because for every bit, it needs to search every inode. To improve checking efficiency, an incorrect solution is to simply start from inode entries and check, for each inode entry, to see if blocks pointed by its *blockptr* field has the corresponding bit in the block bitmap set to 1. Such checking procedure is insufficient because it does not check the consistency properties for those bits whose blocks are not pointed by the *blockptr* field of any inode.

To optimize performance, a correct and efficient alternative is to use a *shadow* block bitmap ($shadowbb.blocks$) based on the block pointers of all inode entries by only traversing all the inodes only once. Then the checker just needs to compare this shadow bitmap with the block bitmap of the file system ($bb.blocks$). In our evaluation, we use this implementation for global checks, as shown in Figure 5.8(a).

Local checker: We provide the local checker for $genBlockBitmap$ instead of for $checkBlock$. A basic part for generating the shadow bitmap incrementally is a block pointer. When a block pointer changes, the local checker needs to know not only the new value, but also the old value, because the checker needs to reset the corresponding bitmap bit for the old value to 0. Therefore, the checker maintains an array for all old block pointer values. Figure 5.8(b) shows the local checker for $genBlockBitmap$. The $checkBlock$ remains the same as in the global checker.

Other modifications: As for $inodeConsistency$, we need to call *newICheckStruct* and *insertInto-ICheckStruct* to add all block pointers to the new set and to call *genericICheck* for a consistency check. We also need to explicitly call $iWatcherOn$ on the entire block bitmap at the beginning.

Performance Our workload program performs the following sequence of file manipulations: first open 145 files (it will create them since they did not exist), next write to each file and repeat this write process 6000 times, then close these files, and finally open them again, read from them, and close them. We perform the $inodeConsistency$ check at the end of *openfile* function because it may change the inode entries and the inode bitmap, and the $blockConsistency$ check at the end of *writefile* function since it may change the block pointers and the block bitmap.

Table 5.3 shows that iChecker's time overhead is 155 times smaller than the global check! It is also twice smaller than the software-only estimation. The space overhead is only 0.3X.

5.4.6 Freeciv

Freeciv is a publicly available, interactive, multi-player game. It was also used in previous studies on detecting and repairing data-structure inconsistencies [DR03, DR05]. Freeciv is a strategy game that simulates development of civilizations. The program uses several data structures, but the most interesting is the one that represents a map of the world. The map has a grid of tiles. Each tile has a city pointer that either points to a city that is on the tile or is NULL if there is no city on the tile.

Consistency properties: We check the following two properties related to tiles and cities: 1) *cityononlyonetile* that requires each city to be on only one tile, and 2) *citymustononetile* that requires each city to be on one tile.

Global checker and local checker: Both the global checkers and local checkers are similar to those in the file system example. All checkers for both consistency properties keep a shadow data structure. In addition, the local checkers for both properties use an array that maintains the old city on a tile for updating the shadow structure. Due to space constraints, we do not present all details.

Performance Our workload program consists of the computer playing several turns of the strategy game against itself. We identify some important functions in Freeciv and check the consistencies at the end of these functions. As shown in table 5.3, the time and space overheads of incremental check are small.

5.4.7 Negative Examples

While incremental checking can significantly reduce the checking overhead over global checks in most programs, its performance gain depends on the size of the structure, the check frequence, and the complexity of properties. Here, we will only demonstrate for small structures the overhead could offset the performance gain, by scaling down our sorted doubly-linked list and binary search tree. As shown in Table 5.5, incremental checks (both iChecker and the software-only estimation) have larger overheads than global checks due to the overhead for tracking updates and maintaining dependency sets. Therefore, in these cases, it is not a good idea to use incremental checks.

