Hardware Mechanisms for Distributed Dynamic Software Analysis

by

Joseph Lee Greathouse

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2012

Doctoral Committee:
Professor Todd M. Austin, Chair
Professor Scott Mahlke
Associate Professor Robert Dick
Associate Professor Jason Nelson Flinn

To my parents, Gail and Russell Greathouse.
Without their support throughout my life, I would never have made it this far.

Acknowledgments

First and foremost, I must thank my advisor, Professor Todd Austin, for his help and guidance throughout my graduate career. I started graduate school with the broad desire to "research computer architecture," but under Professor Austin's watch, I have been able to hone this into work that interests us both and has real applications. His spot-on advice about choosing topics, performing research, writing papers, and giving talks has been an invaluable introduction to the research world.

The members of my committee, Professors Robert Dick, Jason Flinn, and Scott Mahlke, also deserve special gratitude. Their insights, comments, and suggestions have immeasurably improved this dissertation. Together their expertise spans low-level hardware to systems software and beyond. This meant that I needed to ensure that any of my ideas were defensible from all sides.

I have been fortunate enough to collaborate with numerous co-authors. I have often published with Professor Valeria Bertacco, who, much like Professor Austin, has given me invaluable advice during my time at Michigan. I am extremely lucky to have had the chance to work closely with Ilya Wagner, David Ramos, and Andrea Pellegrini, all of whom have continued to be good friends even after the high-stress publication process. My other Michigan co-authors are Professor Seth Pettie, Gautam Bhatnagar, Chelsea LeBlanc, Yixin Luo, and Hongyi Xin, each of whom has significantly contributed to my graduate career and whose help I greatly appreciate.

I was also fortunate to have spent five months doing research at Intel. This opportunity broadened my research into concurrency bugs and helped break me out of my post-quals slump. Matt Frank, Zhiqiang Ma, and Ramesh Peri were co-authors on the paper that resulted from this work, and I heartily thank them for all of their help. I also wish to thank the entire Inspector XE team, whose product was amazing to work with. Matt Braun, especially, was always available to help keep my project moving forward.

Matt Frank also aided me immensely when I was an undergraduate at the University of Illinois. He and Professors Sanjay Patel and Donna Brown were instrumental in convincing me to go to graduate school in the first place. I am certain that their recommendation letters

were the main reason I was admitted into Michigan. I am also thankful to Nick Carter, who was a faculty member at UIUC at the time, for allowing me to do research with his group during my senior year. I failed completely at the tasks he assigned me, but it was an incomparable learning experience.

I doubt I would have been able to complete this dissertation without the support of the other graduate students in my lab. I'm extremely lucky to have had such amazing people in close proximity, which has really proven to me the power of a good group of motivated people. Thanks to all of you for being great: Bill Arthur, Adam Bauserman, Kai-hui Chang, Debapriya Chatterjee, Jason Clemons, Kypros Constantinides, Shamik Ganguly, Jeff Hao, Andrew Jones, Rawan Khalek, Shashank Mahajan, Biruk Mammo, Jinsub Moon, Andreas Moustakas, Ritesh Parikh, Robert Perricone, Steve Plaza, Shobana Sudhakar, and Dan Zhang.

There are also loads of people in the department that have been an inspiration. Michigan's CSE program is strong because of people like you: Professor Tom Wenisch, Timur Alperovich, Bashar Al-Rawi, Héctor Garcia, Anthony Gutierrez, Jin Hu, Anoushe Jamshidi, Daya Khudia, Dave Meisner, Jarrod Roy, Korey Sewell, and Ken Zick. I think it's also important to acknowledge everyone who has volunteered to help out with the ACAL reading group.

Andrew DeOrio deserves special thanks for putting up with me as a labmate for five years and as a roommate for three. We started at nearly the same time and are finishing at nearly the same time, even if our day-to-day schedules almost never overlap.

My friends in Illinois were also a big help in making it through this process. Paul Mathewson, especially, kept me focused on things besides engineering (and let me crash on his couch during my internship). Dan Bassett and Dan Baker both kept me grounded in reality in their own ways.

Finally, my family is the biggest reason that I've been able to do this work. Without their constant support throughout childhood, college, and graduate school, I would've stopped a hundred times along the way. My parents and grandparents have always encouraged me in any decision I've made. I couldn't ask for anything more.

Table of Contents

List of Tables

Table

List of Figures

Figure

Abstract

The complexity of modern software makes it difficult to ship correct programs. Errors can cost money and lives, so developers often use automated software analysis tools to hunt for bugs. By automatically analyzing a program's runtime operation, dynamic analysis tools can find subtle errors that would normally escape the notice of even careful human testers.

These dynamic analyses are more likely to find errors if they observe the program under numerous and varied runtime situations, so, ideally, users would analyze programs as they run them. Regrettably, these tests cause orders-of-magnitude slowdowns, which few users would tolerate.

This dissertation presents methods of accelerating dynamic software analyses in order to distribute these trials to users. It begins by observing that the overhead that any individual observes can be lowered by allowing each user to analyze a small, random sample of the program's operations. It then describes a sampling mechanism for dynamic dataflow analyses that only requires access to the virtual memory system.

Other dynamic software analyses cannot use this technique, so the remainder of this dissertation focuses on novel hardware-based acceleration mechanisms. The first, Testudo, allows faster and more accurate dataflow analysis sampling by storing meta-data in an on-chip cache and stochastically removing these values whenever it overflows.

Next, a method for performing demand-driven data race detection using existing hardware performance counters is detailed. These counters make inter-cache sharing events visible to software, allowing the data race detector to be enabled only when needed.

Finally, fine-grained memory monitoring is a crucial for every analysis studied in this work. This dissertation closes by examining a hardware mechanism for allowing unlimited fine-grained watchpoints, which can be used to reduce the overheads of many analyses.

In total, this dissertation demonstrates that hardware designers can play a role in mitigating software bugs, as many of the acceleration mechanisms explored in this work require hardware support in one form or another. Taken together, these works represent a step forward for accelerating dynamic software analyses and distributing them to end-users.

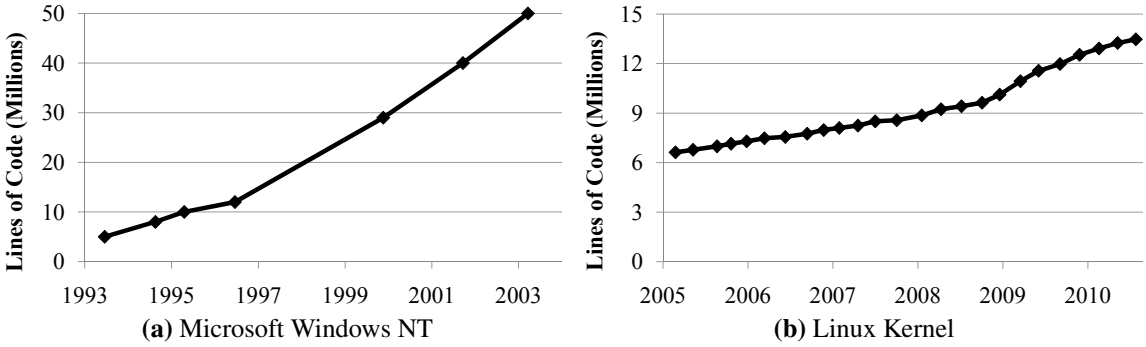
Chapter 1
Introduction

> It was on one of my journeys between the EDSAC room and the punching equipment that "hestitating at the angles of stairs" the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.
>
> *Memoirs of a Computer Pioneer*
> Maurice V. Wilkes

High-quality software is difficult to create. Complex applications not only require optimized algorithms underlying their logical structure, but they must also be implemented efficiently and correctly. As programs increase in complexity, this correctness and efficiency are more difficult to attain, leading to bug-ridden software that, even when working, is disagreeably slow.

Billions of dollars and millions of man-hours are spent each year attempting to build, modify, and maintain innumerable programs. Though the field of software engineering exists to formalize methods of creating high-quality software [1], our ability to accurately build such complex artifacts lags behind our appetite for new features and speedier results. On the whole, this leads to modern applications that are riddled with errors that often frustrate their users and developers. As Bessey *et al.* state, "Assuming you have a reasonable [software analysis] tool, if you run it over a large, previously unchecked system, you will always find bugs" [23].

Such bugs can lead to significant hardships for both users and designers. The National Institute of Standards and Technology estimated that, nearly a decade ago, software errors cost the United States economy up to $59.5 billion per year [212]. These errors can also result in security-related problems, such as the thousands of exploitable vulnerabilities publicly disclosed yearly by the National Vulnerability Database [152] or sold to criminals and government agencies through gray and black markets [82, 140]. Vulnerabilities like these contributed to an estimated $67 billion lost by American companies in 2005 due to



(a) Microsoft Windows NT **(b)** Linux Kernel

Figure 1.1 Lines of Source Code in Windows and Linux Increasing Over Time. As software becomes more mature, it has a tendency to increase in size and complexity. New features are added and old features remain for backwards compatibility. **(a)** shows the number of lines of source code in the Microsoft Windows NT operating system from 3.1 up to Server 2003. **(b)** lists the number of source code lines in the Linux kernel from version 2.6.11 to 2.6.35.

computer crime [222]. It has also been speculated that they are used for espionage and cyberwarfare attacks [193].

1.1 Complexity Causes Software Errors

One factor leading to such errors is the tendency for software to become more complex over time. Figure 1.1 illustrates the increase in lines of code within two popular pieces of software. Figure 1.1a shows the number of source code lines in the Microsoft Windows NT operating system from its initial public release in 1993 to Windows Server 2003 [130, p. XXI]. Over the course of this decade, the operating system (including both kernel and user-level code) increased in size by 10×, from 5 million to 50 million lines of code. Figure 1.1b graphs the same data for the Linux kernel from version 2.6.11, released in March, 2005, to version 2.6.35, release in August, 2010 [48]. This five year period saw the code base double from 6.6 million lines of code to 13.4 million.

While lines of code is not a quantitative metric of complexity, Maraia also listed a second indication of the increased complexity of the Windows operating system: the number of developers working on the product, as shown in Figure 1.2, increased at an even faster rate than the code [130, p. XXI]. Over the course of that decade, the programming team increased from 340 to 4400 people, a 13× jump. As Brooks points out (as have many others), the time spent developing software grows faster than linearly as team size increases, because many more interactions must take place between developers [73]. In short, the complexity of a system such as Microsoft Windows increased rapidly over time.

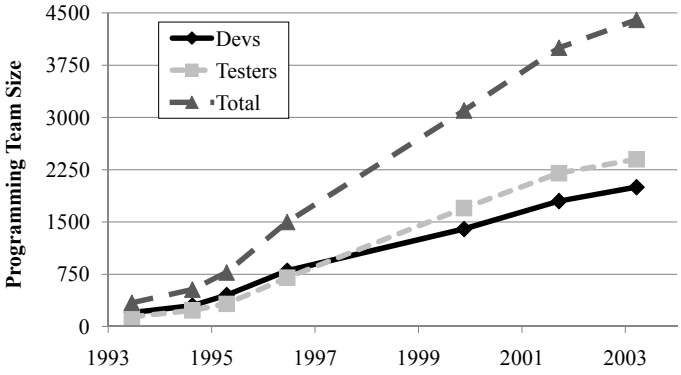


Figure 1.2 Number of Windows Developers Increasing Over Time. Not only did the Microsoft Windows code size increase by 10× between NT 3.1 and Server 2003, the total development team increased by almost 13×. This supports the assertion that the software has become more complicated over time.

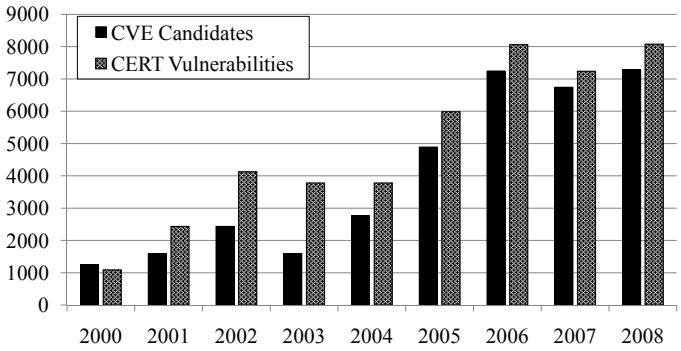


Figure 1.3 Number of Security Vulnerabilities Increasing Over Time. These numbers come from two general security vulnerability databases, the Mitre Corporation's CVE list and the CERT vulnerability database. As software has proliferated, its complexity has resulted in more total vulnerabilities reported each year, growing 7–8× in eight years.

This data, of course, only shows that individual programs or software releases get larger over time. Upon its completion in 1957, barely a decade after the invention of the modern digital computer, the SAGE project already had more than 100,000 instructions and five hundred programmers [102, pp. 105–108]; IBM's System/360 used over 5 million lines of code by 1968 (similar to the amount seen 25 years later in Windows NT 3.1) [179, p. 66]. Nonetheless, we can see the negative effects of increasing complexity on the software corpus by studying bug reports. Figure 1.3 shows the number of security vulnerabilities reported to Mitre Corporation's Common Vulnerability and Exposures (CVE) list [49] and the number of general software vulnerabilities cataloged by CERT [99] from the years 2000 to 2008 (the last year with currently available statistics). Over this eight year period, the number of security vulnerabilities publicly reported each year increased by 8×, another indication of complexity that corroborates the assertion that complex high-quality software is difficult to create.

A simple example of the type of bug that can lead to such security flaws is the buffer overflow. This common problem can result from an unexpected input forcing a function to modify memory far beyond the end of an array. Such inputs can be used to insert and execute malicious code into an unsuspecting program [4]. This can occur in large applications when functions written by different programmers expect contrary things of a shared buffer. An example of this can be seen in a 2005 security flaw in the Lynx newsgroup reader [155]. This exploit attacks a function, `HTrjis()`, that adds escaped characters to an existing string while moving it between two buffers. `HTrjis()` assumed that the destination buffer was large enough to contain strings with multiple new characters, while the function that calls `HTrjis()` instead assumed that both buffers could be the same size. Such implicit assumptions are difficult to curtail in large software projects can cause obscure errors.

While the use of safer languages such as Java and C# have reduced the occurrences of some types of errors in user-level code, flaws in the highly complex runtime environments of such languages are still a source of dangerous errors. Adobe Flash is a widely deployed platform for running ActionScript programs, yet bugs in the underlying scripting engine are frequently exploited to allow these programs to attack the host computer [71]. Additionally, complex applications can still suffer from numerous non-security bugs, despite running in more constrained environments.

1.2 Hardware Plays a Role in the Problem

The increasing complexity of the underlying hardware is another cause of software bugs. Hardware architects often add intricate features to increase the maximum performance of their product. The inattentive programmer may find that these features hurt performance if used haphazardly, as complicated out-of-order execution engines, branch predictors, prefetch units, and similar features require a great deal of effort to effectively utilize. Beyond first-order estimations, it is often difficult to conceptualize the performance that will result from a particular implementation of an application. Many software developers instead rely on profiling to observe the true efficiency of a piece of software, since keeping track of every possible hardware feature is a mind-bending proposition.