However, this does not affect the overall benefit of incremental checks because, after all, bugs that occur in larger data structures are usually much harder to diagnose than those in smaller ones and thereby have stronger demands for efficient ways to check data consistency as frequent as possible. Efficient incremental checking methods such as iChecker exactly serve this purpose.

data structures	n	global	iChecker	SoftImp
sorted doubly-linked list	100	1.67X	15.33X	145X
	1000	3.01X	3.78X	33.53X
binary search tree	100	3.25X	29X	240X
	1000	50.24X	33.11X	257.74X

Table 5.5: Overheads with small data structures (n is the number of nodes inserted in the data structure).

5.5 Summary

We have presented iChecker, a framework for incremental consistency checking of mutable data structures in C programs. Our framework leverages a previously proposed iWatcher hardware to provide a library for efficient, incremental, run-time consistency checking. The main idea is to perform a consistency check using only a local check on the small parts of data structure modified since the last consistency check. To use iChecker, the programmer needs to provide a local check function, and iChecker automatically monitors memory accesses to build dependencies and to invoke the local check function only as necessary. Our experimental results show that for large data structures, iChecker provides a significant speedup (23.3 times on average) over global checks that traverse the entire data structure regardless of modifications.

iChecker requires the programmer to provide the local checker and guarantee the soundness of the local checker. We expect the programmer to be willing to do that to achieve the orders of magnitude smaller overhead (like 155 times less overhead in the file system case).

We believe that iChecker has the potential to change the view that developers have on seemingly high-overhead consistency checking (and assertions in general): as the overhead gets lower, we expect developers to use checking more aggressively in various development tasks such as testing, debugging, and program understanding.

Chapter 6

Conclusions and Future Work

This dissertation work provides architectural support for software debugging. More specifically, it focuses on addressing the three limitations of dynamic monitoring for detecting memory-related bugs: inefficiency, inaccuracy and limited bug coverage. The contributions of this work are it proposes a simple and general architectural framework called iWatcher for low overhead location-controlled dynamic monitoring, a new bug detection method, PC-based invariants, to catch hard-to-find program-specific memory bugs, and a simple architectural extension to iWatcher to further reduce the overhead of PC-based invariant debugging. It builds an automatic detection tool called AccMon that uses PC invariants and leverages iWatcher with the CLB extension, and a pure software detection tool called AccMon-S for PC invariants detection using PIN tool. It also reduces the overhead of data structure consistency check in C programs by proposing an incremental check framework called iChecker that leverages iWatcher for efficient consistency check of mutable data structures.

Particularly, iWatcher automatically detects all accesses to a watched memory location, including those by aliased pointer dereferences. To further reduce overhead and support rollback, iWatcher can optionally leverage Thread-Level Speculation (TLS). The experimental results with seven buggy applications (with various bugs) show that iWatcher detects all the bugs evaluated in our experiments with only a 0.1-179% execution overhead. In contrast, a well-known open-source bug detector called Valgrind induces orders of magnitude more overhead, and can only detect a subset of the bugs.

To catch the hard-to-find bugs with low overhead, AccMon uses a statistics-based method, PC-based invariants, and leverages architectural, run-time system and compiler support. It detects all ten tested bugs in non-server programs 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 uses 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 [HJ92] and CCured [NMW02, CHM$^+$03].

The software implementation of PC-based invariants, AccMon-S, is built by using the binary instrumentation tool PIN. Although it does not need extra hardware, it introduces 10.4-57.8 times execution overheads, orders of magnitude larger than AccMon's overheads. Besides detecting all ten bugs tested in AccMon, AccMon-S also detected two real bugs in two large real-word server applications, Apache and Squid with few false alarms (0-4).

The idea for reducing the consistency check overhead is to perform a consistency check using only a local check on the small parts of data structure modified since the last consistency check. To use iChecker, the programmer only needs to provide a local check function, and iChecker automatically monitors memory accesses to build dependencies and to invoke the local check function only as necessary. Our experimental results show that iChecker provides a significant speedup (up to 155 times) over global checks that traverse the entire data structure regardless of modifications.