To this end, they often use mechanisms such as hardware performance counters when optimizing their code [208]. These devices count hardware events that can reduce software performance, such as branch mispredictions and prefetch misses, and can be used to pinpoint sections of code that are not operating efficiently on the underlying hardware. Over the last 15 years, Intel has quadrupled the number of countable events in their commercial x86 line

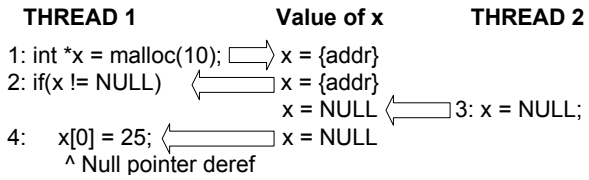


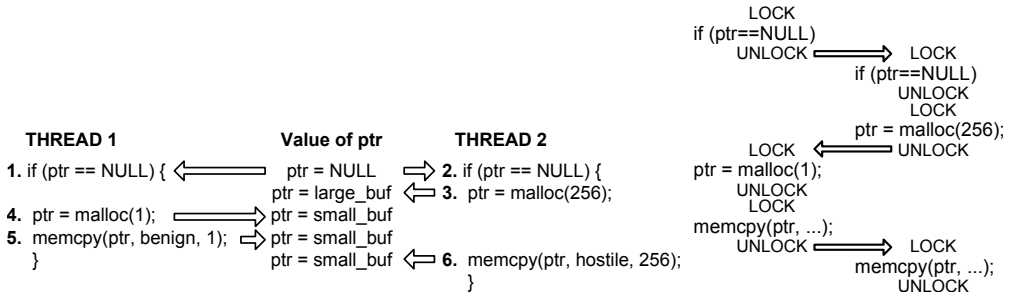
Figure 1.4 A Data Race Causing a Software Bug. This concurrency bug example demonstrates how a data race can cause a non-intuitive program error. The code in thread 1 should be safe, but because another thread can change the shared pointer between the check and dereference, thread 1 might crash.

of processors, indicating that it is becoming more difficult to optimize code for the myriad of new and complex hardware features [58].

Besides making performance optimizations more difficult, complicated hardware features can also lead to correctness and security problems. A hardware feature that undoubtedly falls in this class is shared-memory parallel processing. Originally proposed in the 1990s [86, 176, 224], single-chip multiprocessors were studied further in the early part of the following decade [16, 85, 213]. Consumer-level CPUs have since advanced from mostly single-core devices to chips with two or more cores, starting with the releases such as AMD's Athlon 64 X2 [143], IBM's PowerPC 970MP [98], Intel's Pentium D [90], and Sun's UltraSPARC T1 [113] in 2005. In the years since, commodity processors have increased their core counts up to 12 [5], server chips up to 16 [201], and some embedded chips up to 100 [3]. Besides these homogeneous multicore systems, there is also a push for heterogeneous multiprocessors, or multicore chips where each core can have a different ISA or microarchitecture [114].

Because developers need to worry about problems such as data races, deadlocks, atomicity violations, memory consistency errors, and non-deterministic execution in parallel code, it is much more difficult to program the concurrent software needed by these systems than serial programs that perform the same task [135]. Such difficulties have been noted numerous times over the years; from the Cray X-MP supercomputer [150] to the Sega Saturn video game console [171], scores of developers have found programming parallel processors difficult. Heterogeneous systems, such as IBM's Cell processor, are similarly notorious for their difficulty to program [192]. Despite these issues, architects now rely primarily on increasingly parallel processors to deliver further performance gains, as those from Dennard scaling have dropped off in the last decade due to power density concerns [175, pp. 1–5].

Taken in total, multiprocessors have entered the mainstream, bringing parallel programming and its associated concurrency bugs with them. Figure 1.4 shows a simple example of a concurrency bug that causes a program to crash because of a null pointer exception. When these two threads access a shared pointer without appropriate locking, the interleaving



(a) Data Race **(b)** Atomicity Violation

Figure 1.5 Concurrency Bugs Causing Security Errors. These examples show that concurrency bugs can cause more than just program crashes. In **(a)**, both threads attempt to initialize a shared pointer. Due to the lack of synchronization, thread 1 allocates a small buffer, and thread 2 copies a large amount of data into it. The version of the code in **(b)**, has locks around every access to the shared pointer. There is no data race, but it is still possible to order the threads incorrectly. In both cases, the incorrect parallel execution can cause buffer overflows.

of the accesses may cause the first thread to crash when dereferencing a pointer, even if it appropriately checked that the pointer was not NULL immediately beforehand.

Figure 1.5 shows an example concurrency bug modeled after CVE-2010-3864, a security flaw found in recent versions of the OpenSSL TLS library, which also allowed two threads to access a shared pointer without synchronization [160]. In some circumstances, each thread may attempt to initialize the pointer to a different heap object and subsequently attempt to copy data into the object. If, however, the accesses to the pointer are interleaved (as shown in Figure 1.5a), one thread may attempt to copy a large amount of data into a small heap object, causing a potentially dangerous heap overflow [180]. Similarly, Figure 1.5b demonstrates how atomicity violations, much like data races, could result in similar software problems. Even though the accesses to the shared pointer have locks surrounding them, meaning that there are technically no data races, the threads can still be ordered in such a way that the buffer overflow exists

Other decisions by hardware manufacturers have led to different kinds of software errors. Though there were pushes in the 1970s and 1980s for capability-based architectures, systems that would only allow areas of memory to be accessed by code with appropriate permissions, most of these architectures had limited market penetration [122]. As such, most current architectures leave poorly written software vulnerable to memory errors. Similarly, though most security attacks can still be carried out on Harvard architecture systems [72], the prevalence of von Neumann architectures makes these attacks simpler.

1.3 Attempts to Alleviate the Problem

Unfortunately, while current processors provide facilities such as performance counters and branch recorders to allow engineers to tune software performance [121], few analogous features exist to help correctness. What features *do* exist are primarily focused on a small class of security issues. The UltraSPARC T2, for instance, includes hardware support for RSA, DSA, RC4, 3DES, and AES encryptions algorithms [75]. Intel's processors now also contain hardware support for AES [187]. These features are primarily used to increase performance, but also offer some protection against side-channel attacks.

There are proposals from the industry for mechanisms like transactional memory [88], in an attempt to make parallel programming easier. However, efforts from processor manufacturers to help software developers find other problems, such as buffer overflows, are limited primarily to the research community. Nonetheless, whether hardware developers help or not, though, those in the software realm must attempt to solve these problems if they are to release working programs. To that end, software engineers have built a number of systems to help themselves.

One of the first techniques that the software development community took to heart when attempting to more accurately build their programs was the concept of software engineering [94]. In essence, this means that the programmers should follow structured design principles, based on well-researched theoretical concepts, rather than building their products solely from experiences learned through trial-and-error. Professional societies such as the ACM and IEEE have pushed such methods into educational curricula in an attempt to train stronger developers [1]. Other efforts, such as coding standards, list rules that programmers should follow in order to avoid common mistakes [198]. While such efforts are laudable and helpful, they do not completely solve the problem, as outlined earlier.

Another direction the software community has taken to reduce errors in their code is the introduction of newer, more powerful programming languages. This can be seen with the original introduction of high-level languages, which allowed developers to more succinctly describe their algorithms and hid the complexities of the underlying hardware compared to machine code [111]. Managed languages such as Java remove many of the memory problems found in programs written in languages like C or C++, while scripting languages such as JavaScript and Python hide even more low-level complexity from designers.

For a number of reasons, this dissertation will not focus on this method of reducing software errors. First, though newer languages can make it more difficult to cause certain types of errors, they often open up new styles of bugs. Errors in modern web scripts, for instance, frequently result in SQL injection attacks, even if they don't suffer from buffer

overflow attacks. As Zeev Suraski stated when discussing security features in PHP, the scripting language he helps develop, "[s]ecurity bugs are out there, in fact in web apps they're pretty much a plague (irregardless of the language)" [211]. Beyond the ability of careless programmers to create bugs in any language, it is also the case both that legacy software continues to exist and increase in size (as discussed earlier) and that new software is still created in older languages [110].

Instead, this work will look at another mechanism that developers use to improve their code: software analysis tools. That is, programs that look at other pieces of software and help developers find errors, mistakes, and misconfigurations after they already exist, but hopefully before they can cause problems.

1.4 Analysis Yields Better Software

Software analysis tools are programs designed to inspect other pieces of software in order to find errors. These can be as simple as searches for insecure functions, such as calls to `memcpy()`, or may be as complex as attempting to walk the program through every reachable path in its control structure. In general, software analyses tools focus on finding particular classes of errors and require a number of tradeoffs to meet these goals. Perhaps the most common way of classifying an analysis tool is by whether it tests the software without running it, or whether it does so alongside the execution of the program. The former are called *static analyses*, while the latter, which are the focus of most of this dissertation, are referred to as *dynamic analyses*.

1.4.1 Static Software Analysis

Static analysis tools automatically reason about a program's correctness by testing the source code for certain formal properties. One of the most famous tools in this area is *lint*, a tool that searches C code for constructs that, while not technically illegal under the language specification, are likely to cause runtime errors [107]. Such tools are more powerful than simple string searches, but may have problems finding complex errors that involve logical mistakes or correct code used in incorrect ways. Some researchers have therefore looked at formal analysis methods, where a tool tries to perform rigorous mathematical deductions to see if the software under test can reach an erroneous output [63].

Static analysis tools have been used to find numerous bugs [23], but may not work under all circumstances. Formal methods can have problems analyzing large programs due

to the computational complexity of the underlying algorithms. The CompCert project to formally verify a compiler chain that goes from a subset of C (excluding, for instance, `goto` statements) to PowerPC assembly code took multiple person-years of effort. In addition, it produced enough unique ideas that simply completing it yielded a string of publications in venues as prestigious as the Communications of the ACM [120]. Performing such an effort on complicated programs with less formal grounding than compilers may prove very difficult indeed.

Static analyses may also have problems with false positives, as the developers of the Clang static analyzer point out [8]. Because the tests may make assumptions about how the program will be used at runtime, they may falsely flag errors that could never occur on a real system. This lack of dynamic information can also lead to false negatives.

1.4.2 Dynamic Software Analysis

Some errors can only be found by using knowledge about how the software runs. Therefore, dynamic software analysis tools monitor the runtime state of a program and observe situations that may only arise during actual execution. Because they work alongside the executing program, and are therefore able to check any executed path, these tests can observe situations that static analyses have difficulty checking. An added benefit is the ability to run in conjunction with static analyses, making their design decisions largely orthogonal. This section reviews a series of dynamic analyses, as well as their benefits and disadvantages.

Assertion Checking. Some analysis systems begin their search for errors by statically analyzing the code for operations that can be proven to be either secure or insecure. The tool can then insert dynamic checks around the remaining operations to ensure their correctness at runtime. CCured, for example, uses this combination to ensure type safety [67] and find memory errors such as buffer overflows [162]. The overheads observed with these tools depend both of the complexity of the assertions and the power of the static analysis. The simple memory assertions inserted by CCured slow a selection of computationally-intensive applications up to 87% [47].

Dynamic Dataflow Analysis. Dynamic dataflow analyses associate shadow values with memory locations, propagate them alongside the execution of the program, and check them to find errors. Valgrind's Memcheck tool, perhaps the most widely used of this type of analysis, keeps track of whether a value has been initialized (or is derived from an initialized location), and can find memory leaks and other such errors [167]. Similarly, dynamic heap

bounds checkers, such as Valgrind's Annelid tool, associate pointers with heap allocations and verify that every dereferenced value points to a location within the valid range of its region [165]. Taint analysis follows "untrusted" data through a process to determine whether it is used without being properly validated [170].

The performances of these systems can vary widely and depend on the type of analysis, the optimizations applied, the system used to construct the tool, and the inputs given to the particular execution of the program under test. Memcheck has overheads of about 20×, while Dr. Memory, a similar tool built using a different instrumentation tool, has overheads of about 10× [28]. Overheads for taint analysis tools range from 50% [27] to upwards of 150× [93], and greatly depend on the accuracy and portability of the analysis and instrumentation tools.

Data Race Detection. Data races occur when two parallel threads of execution access a single shared memory location without a guaranteed order of operations between them. More formally, a data race occurs when two or more threads access a single memory location, at least one access is a write, and there is no enforced order between the accesses [168]. These situations can introduce unintended, and sometimes dangerous, values into the program. Because statically finding all data races in a program is NP-hard [169], most race detectors only attempt to find some data races. Dynamic data race detectors search for these problems through methods such finding violations of Lamport's happens-before relation [116], a subset of this called the lockset algorithm [191], or a hybrid combination of the two [173].

Their performance depends heavily on the tool and its optimizations, as well as the amount of sharing seen within any particular dynamic execution of the program. Tests done for this dissertation put current versions of Helgrind, a race detector built using Valgrind, at about a 100× slowdown, while Serebryany and Iskhodzhanov list slowdowns of between 20× and 50× for Google ThreadSanitizer, a commercial data race detector also built using Valgrind [199].

Atomicity Violation Detection. Atomicity violations are concurrency errors that are similar to, yet distinct from, data races. In parallel code, a programmer may desire that some collection of variables be accessed atomically (i.e. no other thread accesses any of the variables until the first thread finishes working on them). However, even if variables are locked in a way that prevents data races, this atomicity is not guaranteed [68]. Lu *et al.* looked at a number of concurrency bugs in large open-source programs and found that atomicity violations were quite common [125].

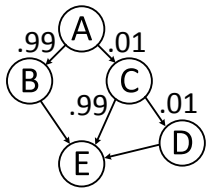


Figure 1.6 Dynamic Analyses May Not Always Catch Bugs. This is a control flow diagram with weighted random probabilities at each branch. It takes an average of 10,000 executions to run the code within basic block **D**. If an bug exists there, a dynamic analysis will only find it 0.01% of the time.

Atomicity violations are difficult to automatically detect, as they are a disconnect between what the programmer *wants* and what she *commands*. While the designer may assume that two regions are executed atomically, a tool cannot learn this assumption from the code. As such, current tools that find these errors focus on training over many known-good executions, recording the observed atomic regions, and later verifying that these regions are always accessed atomically. The online detection mechanisms for these tools have slowdowns that range from 25× [126] to upwards of 400× [146].

Symbolic Execution. Symbolic execution follows a program as it runs and, rather than working with concrete values for any particular variable, attempts to calculate what *possible* values could exist in the program. As the program calls functions with known return values, passes conditional statements, and uses concrete values, the limits on any particular variable can be constrained. These symbolic values can be checked to find inputs that could cause errors such as memory corruption [118].

The system by Larson and Austin looks only at the control path executed on any particular run, constraining symbolic variables to match the bounds that are imposed by the current path. It tests memory accesses along the executed path to find if they could be corrupted by slightly different inputs that still made it to this state [118]. Tools like DART and KLEE, on the other hand, start with completely unconstrained inputs and systematically search for bugs in all possible control paths [30, 74]. Godefroid *et al.* claim that their symbolic execution is "many times slower than [running] ... a program", while Larson and Austin show overheads between 13× and 220×.

The powers of these dynamic analysis tools have limitations, of course: only the portions of an application observed during a particular execution can be analyzed, meaning they are unable to catch errors that lie on unexecuted paths. Consequently, dynamic analyses benefit from seeing large numbers of executions with a variety of inputs, allowing them to test more paths. Figure 1.6 demonstrates a simple example of this. If an error existed in basic block

D of this control flow graph, a dynamic analysis would be unable to observe it until the program executed the code within that block. In this case, it would take an average of 10,000 executions of the program before the tool would observe the problem. Unfortunately, the performance overheads introduced by these approaches make it difficult to observe many different executions. These large overheads lead to myopic analyses because they limit the number and quality of test inputs that a developer can execute before shipping a piece of software, reducing the tool's effectiveness.

1.5 Contributions of this Work

Ideally, end-users would run these dynamic software analyses, as they execute the program more often than developers and use inputs that test engineers may not think to examine. Unfortunately, it is unlikely that end-users would be willing to use programs that have been modified to be $2\times$, let alone $200\times$, slower. Few would purchase a faster computer to run software analyses, and it is doubtful that users who pay for CPU time (such as cloud computing customers) would wish to centuple their costs in order to find errors.

This dissertation will present methods for accelerating dynamic software analyses so that they can be distributed to large populations of end-users. This enables a form of crowd-sourcing for these tests, where users would combine their efforts to find previously unknown bugs. Test distribution could be done by allowing users to opt into testing the program at acceptable overheads, or perhaps requiring that beta versions of the software include analyses. In either case, these users would report potential errors found by the analyses in a similar manner to current crash reports, such as with Breakpad [76] or Windows Error Reporting technologies [32]. This dissertation will not go into details of the mechanisms for bug triage in a system of this type. However, Liblit and others have done a great deal of work designing mechanisms for pinpointing the cause of errors using the numerous reports sent from highly distributed bug detection systems [40, 124], and software such as Socorro is used in commercial software to consolidate crash reports from distributed sites [77].

The first technique that will be discussed in this dissertation takes advantage of the fact that complex software systems, those that are most likely to have difficult-to-find bugs, are also likely to have large user populations. Apache httpd, for instance, runs on over 82 million servers [164], and Microsoft's Office XP sold over 60 million licenses in its first year on the market [138].

As previously mentioned, if the overheads of individual analyses were low enough, these large populations could test this software at runtime. If these users were all running

dynamic analyses, each user could analyze a small portion of the program and still help attain high aggregate bug coverage. The slowdowns that each user experiences can therefore be reduced, as the analysis tool need not be enabled at all times. Chapter 2 presents an argument for sampling dynamic analyses in order to reduce their overheads. In other words, the slowdowns seen by any individual user are reduced by analyzing only limited portions of the program at any one time. That chapter then details a novel mechanism for sampling of dynamic dataflow analyses using only the virtual memory hardware available on modern microprocessors.

Hardware should also change to make program analysis easier, in an attempt to reduce the programming burden caused by complex hardware facilities. It is in the processor vendors' best interests to make systems that help software analyses, as users may otherwise migrate to other, more easily programmable systems. The remainder of this dissertation therefore describes new hardware features can help in this regard. Chapter 3 discusses a hardware design that can perform dynamic dataflow analysis sampling more accurately and at lower runtime overheads than the software-only system. By including taint analysis propagation hardware in the pipeline and an on-chip meta-data storage system known as a sample cache, this system can sample a variety of dataflow analyses in a byte-accurate manner and can perform taint analysis sampling at no overhead whatsoever.

While sampling can reduce the overheads of some tests, accelerating others require new hardware features. Chapter 4 details a method of significantly accelerating another type of dynamic analysis, data race detection, by utilizing hardware performance counters that are available on modern microprocessors.

The systems described in Chapters 3 and 4 are relatively application specific. The former looks at dataflow analyses, while the latter works only on data race detection. For a hardware design to have a strong change of existing in modern commercial processors, it should be as general as possible. The more potential users a new hardware feature has, the more likely it will be worth the design and verification effort. Along these lines, Chapter 5 discusses a generic hardware addition, virtually unlimited fine-grained watchpoints, which can be used to accelerate many dynamic software analyses. By allowing the hardware to inform software that is touching "interesting" data, a large runtime overhead can be removed from many analysis tools

Finally, Chapter 6 concludes the work and discusses future research directions that could continue down the path of exploring hardware to make better software.

In a concise statement, this dissertation will argue that we must design appropriate software analysis techniques to ensure that future software is robust and secure. The high overhead and relatively low accuracy of some dynamic software analyses can be mitigated

by using sampling techniques to spread these tests over large user populations. Novel hardware techniques can be used to further refine these tests into useful, low-overhead tools in the software development chain. Because different analyses require different hardware support, this dissertation investigates the many different mechanisms that hardware can present to software analysis tools. It looks at application-specific mechanisms designed to accelerate distributed analyses, existing hardware used in novel ways, and new designed meant to speed up many different analysis tools.

Chapter 2
Distributed Dynamic Dataflow Analysis

> Software debugging has emerged from a Victorian era in which it was done, but not talked about much.
>
> *Some Requirements for Architectural Support of Software Debugging*
> Mark Scott Johnson

As discussed in Chapter 1, dynamic tests such as taint analysis and data race detection can help create robust software. Because they are unable to catch errors that lie on unexecuted paths, these analyses benefit from seeing large numbers of executions with a variety of inputs. End-users could run these tests, then, checking the abundant situations they encounter.

Unfortunately, these systems suffer from very high runtime overheads, as reiterated in Table 2.1. Such large overheads present a twofold problem: they limit the number of test inputs that a developer can execute before shipping a piece of software, and they severely reduce the number of users willing to run the analyses. In both cases, high overheads hamper the tool's effectiveness.

Researchers have proposed numerous solutions to help combat this problem. Zhao *et al.*, for instance, showed mechanisms to accelerate shadow variable accesses [234]. There have also been works on parallelizing analyses [172] and decoupling them from the original execution [44, 189]. Unfortunately, these techniques do not completely eliminate slowdowns. Umbra reduced the overhead of shadow data accesses from 8–12× to 2–4× [234], but this is still more than most users would tolerate.

This chapter will make the argument that an excellent solution to this overhead problem is to sample the analyses, analyzing a limited portion of the program during any individual execution. Ideally, the tool's ability to detect errors will be proportional to the slowdown, which would allow users to control the overheads they experience while finding some percentage of the program's errors.

Table 2.1 Overheads from a Selection of Dynamic Analyses. Dynamic analyses cause extremely high overheads because they insert extra instructions into the runtime of the program under analysis. If each original instruction in the program required only one extra instruction for the analysis, there would hypthetically be a 2× slowdown. Analyses such as symbolic execution, which must perform numerous heavyweight calculations, see overheads of over 200×.

Analysis	Slowdown
Assertion Checking	5% – 2×
Dynamic Dataflow Analysis	50% – 150×
Data Race Detection	8× – 100×
Atomicity Checking	25× – 400×
Symbolic Execution	10× – 200×

Because the entire dynamic execution will not be analyzed while sampling, no test is guaranteed to find all observable errors. However, distributing these sampled analyses across large end-user populations can offset the lower error detection rate. This larger population will also see many more inputs and dynamic states than a developer running a small test suite, further increasing the power of distributed bug-hunting systems.

2.1 Introduction to Sampling

Sampling and distributed debugging are not new topics. In such systems, small portions of the program are analyzed during each run, allowing performance increases over a system that constantly checks the program. In a well-designed sampling system, the ability to observe errors is related to both the sampling rate and the performance; lowering the sampling rate will increase the performance but subsequently lower the probability of finding any particular error.

This accuracy reduction can potentially be offset by the large increase in the number of testers. As a simple example: if a sampling analysis catches 10% of the executed errors in a program, and were used to test the example in Figure 1.6, it would catch the error once in every 100,000 executions on average. If 5000 users were to run these analyses, each would only need to run the program an average of 20 (rather than 10,000 or 100,000) times before the error would be observed somewhere, and a report could be returned to the developer. This increase in population not only offsets the accuracy loss due to sampling, but alleviates some of the problem of only observing errors on executed paths.

The remainder of this section reviews some of the previous works in this research area that have informed the contents of this dissertation.

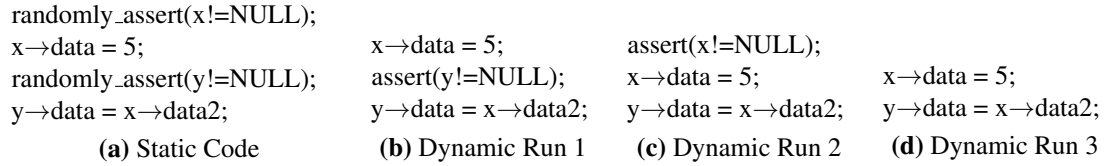


Figure 2.1 Sampling for Assertion Analyses. This system has a random probability of performing any particular test. It is possible that `x==NULL` in, e.g., **(b)**. However, with a large population, errors will be caught proportional to the percent of time each assertion is enabled.

2.1.1 Sampling for Performance Analysis

While performance analysis is not the same as bug-finding, it is educational to review the works on sampling in this area. Arnold and Ryder showed software mechanisms for sampling performance analyses [9], as did Hirzel and Chilimbi [92]. These mechanisms, in a broad sense, enter the analysis on certain software conditions, profile some instructions, and then leave analysis until the next condition. Hirzel and Chilimbi showed that they were able to gather accurate performance profiles at overheads as low as 3–18% (compared to 30%–10× for non-sampled analysis).

The concept of sampling is so well ingrained into the performance analysis community that commodity processors have dedicated performance sampling hardware. While taking interrupts on performance counter rollover is one of the most common ways of analyzing samples of events [208], modern processors also include dedicated performance sampling facilities such as Intel's Precise Event Based Sampling (PEBS) [100] and AMD's Instruction-Based Sampling (IBS) [62]. Such mechanisms amortize interrupt costs by automatically storing information about a subset of instructions or events into physical memory buffers without involving software.

2.1.2 Sampling for Assertion Checking

Liblit *et al.* described an instruction-based method of sampling assertion tests, as demonstrated in Figure 2.1 [123]. As long as the checks are not related to one another, it is possible to perform sampling by only enabling a random subset of the assertions during each execution. They showed that performing this type of sampling was able to lower the maximum overhead of their analysis from 2.81× to 26% when sampling one out of every thousand checks. They demonstrate that such sampling rates can find uncommon errors with user populations of moderate size.

Hauswirth and Chilimbi performed similar types of sampling in order to do memory leak detection [89]. Their sampling mechanism enables checks for groups of instructions

so as to reduce the overhead of deciding when to enable the tool. They also showed that sampling reduces overheads (from an estimated 5× to 5%), unfortunately, about 7% of their error reports falsely identified active pages as leaked because their tool was disabled during a period where a page was actually used.

2.1.3 Sampling for Concurrency Tests

LiteRace logs memory accesses and synchronization operations and performs offline data race detection on them. As Marino *et al.* point out, logging a small, random sample of the memory accesses will allow many races to be detected while significantly reducing the logging overhead. All synchronization points must still be logged, or the sampling system may cause false positives. By choosing to log cold-path code at a much higher rate, they were able to reduce the runtime overhead from 651% to 28%, while still detecting about 70% of the observable data races [131].

PACER performs online race detection sampling, but enables the detector for long intervals. When the analysis is later disabled, each variable is sent through one last check when it is next accessed. In doing so, Bond *et al.* showed that it is possible to find races in rough proportion to the sampling rate while still allowing control over performance [26].

Finally, Erickson *et al.* described DataCollider, a system that samples a small number of variables in the Windows kernel and sets hardware watchpoints on them. By waiting for the hardware to tell them that some thread has accessed the variable, they are able to significantly reduce the amount overhead that their analysis tool causes. However, because current hardware has an extremely limited number of these watchpoint registers (as will be described in Section 5.2.1), this system *must* use sampling [65].

There is little work on sampling atomicity violation detection in the classic sense. Rather than attempting to sample previous atomicity violation mechanisms, Jin *et al.* instead looked at a new way of detecting atomicity violations by keeping track of samples of data sharing statistics from many user runs. They then traced program crashes back to these errors using statistical analyses, finding both data races and atomicity violations [105]. This type of analysis would not be possible without sampling.

2.1.4 Sampling for Dynamic Dataflow Analyses

There is little previous work on sampling dynamic dataflow analyses. Unfortunately, as will be described in Section 2.2.3, the instruction-based sampling mechanisms used for performance analysis, assertion checking, and concurrency bug testing do not work with dynamic

dataflow analyses. Because these methods of sampling do not maintain knowledge of the underlying shadow dataflows, they can lead to extraneous false negatives and undesired false positives.

Arnold *et al.* showed an object-based analysis sampling method that used the extensive program information available to a Java virtual machine to avoid these problems. Instead of checking a subset of dynamic tests, as instruction-based sampling systems do, their system only performed analyses (ranging from assertion checks to typestate checking) on a subset of instantiated objects [10]. Unfortunately, this method of sampling would be difficult to do, if not impossible, in a system with less access to language-level constructs than a JVM.

This chapter presents a novel software-based dynamic dataflow analysis sampling technique that works on unmodified binaries compiled from any language. It uses an overhead manager that observes the runtime impact of the analysis procedures and can constrain the analysis to a set of stochastically chosen shadow dataflows if performance degrades below a user-specified threshold. This allows developers to run larger tests, and lets users cap the slowdown they experience, which permits large end-user populations to run analyses that were formerly restricted to a development setting. Beta testers, production servers, and even mainstream users can then analyze programs in the background.

We built a prototype of this distributed dataflow analysis system by modifying the Xen hypervisor [15] to work with an augmented version of the emulator QEMU [20]. This modified emulator implements a taint analysis system, which was used to run experiments that demonstrate how sampling provides fine-grained control of overheads while still delivering high-quality dataflow analyses. Experiments with real-world security exploits show that this solution easily exposed the security flaws in test applications with a small population of users while significantly reducing the performance overhead that individual users experienced.

This work, much of which was originally published in the 2011 paper "Highly Scalable Distributed Dataflow Analysis" [78], makes the following contributions:

- It shows that **previous code-based dynamic sampling systems are inadequate for dynamic dataflow analyses**. They sample programs' instructions and thus do not take into account the effects of disabling the analysis on shadow dataflows. This leads to both false positives and a large number of false negatives.
- It presents a new technique for performing sampling in dynamic dataflow analysis systems. By sampling dataflows, rather than instructions, it is able to **effectively control analysis overheads** while avoiding many of the inaccuracies of previous sampling systems.

- By allowing individual users to control the sampling rate, this technique **enables the distribution of dynamic analyses to large populations**. This chapter will show that this sampling system is able to observe a large fraction of the errors that a traditional analysis can discover, but at much lower performance overhead.

The remainder of this chapter is organized as follows: Pertinent background works are reviewed in Section 2.2. Section 2.3 presents a method of dataflow-oriented sampling and Section 2.4 details an implementation of this algorithm. Section 2.5 presents an evaluation of this implementation and demonstrates that it can enable strong analyses while controlling performance overheads. Finally, this chapter is concluded with recommendations for future work, in Section 2.6.

2.2 Background

This section presents background concepts that are leveraged in this dynamic dataflow sampling system. It first summarizes how dynamic dataflow analyses operate and then looks at demand-driven analysis, a technique that offers, but does not guarantee, lower performance overhead when performing dynamic dataflow analyses. Finally, it discusses previous works on sampling dynamic analyses and shows how they are inadequate for dataflow sampling.