In the future, I would like to explore hardware support and novel ideas for detecting other types of bugs, apart from memory-related bugs that are the main focus of my thesis research. One of my main targets is concurrency bugs, including data races and atomicity bugs that account for a large portion of all bugs, especially in server software. The urgency of handling such bugs grows as the emerging multicore architectures will lead to more multi-threaded applications. However, it is notoriously hard to expose, reproduce, and catch such bugs due to the nondeterminism. It would be interesting to investigate hardware support for efficiently detecting such bugs in production runs, where they are more likely exposed than in in-house testing.

Another category is semantic bugs which are usually program specific and, thus, hard to detect automatically. The main question is how to effectively collect such program-specific information.

Answering this question requires a deep understanding of semantic bugs and program behavior, for example, by help from machine learning and statistics techniques.

I also plan to apply the incremental check idea to specific domains, such as file systems. It would be very useful if we could significantly reduce the consistency check time for real-world file systems.

References

[ABH02] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *POPL*, 2002.

[ABH03] Umut A. A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. *SIGPLAN Not.*, 38(1):14–25, 2003.

[ABS94] Todd M. Austin, Scott E. Breach, and Gurindar 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.

[BH00] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BW96] Paul E. Black and Phillip J. Windley. Inference rules for programming languages with side effects in expressions. In *TPHOLs '96: Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, 1996.

[BZ93] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 187–196, 1993.

[CCJA98] 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.

[CFPT94] Stefano Ceri, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. Automatic generation of production rules for integrity maintenance. *ACM Trans. Database Syst.*, 19(3):367–422, 1994.

[CHM$^+$03] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 232–244, January 2003.

[CLL+02] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 258–269, June 2002.

[CMT00] M. Cintra, J. Martinez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory systems. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, pages 13–24, June 2000.

[CmWH99] Daniel A. Connors and Wen mei W. Hwu. Compiler-directed dynamic computation reuse: rationale and initial results. In *MICRO*, 1999.

[CPM+98] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather 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.

[CYC+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An empirical study of operating system errors. In *Symposium on Operating Systems Principles*, pages 73–88, 2001.

[dCFF00] Amarildo T. da Costa, Felipe M. G. Franca, and Eliseu M. C. Filho. The dynamic trace memorization reuse technique. In *PACT*, 2000.

[DCRR04] Brian Demsky, Cristian Cadar, Daniel Roy, and Martin C. Rinard. Efficient specification-assisted error localization. In *Second International Workshop on Dynamic Analysis*, 2004.

[DHW+97] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Z. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 292–302, December 1997.

[DR03] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA*, 2003.

[DR05] Brian Demsky and Martin Rinard. Data structure repair using goal-directed reasoning. In *ICSE*, 2005.

[DRT81] Alan Demers, Thomas Reps, and Tim Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *POPL*, 1981.

[EA03] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 237–252, October 2003.

[ECC01] Dawson Engler, David Yu Chen, and Andy 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.

[ECGN99] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, 1999.

[ECGN00] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pages 449–458, June 2000.

[Fab74] R. S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[FLL$^+$02] 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.

[FS01] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*, pages 55–66, August 2001.

[GJKW97] Neeraj K. Gupta, Lalita Jategaonkar Jagadeesan, Eleftherios E. Koutsofios, and David M. Weiss. Auditdraw: Generating audits the fast way. In *RE '97: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97)*, 1997.

[HCXE02] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 69–82, June 2002.

[HJ92] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the 1999 USENIX Winter Technical Conference*, pages 125–136, December 1992.

[HL02] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pages 291–301, May 2002.

[HL03] Jian Huang and David J. Lilja. Balancing reuse opportunities and performance gains with subblock value reuse. *IEEE Trans. Comput.*, 52(8):1032–1050, 2003.

[HLY00] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. *ACM SIGPLAN Notices*, 35(5):311–320, 2000.

[HMC94] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. In *Scalable High Performance Computing Conference*, 1994.

[Int01] Intel Corporation. The IA-32 Intel architecture software developer's manual, volume 2: Instruction set reference. Intel, 2001.