2.2.1 Dynamic Dataflow Analysis

Dynamic dataflow analyses are tests that associate shadow values with memory locations, propagate them alongside the execution of the program, and check them to find errors. Nethercote and Seward built Valgrind, a popular dynamic binary instrumentation tool that is the basis for a number of dynamic dataflow tools because of its strong support for shadow memory [166, 167]. Memcheck is used to find runtime memory errors, such as uses of undefined variables and memory leaks [200], while Annelid is the preliminary design of a dynamic bounds checker [165]. TaintCheck, one of the first software-only taint analysis systems, is also implemented using Valgrind [170].

Figure 2.2 shows an example of dynamic dataflow analysis as performed by a heap bounds checker, a mechanism that associates heap meta-data with pointers. The meta-data details the heap objects to which each pointer refers, and it is moved throughout the program along with the pointers and their derived values. When a pointer is dereferenced, the checker verifies that it correctly points within its associated object.

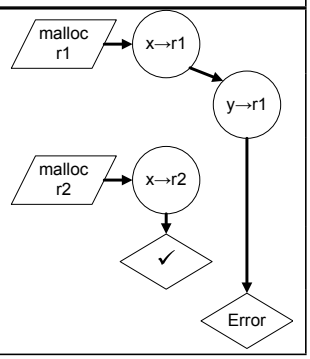
Dynamic Instructions	Analysis Operations	Data Values	Shadow Values	Dataflow
ptr x = malloc(10);	create r1 x.region ⇐ r1	x: a	r1.lo: a, r1.hi: a+9 x.region: r1	
ptr y = x + 15;	y.region ⇐ x.region	x: a y: a+15	r1.lo: a, r1.hi: a+9 x.region: r1, y.region: r1	
x = malloc(20);	create r2 x.region ⇐ r2	x: b y: a+15	r1.lo: a, r1.hi: a+9 r2.lo: b, r2.hi: b+19 x.region: r2, y.region: r1	
dereference(x);	x.region.lo ≤ x ≤ x.region.hi? **Yes. No Error.**	”	”	
dereference(y);	y.region.lo ≤ y ≤ y.region.hi? **No. Boundary Error.**	”	”	

Figure 2.2 Example Dataflow Analysis. Ths figure illustrates how a dynamic bounds checker associates allocated memory regions with pointers and, by propagating the association to derived pointers, builds a dataflow that connects pointers to regions. By checking a dereferenced pointer's address against the bounds of its region, it is possible to find heap errors such as overflows.

In the figure, parallelograms represent points that allocate memory from the heap. These calls to `malloc()` normally only return pointers to the allocated regions. In this heap analysis system, they also produce meta-data that describe the regions and associations between pointers and these regions (represented in the figure by circles that list their region associations). As pointers are copied, the shadow values associating pointers with regions are also copied into the destination variables; this flow of shadow data is represented by arrows between two circles. Finally, diamonds represent the validity checks that take place when a pointer is dereferenced. A checkmark means that a pointer correctly addressed its associated region, while an error means it did not.

In the example, a 10-byte memory region, `r1`, is first allocated from the heap, and its start address, a, is stored into the pointer x. The shadow values of this memory region list this start address and the end address of the region, $a+9$. The pointer x also has a shadow value indicating that x is associated with `r1`. The pointer x is next used in an arithmetic operation that stores the value $a+15$ into another pointer, y. Because x was used as the source value for the operation, the shadow value associated with y receives the same region association as x, `r1`.

The pointer x is then changed to point to a newly associated 20-byte memory region, `r2`. This region begins at address b and has its own meta-data storing its start and end addresses. This also means that the shadow value for x now associates it with `r2`.

When the next instruction dereferences x, the analysis system checks that its value, b, is between b and $b+19$, the start and end addresses of `r2`. This address is within the region, so the dereference is considered safe. The last instruction attempts to dereference y, leading

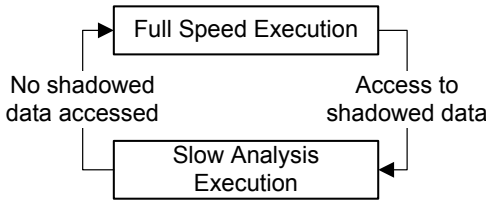


Figure 2.3 Traditional Demand-Driven Analysis. This technique attempts to execute the program normally and enables slow analysis only when shadowed data is encountered.

the analysis system to compare its value, $a+15$, to the limits of its associated region, `r1`. Because the end of *r1* is at $a+9$, the dereference is not safe and an error is raised.

Heap bounds checking can, as shown, be used to verify that accesses to memory regions are correct, a commonly desired security feature in type-unsafe languages [165]. A number of other powerful tools also use dynamic dataflow analyses.

Taint analyses, for instance, associate simple shadow values, or taint bits, with memory locations and propagate them throughout the program. One of the most commonly studied taint analyses is control flow attack detection, which marks memory locations as untrusted when their values originate from unsafe sources [170, 184, 210].

Other examples of this type of analysis include memory corruption detection [38], confidentiality verification [45, 133], malware analysis [231], and error tracing [11]. There is contention in the literature about the efficacy of some types of taint analysis [54, 206, 207], but we note that this does not affect the concept of taint analysis in general. As Schwartz *et al.* point out, the differences in these analyses are the particular semantics of their tainting policies and the desired security gains [197].

Strong dataflow security analyses will sometimes check implicit flows of information which arise because particular control flow paths can yield information about underlying data. The accuracy of taint analyses, for example, can be undermined in certain situations if this information is ignored [33]. However, following such information at runtime requires either costly static analyses before execution [108] or significant and slow extra instrumentation [218]. In either case, this chapter does not focus on sampling such control flow analyses because of the significant difficulties in building the underlying analysis itself.

As previously discussed, the power of dynamic dataflow analyses can also be undermined by the overheads they introduce. Complicated tools such as taint analyses can result in overheads beyond $100\times$ [93], which present a twofold problem: they limit adoption, but more insidiously, they significantly reduce the number of dynamic situations that can be observed within a reasonable amount of time. Because dynamic analysis tools benefit from observing multiple and varied runtime situations, performance overheads have a strong impact on their ability to find errors.

2.2.2 Demand-Driven Dataflow Analysis

Demand-driven analysis is a method to mitigate these performance issues. In this technique, software that is not manipulating meta-data is not analyzed and thus runs much faster. Figure 2.3 illustrates this. The software begins by executing at full speed, with no analyses taking place. However, when it encounters shadowed data[1], the process switches to an alternate mode of execution which also performs the complete (but slow) dataflow analysis. If the analysis system later finds that it is no longer operating on shadowed data, the process is transitioned back to executing with no analysis. Demand-driven analysis leads to higher performance when a program rarely operates on shadowed data.

To avoid as much analysis as possible, some authors have dedicated significant effort to contain the amount of shadowed data in their demand-driven analysis systems. This sometimes leads to executing weaker checks in exchange for performance. For example, in the work by Ho *et al.*, tainted data is only propagated through instructions that have one source operand [93]. Thus, arithmetic and logical operations, which have two operands, produce results with cleared taint bits, regardless of their inputs. This causes a potentially large vulnerability in the protection system. Similar works use lower quality security checks to limit performance and system complexity overhead as well [37]. We observed this trade-off upon adding propagation rules for multi-source instructions into a demand-driven taint analysis system, whereupon it slowed down by 25–35%.

Additionally, even with these optimizations, demand-driven analysis cannot provide performance guarantees. Processes may continually operate within the slow analysis tool if they frequently access shadowed data, yielding no performance improvement. This can be the case for certain worst-case inputs, as Ho, *et al.* demonstrated for their demand-driven taint analysis system [93], or for tools in which nearly all data is shadowed, such as bounds-checkers.

2.2.3 The Deficiencies of Code-Based Sampling

The code-based sampling systems previously discussed, such as assertion checking and concurrency bug testing, analyze a percentage of the instructions within a program's dynamic execution; the probability of performing analysis on any particular instruction is the only parameter available to control performance overhead. Figure 2.4a illustrates one negative effect this may have on dynamic dataflow analysis systems: this example uses the same bounds checking analysis shown in Figure 2.2, except that the final check of the variable *y*

[1]Meta-data itself is referred to as *shadow data*. Variables that have associated meta-data are called *shadow**ed** data*.

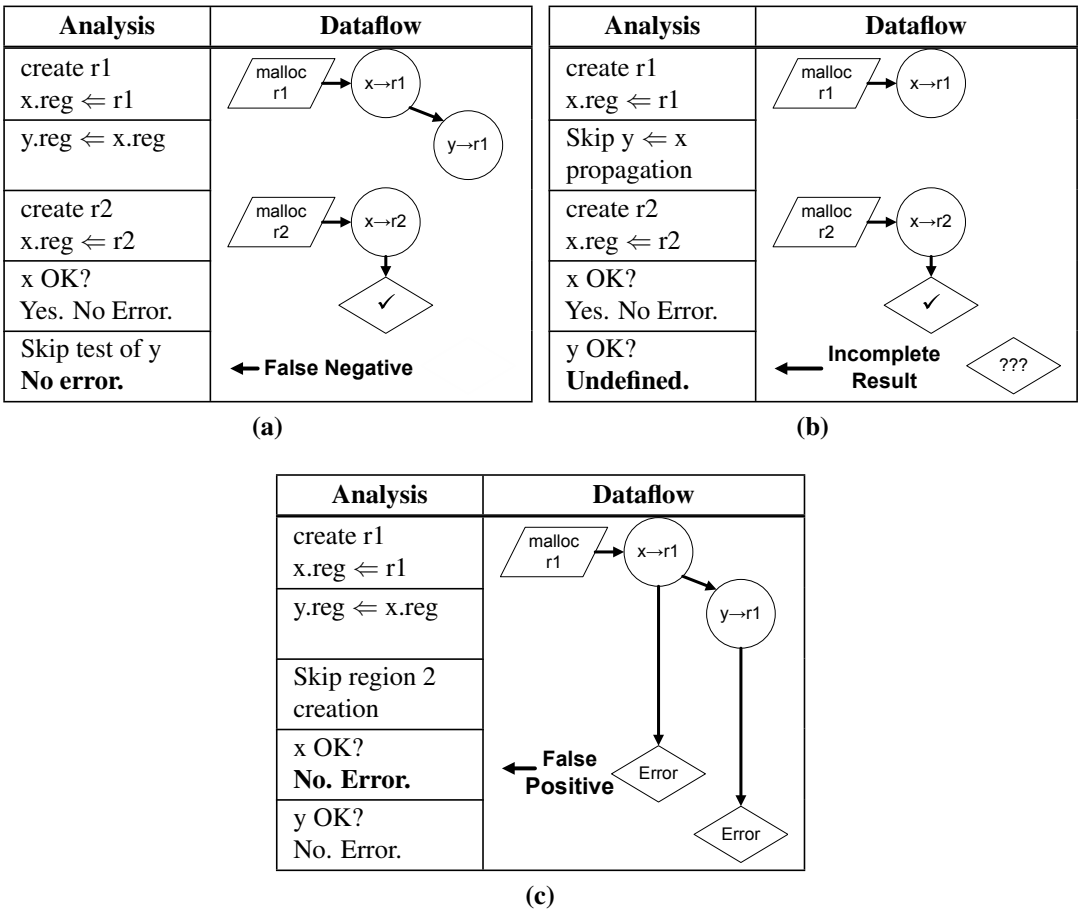


Figure 2.4 Code-Based Sampling Fails for Dataflow Analysis. **(a)** Skipping checks may lead to false negatives. **(b)** Sampling instructions within a dataflow analysis may cause false negatives later in the program due to dataflow changes. **(c)** Worse still, instruction sampling may modify a dataflow in ways that cause false positives.

is skipped when the sampling system attempts to reduce the performance overhead. This results in false negatives, an effect inherent to any sampling system.

While code-based sampling is effective for the previously mentioned tests, it is both inefficient and incorrect for dataflow analyses. By skipping the analysis of dynamic instructions without concern for their shadow values, it is possible (in fact, likely) that the shadow dataflows will differ from the dataflows of the associated variables. These incorrect shadow dataflows may cause the system to catch fewer errors, such as when shadow values indicating erroneous states are never written. This is illustrated in Figure 2.4b, where skipping an earlier shadow propagation results in the system missing the boundary error later in the program.

This problem will compound if the dataflow leading to an error goes through multiple instructions and the probability of analyzing any individual instruction is not high. In the

simplest case, the probability that the dataflow has not been corrupted by skipping a shadow operation is described by the geometric distribution $(1-p)\cdot p^k$, where p is the probability of analyzing any individual instruction and k is the number of instructions that operate on the dataflow until the error. This means that it becomes increasingly likely to corrupt the shadow dataflow as more instructions are analyzed.

An even more worrisome case occurs when an out-of-date shadow value leads to false positives (i.e., reporting errors that do not exist). Figure 2.4c demonstrates this scenario. If analysis is disabled during an operation that should change a variable's region association, the corresponding meta-data remains unchanged. When analysis is later re-enabled, the bounds checker reports an error because the actual pointer addresses a different memory region than the meta-data. ***Code-based sampling techniques often skip meta-data manipulations because they selectively enable instrumentation on instructions. This leads to dangerous false positive and undesirable false negatives.*** A sampling system for dataflow analyses must therefore always be aware of the shadow dataflows on which it operates.

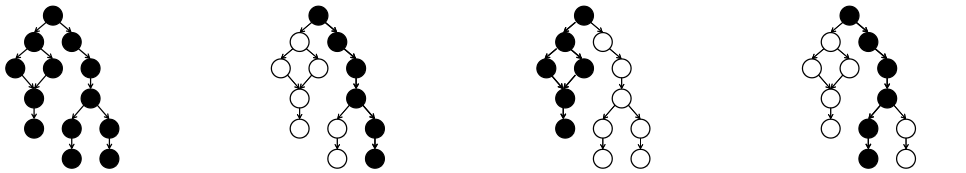
The naïve method of solving this problem is to simply remove the shadow variable associated with the destination of any instruction whose analysis is skipped. However, this would require every memory instruction to attempt to clear shadow variables when analysis is disabled – they *could* be unknowingly touching a shadowed location. This would lead to an unacceptable performance loss, as most programs contain a large number of memory operations, even if they rarely operate on shadowed data.

2.3 Effectively Sampling Dataflow Analyses

To remedy the deficiencies of code-based sampling techniques, this chapter presents a method for sampling dataflows instead of instructions. Rather than disabling analysis for instructions that may modify shadowed data, this technique instead skips the analysis of entire shadow dataflows.

As a program runs, a dynamic analysis system can create numerous shadow dataflows. A deterministic program that is run multiple times with the same inputs will see the same outputs and the same shadow dataflows. Similarly, a program run multiple times with different inputs may have dataflows that repeat. In these cases, there is no need to analyze all dataflows every time to find an error. If a dataflow that leads to an error is analyzed once, we can report the problem and skip the analysis of this dataflow in future runs.

However, because there is no way to know ahead of time where errors reside (knowing this would make analysis superfluous), it is not possible to decide ahead of time to skip



(a) Static Code **(b)** Dynamic Run 1 **(c)** Dynamic Run 2 **(d)** Dynamic Run 3

Figure 2.5 Uncoordinated Distributed Dataflow Analysis. Instead of enabling analysis for random instructions, dataflow sampling must attempt to follow entire shadow dataflows. With enough random dataflow subsets, the population will analyze the entire shadow dataflow.

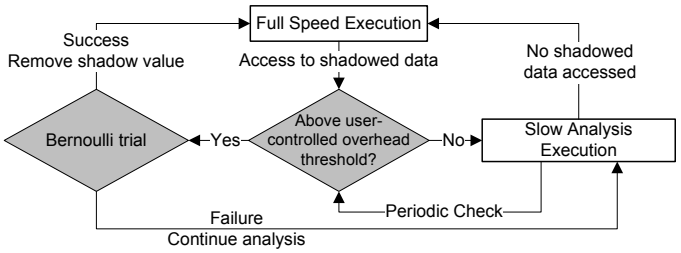


Figure 2.6 Dynamic Dataflow Sampling. This system samples a program's dataflow by stochastically removing shadow values after passing a performance threshold. When combined with demand-driven analysis, this enables dataflow, rather than instruction, sampling.

dataflows that do not lead to errors. Additionally, a scalable distributed system should not require all nodes to communicate with one another, so we also make the assumption that multiple users communicate neither with one another, nor across multiple executions. This means that this sampling system cannot coordinate between executions to determine which dataflows to analyze. It instead randomly selects some dataflows and ignores all others. During the next execution (or for the next user), a different set of dataflows may be analyzed. If the dataflows chosen for analysis are appropriately random, then enough users will, in aggregate, find all observable errors. This is demonstrated in Figure 2.5.

The benefit of only observing a subset of the total dataflows is that the analysis system causes less overhead as fewer dataflows are tested. In fact, we can take advantage of this to allow users to control the analysis overheads they experience. By setting the number of observed dataflows, the user can indirectly control slowdowns. Similarly, the system can directly cap performance losses by removing dataflows after crossing a performance threshold. The instructions that operate on these removed dataflows can subsequently execute without requiring slow analysis.

Figure 2.6 illustrates the operation of this dataflow sampling system. Like a demand analysis system, it disables analysis when it is not operating on shadowed data and activates the analysis infrastructure whenever such data are encountered. However, if the tool's performance overhead crosses a user-defined threshold, it begins to probabilistically discard

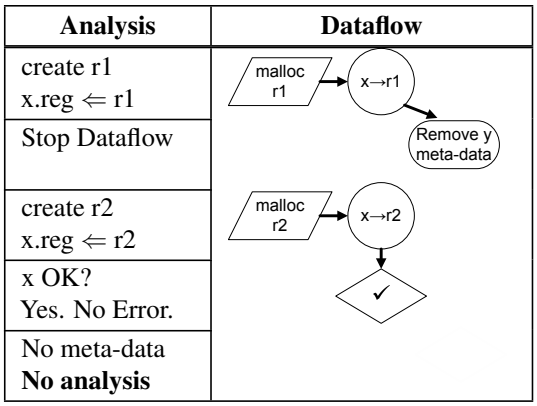


Figure 2.7 Dataflow Sampling May Also Miss Errors. False negatives are part and parcel of sampling analyses. The data that causes the error may be created when analysis is disabled, for instance.

the shadow data that it encounters. In a taint analysis system, for instance, it will randomly untaint some untrusted variables, implicitly marking them as trusted. The removed meta-data should be chosen in some stochastic manner in order to guarantee that different dataflows are observed during each program execution. This is modeled as a Bernoulli trial.

At this point, the demand-driven analysis system allows native execution to resume, as it is no longer operating on shadowed variables. Eventually, as the program encounters fewer shadow values, it will spend less time in analysis and will begin executing more efficiently. The observed slowdown will eventually go below a threshold, and the sampling system can stop removing shadow variables. This may lead to further slowdowns, restarting the stochastic meta-data removal process. Alternately, the system may simply continue to operating in a demand-driven analysis fashion.

As with other sampling approaches, these performance improvements come at the cost of potentially missing some dynamic events, as illustrated in Figure 2.7. However, when we remove shadow data-flows (rather than instrumentation associated with instructions), we not only reduce the amount of slowdown the user will experience in the future, but also maintain the integrity of the remaining dataflows, thereby preventing false positives. As the example in Figure 2.8 shows, eliminating meta-data removes an entire dataflow, which leads to reduced overheads. The remaining meta-data is still propagated and checked as before, thereby eliminating the false positives that were possible with the code-based sampling of Figure 2.4c. **This dataflow sampling technique analyzes samples of dynamic dataflows, rather than dynamic instructions. In doing so, it yields performance improvements without introducing false positives.**

Besides false positives, sampling systems must deal with false negatives (i.e. missing a real error). Although all sampling systems result in some missed errors, an increased

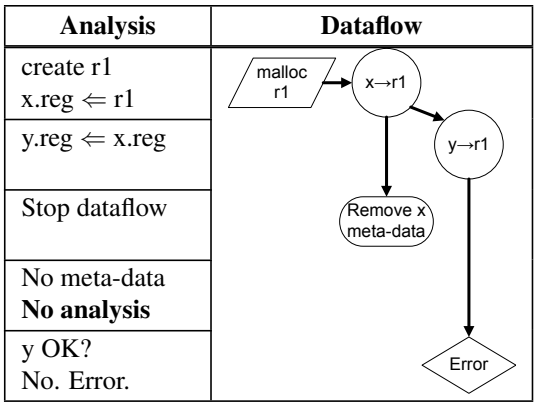


Figure 2.8 Dataflow Sampling Prevents False Positives. Eliminating a shadow value removes the entire dataflow that depends on it. No false positives occur because all propagations and checks related to that data are skipped.

population of testers can help offset this. Because individual users experience much lower overheads, the total population of users willing to run the tests can increase. This may offset the errors missed due to the false negatives caused by sampling, though this dissertation does not further speculate about the number of extra users this technique would deliver. Instead, this work focuses on maintaining a reasonably low false negative rate.

The method of choosing which shadow values to remove should strive to observe many different dataflows across each execution so that numerous uncoordinated users can analyze varied parts of the program. Ideally, each execution should observe as many dataflows as possible, though this may lead to overly high slowdowns that cause the sampling system to engage. Because small dataflows are likely to be completely observed by many of the numerous disparate analysis systems, it may be beneficial to prioritize the more-difficult-to-analyze large dataflows. This chapter begins by discarding values in a uniform random manner, but will look at more complicated methods in Section 2.3.3.

2.3.1 Definition of Performance Overhead

It is useful to precisely define the concept of overhead used in this chapter before detailing the sampling system's operation. Performance overhead is commonly defined as the extra time taken to complete the program's execution with analysis versus the time required without. For example, if a function takes one second to execute normally, but six seconds when under analysis, the analysis is causing a 500% overhead. This computation is straightforward after the execution has completed, but challenging to perform, or even estimate, while the program is still running.

This chapter instead uses on a slightly different definition: overhead is the amount of time spent performing analysis compared to the overall computation time. For instance, if a program completes in six seconds, and five of those seconds are spent in analysis, then the overhead is $5/6 = 83\%$. Note that were analysis disabled, the program may have required less than the one second of computation. Effects such as extra interrupts and cache misses may cause additional overhead beyond the time spent in analysis. Since observing these effects would require running the program twice, both with and without analysis, this definition instead allows tracking overhead costs in a simpler, dynamic way.

Based on this definition, dynamic overhead values can be estimated as $(cycles\_analysis)/(cycles\_analysis + cycles\_native)$. Overheads can range from 0 (no analysis, full speed) to 1 (always in analysis), inclusive. Though this does not tell us the actual slowdown experienced by the system, we can pessimistically estimate the overhead experienced by the user by assuming that the analysis system makes no forward progress. In actuality, the performance can vary greatly, with the analysis systems used in for this chapter causing an average slowdown of 150×. The no-forward-progress estimate, however, allows a quick, relatively accurate estimate of dynamic slowdown. If the overhead is measured to be 0.95, the actual slowdown (assuming the analysis system slows performance by 150×) is 17.8×. Our estimate yields a pessimistic, but close, value of 20×, assuming that only 5 out of every 100 cycles make any forward progress.

2.3.2 Overhead Control

At the core of this sampling system is the user's ability to set the maximum performance overhead they are willing to experience. Once it has been reached, analysis should stop, native computation should resume, and performance will improve. To accomplish this, users are allowed to set an overhead threshold that is of the same form as the above overhead calculations: the maximum percent of time (over some past period) that the system should be in analysis rather than native computation. Both the threshold and the period of time over which to calculate overheads are user-controllable. As shown in Figure 2.6, the sampling system checks the current measured overhead against the user's threshold both when the system prepares to enter analysis and at regular intervals during analysis. This ensures that the performance overhead does not greatly exceed the user's threshold before returning to fast execution.

A simple method of overhead control is to deterministically stop the analysis every time the overhead exceeds the user's threshold. This gives the user tight overhead control. However, if every analysis were interrupted immediately after crossing the threshold, it

could become impossible to observe errors that occur deep in the program's execution. This chapter instead explores a method that allows for the possibility of continuing an analysis even after the crossing the overhead threshold: individual attempts to interrupt the analysis are modeled by Bernoulli trial. If the probability of stopping the analysis is constant for every check, the total number of attempts needed before the tool is disabled can be modeled by a geometric distribution.

We start by defining the probability of exiting analysis after the system crosses the overhead threshold as a constant value, P_{st}. In other words, every attempt to stop the analysis succeeds with probability P_{st}. In this case, the number of attempts until the analysis ends can be modeled with a geometric distribution, and $G(k)$, the number of users that will continue to analyze the program after k attempts to stop, can be described as:

$$G(k) = 1 - CDF\left(geom(k)\right) \text{ where } CDF\left(geom(k)\right) = 1 - (1 - P_{st})^k$$

This simplifies to: $G(k) = (1 - P_{st})^k$

The mean slowdown seen by this system once it crosses the overhead threshold is also related to the geometric distribution. The following parameters are needed to describe the overhead:

- Executing the software and analysis together is $Q\times$ slower than execution of the software alone.
- The number of basic blocks that are analyzed between each attempt to stop analysis is N_{bb}
- The average number of instructions in a basic block is I.

Because $G(k)$ is a geometric distribution, the mean number of attempts to stop the analysis before succeeding is $1/P_{st}$. Every time an attempt to stop analysis fails, N_{bb} basic blocks are executed, each containing an average of I instructions. Each instruction runs $Q\times$ slower than if analysis were disabled. Therefore, the mean number of extra instructions observed after crossing the overhead threshold is:

$$\mu_{slowdown} = \left(\frac{1}{P_{st}} - 1\right) \cdot (N_{bb} \cdot I \cdot (Q - 1))$$

With a 5% probability of stopping the analysis, the expected number of trials before the analysis is disabled is 20. A user will attempt to stop an average of 20 times before successfully moving back to executing at full speed. If $I = 5$ [178], $N_{bb} = 10,000$, and

$Q = 150$, then $141,500,000$ extra instructions will be executed. If the window in which overheads are checked is large (say 10 seconds), then those extra instructions are very lightweight ($< 1\%$ on a 2 GHz system).

Using the same reasoning, nearly six out of every 1000 users will still be analyzing after 100 attempts ($0.95^{100} = 0.59\%$). These users provide the opportunity to find deep and difficult errors during execution, while the average overhead perceived by all users is still close to the threshold.

Setting a constant P_{st} allows users to control the mean overhead they observe and enables us to temporarily slow down a small fraction of users for a chance to analyze the program at a greater depth. However, the probability of analyzing any particular part of the program after reaching the overhead threshold is also directly related to P_{st}, and thus the average overhead.

It may be difficult to analyze some parts of a program's dataflow, even if P_{st} is set very low. Extremely long dataflows of dependent data, where stopping the analysis at any point will cause subsequent analysis opportunities to be missed, may not be completely analyzable in a sampling system. Because of this, there may be a fundamental tradeoff between the probability of finding some of these difficult errors and the ability to perform sampling in a dataflow analysis tool. This is not explored further in this work. It is worth noting, however, that it may be possible to analyze these dataflows offline. If analysis is forcibly stopped on a difficult-to-analyze dataflow, information could be sent back to the developer for use in a full, non-sampling analysis run offline. In a sense, the online dataflow analysis sampling would act as a filter for these offline analyses, allowing developers to only focus on potentially useful inputs.

2.3.3 Variable Probability of Stopping Analysis

Users may desire still-finer control over both the average overhead and the probability of participating in long analyses. To this end, it is possible to make the probability of stopping the analysis a function of the number of previously failed attempts to stop analysis, rather than a constant value. For example, a user may set the probability to stop the analysis at 5% once the overhead threshold is crossed, with a linear decrease down to 1% over the next 100 attempts. This allows most users to see an overhead as if they had approximately a 5% chance to stop, but a few executions still analyze a much larger fraction of the program, close to that achievable at a 1% probability.

Of course, the efficacy of such a scheme depends on the user's desired outcome. The user may still be unhappy if the few executions that are "chosen" to continue the analysis

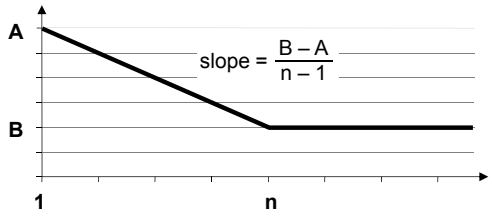


Figure 2.9 Variable Probability of Stopping Analysis. Rather than having a constant chance of leaving the analysis after crossing the overhead threshold, this system allows the probability to linearly decrease from $A\%$ to $B\%$ over the first n attempts to stop. Appropriately setting these values allows a user to set both a desired mean overhead as well as the average number of users who analyze deeply into a program.

spend too long running slowly. Long periods of slow execution effectively decrease the responsiveness of interactive applications, which has been shown to increase user dissatisfaction. Once user-perceived latency crosses expected thresholds (often of a few seconds), dissatisfaction sharply rises [70, 112].

In order to stay below such a threshold, this sampling system may require some sort of overhead watchdog timer. If the timer fires, the analysis could be forcibly disabled and the system state could be sent back to the developer for offline analysis.

Nonetheless, this section shows how using a non-constant probability of stopping analysis allows control over the mean overhead with separate control over the amount of the program some random individuals analyze. The latter case means that it is possible to control the probability that any individual stays in an analysis long after crossing the overhead threshold. The running example in this section, as seen in Figure 2.9, is a function where the probability of stopping the analysis linearly decreases from a value A to a value B across the first n checks. After the n^{th} check, the probability remains constant at B.