[Int04] Intel Corporation. The IA-32 Intel architecture software developer's manual, volume 3: System programming guild, 2004.

[JK97] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging*, 1997.

[Joh82] M. S. Johnson. Some requirements for architectural support of software debugging. In *Proceedings of the 1st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 140–148, March 1982.

[KAI] KAI-Intel Corporation. Intel thread checker. URL: http://developer.intel.com/software/products/threading/tcwin.

[KCE92] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architectural support for single address space operating systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–186, October 1992.

[KH92] G. Kane and J. Heinrich. *MIPS RISC architecture*. Prentice-Hall, 1992.

[L^+00] David Lie et al. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 168–177, November 2000.

[LAZJ03] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[LCM^+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[Lev84] Herry M. Levy. *Capability-based computer systems*. Digital Press, Bedford, MA, 1984.

[LJE03] Sang-Ik Lee, Troy A. Johnson, and Rudolf 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.

[LLMZ04] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan 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.

[LS95] J. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.

[LT95] Yanhong A. Liu and Tim Teitelbaum. Systematic derivation of incremental programs. *Sci. Comput. Program.*, 24(1):1–39, 1995.

[LW94] Alvin R. Lebeck and David A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, 1994.

[LYHR01] Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas W. Reps. Debugging via run-time type checking. In *the 4th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 217–232, April 2001.

[MA87] Samiha Mourad and Dorothy Andrews. On the reliability of the ibm mvs/xa operating system. *IEEE Trans. Softw. Eng.*, 13(10):1135–1139, 1987.

[MFH95] J. Mayfield, T. Finin, and M. Hall. Using automatic memoization as a software engineering tool in real-world ai systems. In *11th Conference on Artificial Intelligence for Applications*, 1995.

[MMFC01] Andreas Moshovos, Gokhan Memik, Babak Falsafi, and Alok 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.

[MPC$^+$02] Madanlal Musuvathi, David Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (SOSP)*, pages 75–88, December 2002.

[MRH$^+$02] José F. Martínez, Jose Renau, Michael C. Huang, Milos Prvulovic, and Josep Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *International Symposium on Microarchitecture (MICRO)*, 2002.

[MS00] E. Marcus and H. Stern. *Blueprints for high availability*. John Willey and Sons, New York, 2000.

[MS01] Anders Moller and Michael I. Schwartzbach. The pointer assertion logic engine. In *PLDI*, 2001.

[Nat02] National Institute of Standards and Technlogy (NIST), Department of Commerce. Software errors cost U.S. economy $59.5 billion annually. NIST News Release 2002-10, June 2002.

[NMW02] George C. Necula, Scott McPeak, and Westley 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.

[NPC05] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA05*, 2005.

[NS03] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Proceedings of the 3rd International Workshop on Runtime Verification (RV)*, July 2003.

[OL02] Jeffrey Oplinger and Monica S. Lam. Enhancing software reliability with speculative threads. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 184–196, October 2002.

[One96] A. One. Smashing the stack for fun and profit. Phrack Magazine, November 1996.

[Pau94] L. C. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828, 1994.

[PF95] H. Patil and C. N. Fischer. Efficient run-time monitoring using shadow processing. In *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG)*, pages 119–132, May 1995.

[PF97] Harish Patil and Charles Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software - Practice and Experience*, 27(1):87–110, January 1997.

[PLL02] Jih-Kwon Peir, Shih-Chang Lai, and Shih-Lien Lu. Bloom filtering cache misses for accurate data speculation and prefetching. In *Proceedings of the 16th Annual ACM International Conference on Supercomputing (ICS)*, 2002.

[PT03] 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)*, pages 110–121, June 2003.

[Pug88] William Pugh. An improved replacement strategy for function caching. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, 1988.

[RR93] G. Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *POPL*, 1993.

[S$^+$94] Ioannis Schoinas et al. Fine-grain access control for distributed shared memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 297–306, October 1994.