In this case, the probability of a user still analyzing the software after the first attempt to leave, immediately after crossing the threshold, is $1-A$. A user that continues will then execute N_{bb} basic blocks before again attempting to disable analysis. The probability that a user continues to run the analysis for $2 \cdot N_{bb}$ basic blocks is $(1-A) \cdot \left(1 - \left\{A + \frac{B-A}{n-1}\right\}\right)$. The probability of a user continuing analysis for $N_{bb} \cdot k$ basic blocks after crossing the overhead threshold, where $0 < k \leq n$, is therefore:

$$F(k) \quad = \quad \prod_{x=0}^{k-1} \left[1 - \left\{ A + \frac{x \cdot (B-A)}{n-1} \right\} \right]$$

Once $k > n$, the probability of stopping any individual analysis is constant, so the equations from Section 2.3.2 apply. Therefore, the probability that a user continues to analyze

the program after $N_{bb} \cdot k$ basics blocks, if $k > n$, is:

$$F(k) = \left(\prod_{x=0}^{n-1} \left[1 - \left\{ A + \frac{x \cdot (B-A)}{n-1} \right\} \right] \right) \cdot (1-B)^{(k-n)}$$

The mean number of attempts to stop can be computed as follows: 100% of users perform at least one check, $(1-A)$ perform two checks, $(1-A) \cdot \left(1 - \left[A + \frac{B-A}{n-1}\right]\right)$ compute three checks, etc. We can calculate the mean number of attempts by first calculating the estimated mean, $\widetilde{u}$, for terms up to n.

$$\widetilde{\mu}_{k<n} = 1 + \sum_{k=0}^{n-1} F(k); \text{ for } k < n$$

Because n is finite, this is some constant number whose value depends on $F(k)$.

When $k \geq n$, each probability of stopping analysis is a constant B. Therefore, the probability to continue analysis after n attempts is $F(n)$; after $n+1$ attempts it is $F(n) \cdot (1-B)$; after $n+1$ attempts it is $F(n) \cdot (1-B)^2$; etc. In general, the probability is $F(n) \cdot (1-B)^{k-n}$.

Consequently, as k increases from n to ∞:

$$\begin{aligned} \widetilde{u}_{k \geq n} &= F(n) \cdot \left(1 + (1-B) + (1-B)^2 + ...\right) \\ &= F(n) \cdot \left(\sum_{k=n}^{\infty} (1-B)^{(k-(n))} \right) \end{aligned}$$

Because the first term, $k = n$, yields $(1-B)^0$, the second $(1-B)^1$, etc., and this is an infinite series, it is possible to replace this function with:

$$\widetilde{u}_{k \geq n} = F(n) \cdot \left(\sum_{k=0}^{\infty} (1-B)^k \right)$$

Noting that this is a geometric distribution with probability B, we can show that the mean number of checks after $k > n$ is $\frac{1}{B}$.

Because this sum converges, and the previous function is a constant, we can say that the mean number of attempts to stop the analysis is:

$$\mu_{attempts} = 1 + \sum_{k=0}^{n-1} [F(k)] + F(n) \cdot \frac{1}{B}$$

This mean number of attempts can then be used to find the mean slowdown associated with a system that uses a functional probability to stop analysis:

$$\begin{aligned}\mu_{extrainstructions} &= \mu_{attempts} \cdot N_{bb} \cdot I \cdot (Q-1) \\ &= \left(1 + \sum_{k=0}^{n-1} [F(k)] + F(n) \cdot \frac{1}{B}\right) \cdot N_{bb} \cdot I \cdot (Q-1)\end{aligned}$$

If $A = P_{st}$ and $B > A$, then this will have a higher mean slowdown than the constant-probability case. However, intuitively, most users will stop early due to the higher probability A, while a small number of users will continue the analysis, using the lower probability B to attempt to leave. By setting A higher than P_{st}, B lower, and appropriate setting n, it is possible to yield a mean overhead that is in line with the constant probability case, but with a higher probability of analyzing later portions of the program.

2.4 Prototype System Implementation

A prototype dataflow sampling system was built on top of a demand-driven taint analysis system that uses the Xen virtual machine monitor [15] and the emulator QEMU [20]. Entire virtual machines, called domU domains in Xen, are monitored in this design. Data that enter the virtual machine from network or keyboard inputs are marked as tainted, a status that is stored in a binary shadow variable associated with each register and every byte in physical memory. The virtualization system allows user-level applications within the domain to run at full speed on the unmodified hardware when they are operating on untainted data.

The hypervisor marks pages that contain shadowed data as unavailable, causing the system to take a page fault whenever instructions attempt to access them. The hypervisor's page fault handler then checks the page number against a list of pages that contain shadowed values. If the process is attempting to access one of these pages, the entire state space of the domU domain is transferred into an instance of the modified version of the QEMU x86 emulator running within the dom0 administrative domain. The domU domain then begins executing within the emulator, which also performs taint analysis by propagating shadow values through copy and arithmetic operations. The emulator also checks the taint status of inputs to instructions such as dereferences and control flow operations and raise an error if these inputs are tainted.

When the emulator is no longer operating on tainted values, the domain's state is transferred back to the native hardware. Further information about this demand-driven taint

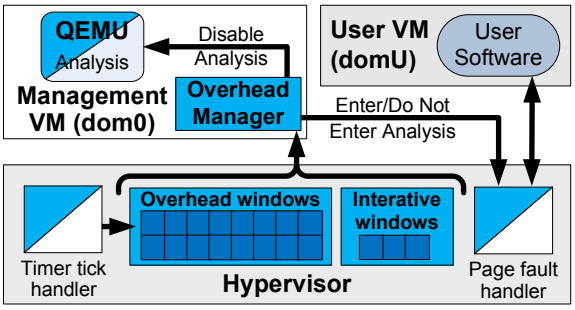


Figure 2.10 Prototype Dataflow Sampling System. This system makes a number of extensions to Xen. The user software executes in a VM during normal operation and in emulation within the management virtual machine during analysis. The overhead manager monitors the amount of time spent in emulation and can force the system to untaint values, forcibly ending shadow dataflows. The interactive windows track a recent subset of the time in analysis for use when a user is interacting with the system.

analysis system, including its method for detecting tainted values and the overheads incurred by false sharing in the page table, can be found in the demand emulation paper by Ho *et al.* [93]. This section describes the modifications needed to build a dataflow sampling system on top of this framework, as shown in Figure 2.10.

This overhead calculation, as defined in Section 2.3.1, is done using a rolling window. The length of this window is user-defined, and each element tracks the number of cycles spent within QEMU during the last tick of the 100Hz system clock. The cycles stored in each element are updated both when a domain or QEMU is scheduled and on clock ticks. Finally, on every clock tick, the oldest element is dropped from the window. This allows us to efficiently calculate the overhead for the last $window\_length \cdot clock\_tick\_length$ period of time. The primary extensions to the baseline system to support this overhead tracking method include the addition of timer windows into the hypervisor and dedicated timekeeping code in the timer interrupt handler and the scheduler code of dom0 and the hypervisor.

This system also includes a program that allows an administrator in the dom0 domain to set the overhead threshold for any domU. Similar to the overhead manager (OHM) described in QVM [10], this system then uses a daemon running in dom0 to check each domain's overhead against its threshold. If the overhead is higher than this limit, the OHM marks the domain as "above threshold" and sets a probability for untainting active data, effectively resuming native execution. This is the Bernoulli probability from Figure 2.6, and it can also be changed by the administrative program in dom0.

QEMU periodically checks the overhead-related variables stored by the OHM, and it probabilistically decides to stop the analysis based on the stored Bernoulli probability if the domain is "above threshold". If the emulator determines that it must stop, it clears all

registers of their shadow values and returns control back to the hypervisor, where it resumes running normally.

If QEMU were to check the overhead variables stored in the hypervisor at the end of every basic block's execution (the smallest atomic unit with which QEMU works), the amount of time waiting on the hypercalls would far outweigh the amount of time actually performing dynamic analysis inside emulation. Thus, to balance their impact, QEMU instead executes a user-controllable number of basic blocks between each check. As will be shown in the experiments, this value and the probability of leaving QEMU interact with each other, affecting the overall performance of a system once an application's analysis impact goes beyond its overhead threshold.

While leaving the emulator clears the shadow values associated with the data in the registers, the system must also be able to clear meta-data associated with memory locations. Therefore, the page fault handler also checks the values set by the OHM before moving a domain into QEMU. If the overhead is beyond the desired threshold, the handler probabilistically decides whether to enter the emulation or skip the analysis. If the analysis is skipped, the shadow tags associated with all data on the page are cleared, the page table entry is marked as available, and execution continues natively at full speed.

2.4.1 Enabling Efficient User Interaction

The goal of this overhead management system is to ensure that overall system performance does not fall below a level that the user would find disruptive. While the current design can guarantee the average performance of the system, long intervals of low performance, even if followed by periods of full-speed execution, can still result in a negative user experience. There is a long history of research showing that for many users, responsiveness (not raw performance) is the most important factor in deciding their satisfaction with the system [188]. This means that users are more willing to tolerate slowdowns in non-interactive applications, since the effect is less noticeable [83, 217].

The ideal amount of continuous time to spend analyzing (and thus slowing down) interactive applications is somewhat contentious. Many users would claim that *no* slowdown is acceptable, as they want the fastest computers possible. Older literature on human-computer interaction, however, finds that users can tolerate complete system pauses for up to one second before they begin to feel uncomfortable [141]. This exact value may be out-of-date with current performance expectations, but Miller brings up the point that numerous factors (such as the type of application, the expected performance, etc.) work together to make some amount of slowdown acceptable for the general population.

The user interface design book by Shneiderman *et al.* lays out the rule that, for interactive tasks, the system should now respond to user actions within 50 to 150 milliseconds [202, p. 427]. Tolia *et al.* agree with these numbers, claiming that response times below 150 ms are "crisp," while those above range from "noticeable to annoying" to "unusable" when dealing with interactive tasks [217]. Dabrowski and Munson, on the other hand, showed that these constraints can be relaxed for some methods of interaction. They found that most users did not notice keyboard latency until it reached 150 ms, nor did they notice mouse latency until it was above 195 ms [51]. Nonetheless, Martin and Corl showed that user productivity (though perhaps not happiness) rises linearly as response time improves – they found no magical cutoff point where latency is acceptable, though they only study systems with latencies above 100ms [132]. Using a slightly different way of examining the problem, Shye *et al.* directly looked at user's mood while changing the performance of the system. They used feedback from physiological monitors to prevent the user becoming angry at resource stealing [204].

Because the exact amount overhead users are willing to accept in interactive applications depends on so many factors, and because a feedback-based system requires extra hardware to monitor the user's mood, the sampling system in this chapter instead allows users to set their own maximum emulation intervals through the overhead manager. To improve system responsiveness, the sampling system provides a second, shorter window to evaluate overhead; this window is checked by the overhead manager only when a user is interacting with an application under analysis. This interactive window is also shown in Figure 2.10. This feature improves the application's interactivity by reducing the amount of continuous time spent in analysis.

Reducing this time may make it more difficult to observe errors deep in dataflows. However, as discussed at the end of Section 2.3.2, it may be possible to save the analysis information for offline analysis rather than risk disturbing the user and making the application less usable. It may also be possible to use multi-processor techniques to perform these analyses "offline" on the same system after crossing the interactive performance threshold [172]. This would allow the program to have good interactive performance while also potentially allowing difficult dataflows to be fully analyzed.

2.5 Experimental Evaluation

The virtual machines used on the experimental system run a modified version of Xen 3.0 and Linux with modified 2.6.12.6-xen0 and 2.6.12.6-xenU kernels. Tests were executed on a

```
loop
    value ⇐ shadowed_data
    for i = 0 to N do
        value ⇐ value + 1
    end for
end loop
```

Figure 2.11 Worst-Case Synthetic Benchmark. This program is designed to show the limits of our dataflow sampling system by continually operating on shadowed data. It will remain in emulation until the sampling system removes the shadow value.

```
declare buffer[N]
value ⇐ from_network
for i = 0 to N − 1 do
    buffer[i] ⇐ value
end for
buffer[N + 4] ⇐ value
```

(a) wide_function

```
declare buffer[N]
value ⇐ from_network
for i = 0 to N − 1 do
    value ⇐ value + 1
end for
buffer[N + 4] ⇐ value
```

(b) long_function

Figure 2.12 Synthetic Accuracy Benchmarks. The program in **(a)** copies a large amount of tainted data into many different locations, then uses the original value to cause an exploit. The program in **(b)** repeatedly copies a large amount of tainted data into the same location and uses that data to cause an exploit. It is impossible to observe the exploits if the sampling system ever skips an analysis step, as it would remove the tainted status of the value that causes the exploit.

1.8GHz AMD Opteron 144 with 1GB of RAM and a Broadcom BCM5703 gigabit Ethernet controller. For all experiments, 512MB of RAM were allocated to both dom0 and domU.

2.5.1 Benchmarks and Real-World Vulnerabilities

The simplest test, shown as pseudocode in Figure 2.11, is a synthetic benchmark that tests if this sampling framework controls overheads while performing intensive dataflow analyses. It takes in shadowed data and repeatedly performs computations on it, forming a single large dynamic dataflow. Without sampling, this program is constantly under analysis.

The second synthetic benchmark, *wide_function*, is a representation of a server that suffers from a basic buffer overflow exploit. Pseudocode for this program is shown in Figure 2.12a. This benchmark takes in data from the network and copies it to a large number of memory locations, creating a very wide dataflow rooted at the input value. The initial value from the network is used as the payload for a buffer overflow at the end of the large series of copies.

The third and final synthetic benchmark, *long_function*, is shown in Figure 2.12b. It is very similar to *wide_function*, but rather than copying tainted data into a large amount

Table 2.2 Security Benchmarks. This lists a series of applications with known security exploits which were used to test the accuracy of the sampling taint analysis system.

Benchmark Name	CVE Number (Error)	Error Description
Apache	CVE-2007-0774 [158]	A stack overflow in Apache Tomcat JK Connector v 1.2.20
Eggdrop	CVE-2007-2807 [159]	A stack overflow in the Eggdrop IRC bot v 1.6.18
Lynx	CVE-2005-3120 [155]	A stack overflow of the Lynx web browser v 2.8.6
ProFTPD	CVE-2006-6170 [156]	A heap smashing attack of ProFTPD Server v 1.3.0a
Squid	CVE-2002-0068 [153]	A heap smashing attack of the Squid proxy v 2.4.DEVEL4

memory, it instead performs numerous arithmetic operations on the data received from the network. This creates a very long dataflow, as each new meta-value relies on the previous. Afterwards, it uses this data as an exploit payload. The dataflow analysis tool will be unable to observe the exploit if the sampling system stops the analysis in either *long_function* or *wide_function* at any point. Doing so in *wide_function* would untaint the source value, meaning no analysis would occur when the buffer overflow happens. Doing so in *long_function* would break the long dataflow, rendering the rest of the calculations untainted. It is therefore possible use these benchmarks to test the maximum amount of analysis that this system can perform under any particular overhead threshold.

We also incorporated the *netcat* and *ssh* network throughput benchmarks used by Ho, *et al.* to more directly compare dataflow sampling to a demand-driven analysis system [93]. In their work, demand-driven taint analysis experienced orders-of-magnitude slowdowns for applications that operated on large network transfers. In *netcat_receive*, 1.9 GB of data is moved from an external machine using a simple TCP connection. Both *ssh_receive* and *ssh_transmit* are similar, but instead move 781MB over an encrypted connection. We skip *netcat_transmit* because the only shadowed data it accesses is the IP address response from a DNS query. Our dataflow sampling system quickly clears this shadow value and the benchmark maintains high throughput until its completion.