[SBN$^+$97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[SBV95] G.S. Sohi, S.E. Breach, and T.N. Vijayakumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 414–425, June 1995.

[SCG$^+$03] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS)*, pages 160–171, June 2003.

[SCZM00] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, June 2000.

[SD95] Ulrich Stern and David L. Dill. Automatic verification of the SCI cache coherence protocol. In *Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 21–34, October 1995.

[SDB$^+$03] Simha Sethumadhavan, Rajagopalan Desikan, Doug Burger, Charles R. Moore, and Stephen W. Keckler. Scalabel hardware memory disambiguation for high ILP processors. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO-36)*, Dec 2003.

[SE94] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205, June 1994.

[SL96] Eric Schnarr and James R. Larus. Instruction scheduling and executable editing. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 288–297, December 1996.

[SPA92] SPARC International. *The SPARC architecture manual: Version 8*. Prentice-Hall, 1992.

[Spr02] Brinkley Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, July-August 2002.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[SS97] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *ISCA*, pages 194–205, 1997.

[TCV04] Nathan Tuck, Brad Calder, and George Varghese. Hardware and binary modification support for code pointer protection from buf fer overflow. In *MICRO37*, 2004.

[THA$^+$99] Jenn-Yaun Tsai, Jian Huang, Christoffer Amlo, David J. Lilja, and Pen-Chung Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, September 1999.

[UD90] S. D. Urban and L. M. Delcambre. Constraint analysis: A design process for specifying operations on objects. *IEEE Trans. on Knowledge and Data Engineering*, 2(4):391–400, 1990.

[Wah92] R. Wahbe. Efficient data breakpoints. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 200–212, October 1992.

[WCA02] E. Witchel, J. Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 304–316, October 2002.

[WLG93] Robert Wahbe, Steven Lucco, and Susan L. Graham. Practical data breakpoints: Design and implementation. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, June 1993.

[XBH03] Min Xu, Rastislav Bodík, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30st Annual International Symposium on Computer Architecture (ISCA)*, pages 122–133, June 2003.

[XKPI02] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K. Iyer. Architecture support for defending against buffer overflow attacks. 2nd Workshop on Evaluating and Architecting System Dependability (EASY), October 2002.

[YS88] D. Yellin and R. Strom. Inc: a language for incremental computations. In *PLDI*, 1988.

[ZL98] Yuchen Zhang and Yanhong A. Lin. Automating derivation of incremental programs. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, 1998.

[ZLF$^+$04] Pin Zhou, Wei Liu, Long Fei, Shan Lu, Feng Qin, Yuanyuan Zhou, Samuel Midkiff, and Josep Torrellas. AccMon: Automatically Detecting Memory-related Bugs via Program Counter-based Invariants. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO)*, 2004.

[ZLKR04] Karen Zee, Patrick Lam, Viktor Kuncak, and Martin Rinard. Combining theorem proving with static analysis for data structure consistency. In *2nd Workshop on Software Verification and Validation*, 2004.

[ZQL$^+$04] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 224–235, June 2004.

[ZZQ$^+$05] Yuanyuan Zhou, Pin Zhou, Feng Qin, Wei Liu, and Josep Torrellas. Efficient and Flexible Architectural Support for Dynamic Monitoring. *ACM Transactions on Architecture and Code Optimization*, 2(1):3–33, 2005.

Author’s Biography

Pin Zhou completes her Ph.D degree in October 2006 from the Department of Computer Science at the University of Illinois at Urbana-Champaign. She holds an M.E. from Tsinghua University, China. Her research interests are hardware and system support for improving software dependability, operating systems, memory management, and storage systems. The work presented in this dissertation appeared at ISCA, MICRO, IEEE Micro’s Top Picks, and ACM TACO. She received the W.J. Poppelbaum Memorial Award in 2004, an honor given to one or two graduate students every year in computer architecture at University of Illinois. Two of her papers were selected in the IEEE Micro Special Issue: Micro’s Top Picks from Computer Architecture conferences in 2004.