We also analyze a collection of network-based benchmarks that suffer from security exploits that allow remote code execution, as shown in Table 2.2. These programs help verify that dataflow sampling can still observe a sufficient fraction of the shadow dataflows in complex programs to find real-world errors even when sampling with low overhead thresholds. These programs were run within the sampling system and exploits were transmitted at random points in time in order to obtain high confidence in the error-finding capabilities of this prototype. These benchmarks focus on buffer overflows (that result in stack and heap smashing attacks) because the dataflow analysis utilized in this prototype is designed to

detect this class of errors. While different dataflow analysis systems may be better suited for other types of software errors, their specific design is orthogonal to this work on sampling mechanisms.

2.5.2 Controlling Dataflow Analysis Overheads

We start with performance measurement benchmarks to ensure that the observable overhead when using this sampling framework is both limited and controllable. We then perform a series of tests that demonstrate the relationship between analysis parameters and observed overhead.

Limiting Overheads in the Synthetic Benchmark. The first set of experiments uses the synthetic benchmark detailed in Figure 2.11 to plot the instantaneous runtime overhead of the taint analysis tool. The program executes its conditional loop normally for 30 seconds, leading to a computational throughput of about 1,800 million instructions per second (MIPS). Afterwards, shadowed data arrives from the network at a rate of one packet every five seconds. The content of the incoming packets is used for the computation within the loop, so the taint analysis system invokes the analysis mode. The sampling system for this experiment uses a 30 second long overhead window, with a 100% probability of leaving analysis after crossing the overhead threshold.

As soon as packets begin to arrive, the performance of the non-sampling system plummets to an average of 21.3 MIPS. The system never leaves emulation when sampling is disabled because nearly every operation following the arrival of the first packet is shadowed.

Figure 2.13a shows the performance of this system when sampling is enabled with an overhead threshold of 10%. While instantaneous performance still drops when the first packet arrives, the overhead manager periodically forces the taint value to be cleared, returning the performance to its original value. This pattern repeats itself every time the rolling overhead window drops the short interval of low performance, resulting in an average performance loss of 11%.

Figure 2.13b plots both average slowdown and size of the analyzed dataflow over a range of threshold values. This shows that sampling allows users to effectively control the amount of time spent in dynamic analysis, although it may limit the number of dataflows that can be completely analyzed in one execution.

Performance Impacts for Real Programs. In order to test the ability of this dataflow sampling technique to improve performance over the baseline demand-driven analysis

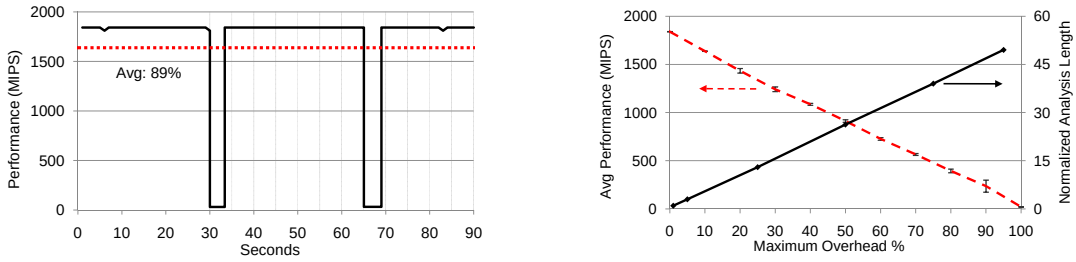


Figure 2.13 Synthetic Benchmarks: Performance and Analysis Accuracy. **(a)** Instantaneous (solid) and average (dashed) performance after the first untrusted packet arrives with a user-set overhead threshold of 10% in 30 seconds. **(b)** Average performance (dashed) and the maximum observable dataflow length normalized to a 1% overhead threshold (solid) for a range of overhead thresholds. Error bars show 95% confidence intervals.

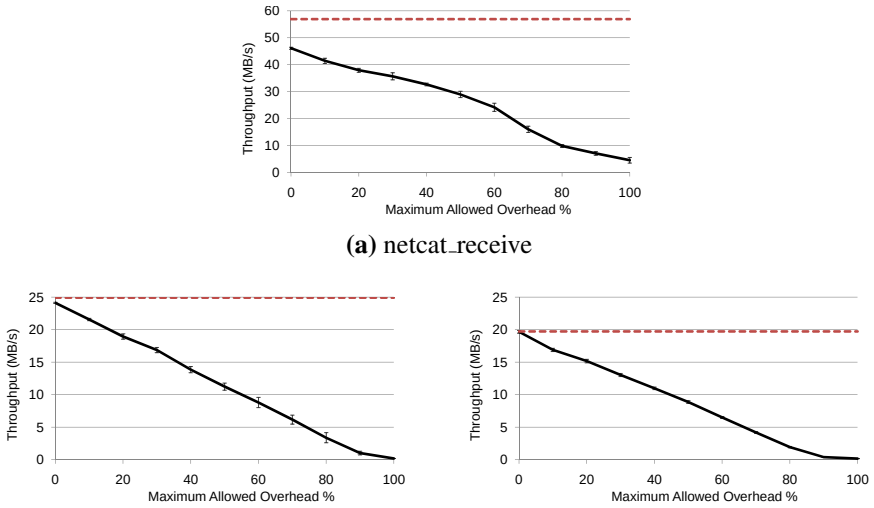
Table 2.3 Demand-Driven Analysis Network Throughput.The netcat benchmarks move 1.9GB of data either out of (transmit) or into (receive) the system under test through a simple TCP connection. The SSH benchmarks similarly move 781 MB of data through an encrypted connection. The data includes 95% confidence intervals.

netcat (MB/s)		
	Transmit (1.9 GB)	Receive (1.9 GB)
Native Execution	110.66 ± 0.06	59.90 ± 0.54
Demand-Driven Analysis	0.67 ± 0.04 $(165\times)$	4.6 ± 0.3 $(13\times)$
SSH (MB/s)		
	Transmit (781 MB)	Receive (781 MB)
Native Execution	24.91 ± 0.10	19.72 ± 0.05
Demand-Driven Analysis	0.19 ± 0.01 $(131\times)$	0.17 ± 0.01 $(116\times)$

system, the throughput of network-intensive benchmarks was measured under a number of circumstances. Because network data is the source of shadow values in this system, demand-driven analysis still causes the throughput (i.e. performance) of these programs to drop precipitously. Table 2.3 demonstrates the throughput differences between native execution and analysis within the purely demand-driven system. As an example of the extreme overheads seen when constantly working on tainted data, *ssh_transmit* suffers a $131\times$ slowdown on a non-sampled system.

These worst-case numbers are worse than those seen by Ho *et al.* because this emulator performs a more thorough analysis: it enables taint propagation through two-source instructions (e.g. addition and subtraction) which causes more, and slower, emulation [93].

Figure 2.14 shows the throughput of these network benchmarks with dataflow sampling. The X-axis in each chart represents the maximum overhead threshold. The probability of



(a) netcat_receive

(b) ssh_transmit

(c) ssh_receive

Figure 2.14 Dataflow Sampling Increases Network Throughput. The dashed red line represents performance with analysis disabled. Error bars represent 95% confidence intervals. **(a)** Netcat receive throughput increases nearly linearly as the maximum overhead threshold is lowered. The maximum performance is below that of a system with analysis disabled because of the page faults incurred before clearing shadow data. Both **(b)** SSH transmit and **(c)** SSH receive allow nearly linear control over their overheads.

leaving the analysis after crossing the threshold is set to 100%, meaning that this does not worry about the Bernoulli probabilities. The solid line plots average network throughput at that threshold, while the dashed red line represents the highest throughput each benchmark could reach if analysis were completely disabled. Note that an overhead threshold of 100% is the same as a demand-driven analysis system.

This data shows that dataflow sampling can greatly improve network throughput even for extremely slow analyses. Without sampling, for instance, transmitting data over *SSH* has an average throughput of only 171KB/s. This is 116× slower than the no-analysis case. However, as Figure 2.14b shows, we can directly control throughput by removing shadow values after the overhead threshold is reached, increasing throughput to nearly the full 19.7MB/s. The maximum throughput of *netcat_receive* when performing dataflow sampling is lower than the throughput attainable if no analysis is ever attempted. This is due to the page faults taken when touching tainted values, which are needed so that the sampling system can then decide to untaint the data.

2.5.3 Accuracy of Sampling Taint Analysis

The next important question is whether these performance increases significantly affect the ability of the analysis to detect errors. This section details accuracy tests for the dataflow sampling system on a number of programs in order to demonstrate that it can find exploits in real software. They verify that this design can effectively analyze these programs, even when the maximum overhead is low. Finally, because programs that do not directly contribute to an error still require analysis and limit the time our erroneous process can be under analysis, we run a second set of tests with significant amounts of spurious system load and demonstrate that our prototype can still detect errors when sampling is being used heavily.

Accuracy for a Lightly Loaded System. The five programs shown in Table 2.2 were run through the sampling system with an overhead threshold of 1% in a 10 second window and a 100% probability of removing shadow values after reaching the threshold. For parameters as unforgiving as these, the sampling taint analysis system was still able to find the security faults caused by the vulnerabilities in these benchmarks ***on every attempt***.

Though it is impossible to guarantee this is true for *all* security flaws, it is often true that that vulnerabilities that appear early in the dataflow of a program are easier to exploit. Exploit writers must build their inputs such that every operation modifies them to be in the correct form and place to attack systems. Intuitively, the fewer operations between the input and the final destination, the easier these inputs are to build. Errors that take place earlier in the dynamic dataflows are also easier to catch using sampling.

Accuracy for a Heavily Loaded System. The previous experiments focused on the analysis quality for benchmarks that were executed in isolation; the only dataflows were caused by exploits communicating with the vulnerable applications. Servers in real-world situations are typically heavily-utilized, with a resulting increase in the number of dataflows that the analysis system must check. Most dataflows in these high-utilization environments do not lead to errors, but still require time within the analysis tool. Because this sampling system only allows a limited amount of time in analysis before attempting to remove shadow dataflows, these extra analyses raise the probability of ignoring erroneous dataflows. This will lower the probability of finding an error. To quantify this difference, we studied the probability of observing an exploit while the rest of the system was heavily loaded with benign dataflows.

Two sets of benchmarks were combined to simulate such an environment. A network throughput benchmark (*netcat_receive* or *ssh_receive*) runs in the background to simulate a large amount of harmless operations that still require analysis, while one of the vulnerable

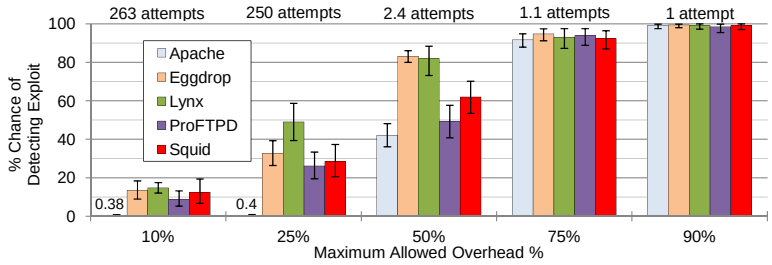


Figure 2.15 Exploit Observation with Netcat Background Execution. Each bar represents the probability of detecting a particular program exploit if the benchmark *netcat_receive* is constantly causing benign emulation in the background. The number above each set is the expected number of attempts (or users) needed to observe the most difficult error. Error bars are 95% confidence intervals.

security benchmarks (from Table 2.2) is exploited at some random point in time. Each benchmark was started by running the throughput program alone long enough to fill one overhead window. This removed any chance of observing the exploit solely because the overhead window is not filled with background operations. The exploit for the second program is then sent after a random time interval; it may arrive at any point within an overhead window.

All experiments were run with a 10 second overhead window and a 100% probability of leaving the analysis after crossing the overhead threshold. The 100% probability of stopping analysis means that the only dataflow randomization is caused by nondeterminism in the system and the random arrival time of the exploit. These are unforgiving settings that favor the user experience over analysis quality, as the short window and guaranteed halting of analysis yield short periods of time where any vulnerability can be observed and detected.

As previously detailed, this framework was able to consistently observe the exploits with an overhead threshold as low as 1% on the system with no benign dataflows. Figure 2.15 shows the probabilities of observing each benchmark's exploit while the *netcat_receive* benchmark runs in the background. Figure 2.16 plots the probability of observing each benchmark's exploit on a heavily loaded system where *ssh_receive* runs in the background. The error bars represent the 95% confidence interval for finding the error, while the number of attempts needed to observe the most challenging exploit is listed above each overhead threshold. This can be thought of as the expected number of users required to run the analysis before detecting the hardest error. Note that these attempts need not take place on the same computer.

In general, *netcat_receive* causes less background analysis and thus has a higher probability of finding the exploits. At high overhead thresholds, it is almost guaranteed to observe the exploit, while at a 10% overhead threshold, most of the exploits are still observed more

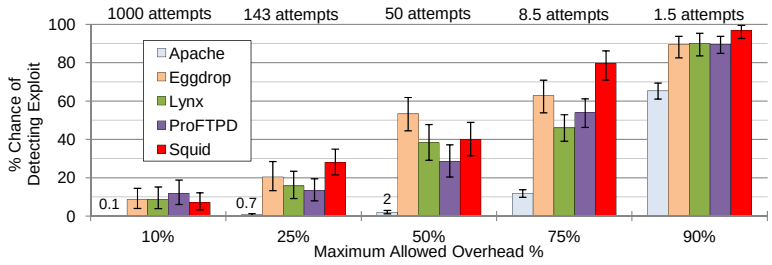


Figure 2.16 Exploit Observation with SSH Background Execution. Each bar represents the probability of detecting a particular program exploit if the benchmark *ssh_receive* is constantly causing background emulation. The number above each set is the expected number of attempts (or users) needed to observe the most difficult error. Error bars are 95% confidence intervals. Because *ssh_receive* spends more time in emulation than *netcat_receive*, it is more difficult to observe the exploits.

than 10% of the time. The extraneous dataflows caused by *ssh_receive* make it more difficult to observe the exploit than in the case with no background execution. Rather than observing every exploit, a small number of exploits are missed at high overhead thresholds. At a 10% overhead threshold, most of the exploits are still observed nearly 10% of the time.

The Apache exploit is more difficult to observe, however. This benchmark moves a large amount of shadowed data numerous times before the exploit occurs, making the dataflow leading to the exploit relatively long and surrounded by other data that, while tainted, do not actually take part in the exploit. This system still finds this flaw with a 0.1% probability at the lowest overhead threshold and most difficult background noise. This particular test shows that the dataflow sampling technique works well even with nearly every option set to particularly difficult choices. If *netcat_receive* were used instead of *ssh_receive*, for instance, the ability to observe the Apache exploit rises quite precipitously, as *ssh_receive* causes much more background analysis.

Exploring the Depth of Analysis Possible Besides analyzing the ability of system to find real-world exploits, it is also illuminating to examine how deeply programs can be analyzed as a function of maximum overhead. To test the ability to observe deeply-embedded flaws, we used the benchmarks *long_function* and *wide_function*.

The amount of data (or the amount of continuous operations) this system can analyze before enabling sampling is a function of the maximum desired overhead. It is still able to see up to 200,000 loops at a threshold of 1%; these loops result in millions of analysis operations. Our real-world benchmarks sit at, or below, this level of operations before the exploit. As Figure 2.17 shows, we can see millions of loop iterations even at low overhead thresholds using the sampling system. These tests were done with an overhead window

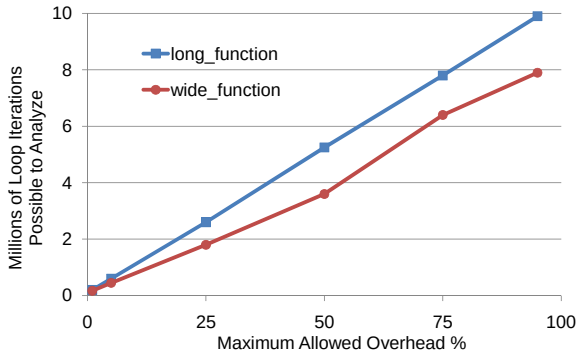


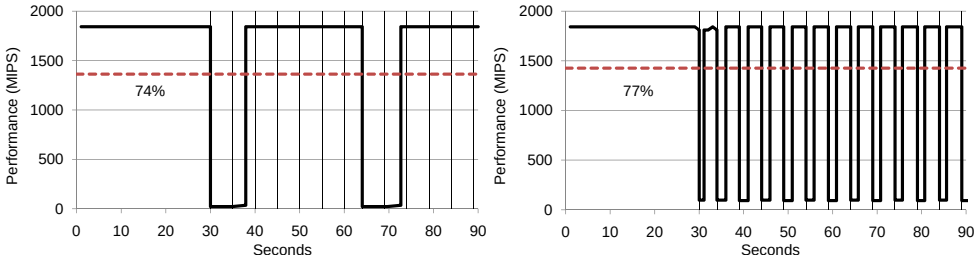
Figure 2.17 Maximum Analysis Possible Before the First Sampling Event. As maximum overhead threshold is increased, it is possible to analyze proportionally more operations in each program. At a 100% threshold, the exploit would be observable at any number of loop iterations because the analysis would never stop. However it is still possible to see millions of operations at a threshold as low at 1% in these worst-case benchmarks.

of 10 seconds. Were the window size increased, it would be possible to cover more loops before the overhead threshold was crossed. This small window was chosen, however, to show that it is still possible to analyze large programs with tight bounds on the analysis potential. If we lowered the probability of stopping analysis, some users would eventually see deep into any program.

2.5.4 Exploring Fine-Grained Performance Controls

Reducing User-Perceived Performance Impacts. Although the average performance of the system is useful when looking at throughput, (i.e. the total amount of computation accomplished) this metric does not take into account other important characteristics of a sampling system. As discussed in Section 2.4.1, maintaining responsiveness when interacting with users is a crucial part of making techniques such as sample-based analysis deployable to large populations. To maintain an acceptable level of responsiveness in these situations, the sampling framework was extended with a special mode of operation that detects interactions with the machine in an effort to balance the quality of analysis with the quality of the user's experience.

A set of experiments were then run to verify that this scheme stops the system from pausing (or going into slow emulation) for long stretches of time, while still performing a commensurate amount of aggregate analysis. The sampling system was modified to emit frequent bursts of interaction and while a series of tests were run using the first synthetic benchmark with a 25% overhead threshold with a 100% probability of stopping the analysis after crossing this threshold. For the non-interactive mode, the overhead window was 30



(a) Performance without interacting **(b)** Performance while interacting

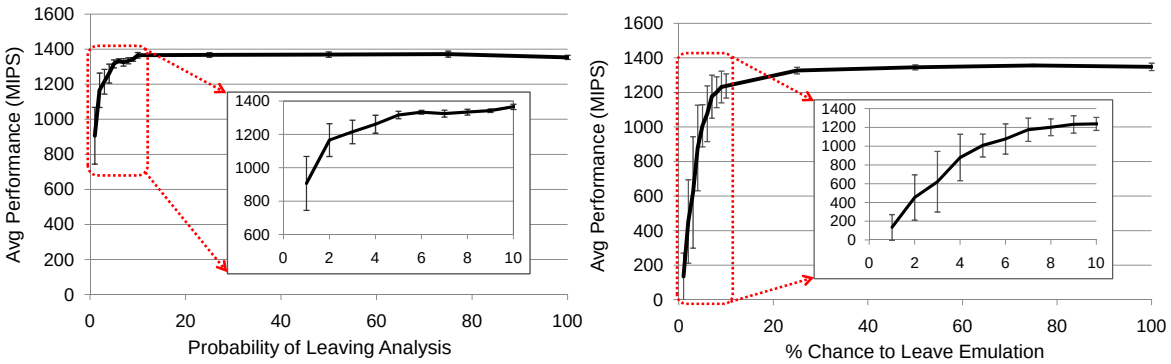
Figure 2.18 Changing the Interactive Performance. **(a)** A sampling system with a 25% threshold and a 30 second window. **(b)** Sampling system with a 25% threshold and a maximum of one continuous second in emulation due to user interaction. With the window size reduced for the cases when the user is interacting with the system, we are able to shorten pauses due to emulation and increase the responsiveness of the system, consequently improving the user experience.

seconds long. We wanted to avoid pauses longer than one second when interacting with the system, resulting in a four-second-long interactive window with a 25% overhead.

Figure 2.18 shows that moving to a shorter window when the system is in interactive mode results in shorter, but more frequent periods of slow analysis. Both tests average nearly 75% of the maximum performance, but the interactive case guarantees that the system does not pause for more than one second at a time. The disadvantage of this system is that the faster rate of sampling reduces the amount of analysis that can be performed on very large dataflows, potentially reducing the quality of the analysis on the program at that point.

Changing the Probability of Removing Dataflows. The previous experiments were all performed with P_{st}, the probability of stopping the dataflow analysis once the overhead threshold is exceeded, set to 100%. This choice can obviously affect the performance of the system and its ability to fully observe large dataflows. A lower P_{st} permits some analyses over the overhead threshold to continue before the shadow values are cleared and analysis is forcibly stopped. While this gives users less control over the exact slowdowns they perceive, it also allows some users in the population to observe more dataflows and analyze deeper into large dataflows. This boosts the chances of finding some errors.

Figure 2.19 shows the performance of the system running the first synthetic benchmark with a 25% overhead threshold for two ranges of sampling options. Figure 2.19a shows how performance is affected when sweeping the probability of stopping the analysis, P_{st}, with the number of basic blocks per check, N_{bb}, equal to 10,000. Setting P_{st} to 1% leads to a mean performance of 900 MIPS with a standard deviation of $\pm$170 MIPS. The mean is nearly the same as the performance attained with $P_{st} = 100\%$ at a 50% overhead threshold. However, because of the high standard deviation, more users will see a performance above 1070



(a) 10,000 Basic Blocks per Check **(b)** 2 Million Basic Blocks per Check

Figure 2.19 Changing the Probability of Removing Dataflows Affects Overhead. Low probabilities of removing meta-data once the OH threshold is crossed result in more performance variance; some users observe many more dataflows than others. Error bars show standard deviation.

MIPS, nearly the same as a user with $P_{st} = 100\%$ at a threshold of 40%. These particular users experience less overhead than they would with a threshold of 50%, while the analyses they cannot perform can be made up for by the users whose performance is more than one standard deviation lower.

Figure 2.19b increases N_{bb} to 2 million. Decreasing the number of checks in this way obviously decreases performance, as the number of basic blocks probabilistically analyzed increases linearly with N_{bb}. However, it also increases the standard deviation of the performance at any particular mean. For example, the 4% data point in Figure 2.19b has nearly the same mean as the 1% data point in Figure 2.19a, but it has a larger standard deviation.

By tweaking the overhead threshold, N_{bb}, and P_{st}, it is possible for some users to analyze much deeper than they normally would while maintaining the same average overhead. The average slowdown of a user with $P_{st} = 100\%$ at a 50% overhead threshold is the same as a user with $P_{st} = 1\%$ at a 25% threshold if both have $N_{bb} = 10,000$. However, some individual runs for the latter user will analyze more of the program. Similarly, a user with $P_{st} = 4\%$ at a 50% overhead threshold will see the same average slowdown if at $N_{bb} = 2,000,000$. This user will probabilistically observe even more of the program.

This system may not be appealing to all users, but primarily serves as an example that modifying the sampling probability can significantly change both the overhead of a dataflow analysis and the probability of observing errors. The appropriate ratio between average performance, instantaneous performance, and program analysis coverage will be determined by each user, and is not further studied in this work.

2.6 Chapter Conclusion

This chapter presented a software-based distributed dataflow analysis technique. It detailed a novel method of performing dynamic sampling that allows users to control the overheads they observe when running heavyweight dataflow analyses. This can allow developers to distribute their software with dataflow analyses to large populations, allowing numerous users to test software for security, correctness and performance bugs.

The current design of the dataflow sampling system described in this chapter requires that shadowed data be set unavailable in the page table to indicate when it is accessed. Unfortunately, this has its down-sides. For instance, the granularity of virtual memory (typically 1-8 KB) is much coarser than that of most dataflow analyses (typically 1-8 bytes). This causes overhead problems, because page faults occur when tainted data is not actually being touched. It also affects the accuracy of the sampling system. The removal of shadow data from a region must be done on a page-granularity basis, meaning that shadowed values that share a page with the currently active dataflow will also be removed when that dataflow is forcibly ended. This can prematurely end the analysis of *multiple* dataflows, negatively affecting the accuracy of the sampling system.

Finally, the virtual memory system is difficult to use for multi-threaded programs. Because the VM status of any page is shared between each thread in a process, *every* thread would take a fault upon accessing a page that contained a byte shadowed by *any* thread. This has the potential to drastically increase the number of false faults, as well as the side-effect of eliminating the ability to perform demand-driven analyses such as data race detection.

The remainder of this dissertation will look at hardware systems to solve these problems. The next chapter will discuss Testudo, a hardware-based taint analysis system that can perform dataflow analysis in hardware, and that uses a simple on-chip cache to perform sampling of byte-accurate shadow values.

Chapter 3

Testudo: Hardware-Based Dataflow Sampling

> Each individual member of a co-operative society being the employer of his own labor, works with that interest which is inseperable from the new position he enjoys. Each has an interest in the other; each is interested in the other's health, in his sobriety, in his intelligence, in his general competency, and each is a guard upon the other's conduct.
>
> ---
>
> *Co-Operation of Labor*
> Leland Stanford

As discussed at the end of the previous chapter, software-based dynamic dataflow analysis sampling is limited by its reliance on the virtual memory system. The granularity of virtual memory pages is much larger than that of shadowed values, which means that there are overheads associated with unnecessary page faults that cannot be avoided without removing the meta-data from an entire page. Unfortunately, modern architectures do not contain any other mechanisms that can inform software when it is touching some of a large amount of shadowed data.

This chapter will detail a new hardware mechanism, Testudo, that performs dataflow analysis in hardware. Testudo, like other hardware dynamic analysis mechanisms before it, solves the virtual memory granularity problem by storing shadow values on a per-byte basis. This would normally require extra off-chip storage or expensive extensions to the memory hierarchy, but Testudo only stores shadow values on-chip. If this storage becomes full, the hardware then begins stochastically deleting shadow values, performing dataflow sampling in order to eliminate the need to store these values off-chip.

3.1 Limitations of Software Sampling

This chapter will mainly focus on taint analysis, a dynamic dataflow analysis that can be used to detect software security errors resulting from insufficiently validated external inputs. Like the taint analysis discussed in Chapter 2, the analysis in this chapter attempts to find control-flow attacks by tagging data that enters the application through I/O. These tags are then propagated during execution to other variables that are derived from the external data. Finally, they are used to determine the presence of potential security vulnerabilities when a tainted data value is used to access memory (e.g., as a base register in a memory access) or redirect program control (e.g., as an input to an indirect jump) [210].

Many taint analyses assume that relational tests such as "jump if this value is greater than x" are sufficient to ensure that a specific variable is safe, and they clear the taint bit from variables that are checked in such a manner. However, bug analyses have shown this to not always be the case [41]. Many errors are the result of input checks that are insufficiently constrained or outright wrong. Consequently, more powerful tests, such as *symbolic analyses*, can also be used to validate the bounds checks applied to external inputs [118]. This approach works by attaching symbolic expressions to externally derived data. For example, if the program flow passes a branch instruction in which an external value x was tested to be less than 5, then it satisfies the constraint $x < 5$ at that point in the program. When potentially dangerous operations are executed, the symbolic expression is checked to verify that derived constraints are sufficiently tight to prevent errors such as buffer overflows. This chapter will also look at methods of performing these types of analyses in hardware.

As discussed in Chapter 1, dynamic analysis techniques may be powerful, but they suffer from extreme slowdowns. Simple taint analyses may see overheads of 3-150$\times$ [93, 184], while the more advanced symbolic analyses can be upwards of 220$\times$ slower [118]. These slowdowns are the result of the significant software instrumentation needed to record, propagate and check shadow values. These overheads limit the number of tests that can be performed on a piece of software and reduce the number of users who would be willing to analyze programs. Chapter 2 discussed a system of performing *dataflow analysis sampling* which reduced the overheads seen by dynamic dataflow analyses by testing a subset of the observed dataflows seen during any individual execution.

Unfortunately, the method used to perform this software-based dataflow sampling has its limitations. At its core, dataflow sampling requires the ability remove any shadow values that are touched when the analysis tool is disabled. This is required to stop false positive results, where errors are declared even though they do not really exist. In order to do this, a dataflow sampling system must remove shadow values and then run faster when operating

upon unshadowed data. In other words, a demand-driven analysis system is the key to dataflow sampling.

Demand-driven analysis requires that software be informed whenever it is operating on data that should be analyzed. If software dataflow analyses were required to manually check for shadow values before each instruction, demand-driven dataflow analysis (and thus dataflow sampling) would still be too slow to effectively distribute to large populations of users. Simply accessing shadow values, as demonstrated by Umbra, can cause a program to run up to 3× slower [234]. Instead, demand-driven dataflow analyses, such as the one presented by Ho *et al.*, show that it is beneficial to have hardware inform the program on shadow value accesses. This means that the software runs at full speed when it is not accessing analyzable data, as the hardware checks these values in parallel to the regular execution. The hardware can then interrupt the software when it accesses shadowed data, enabling the analysis tool only when it should be turned on.

This could be done using hardware-supported watchpoints, but as we will detail in Chapter 5, there are an extremely limited number of these on modern processors [223]. Instead, Ho *et al.* use the virtual memory system [93] in a similar manner to the data breakpoint method pioneered in the DEC VAX architecture [19], formalized by Appel and Li [7], and subsequently patented by IBM two decades later [145]. They mark pages that contain shadowed data as unavailable in the virtual memory system, which causes a page fault whenever any data on these pages are accessed. The software page fault handler then checks if the accessed memory was shadowed, and if so, informs the software. This method can be much cheaper than checking each access with a software routine, as it will cause no overhead when operating on pages that contain no shadowed data. Ho *et al.* were able to show speedups as high as 50× over a system that was in the analysis tool at all times.

Unfortunately, this method of demand-driven analysis has its limitations. Page faults have high overheads because they must change permission levels to go to the kernel, and these cannot easily be removed. Figure 2.14a shows an example of this in the software-based dataflow sampling system built using virtual memory watchpoints. In this figure, the page faults that must be taken in order to remove shadow values cause a 19% drop in performance that cannot be recovered by sampling.

This problem can be exacerbated by a phenomenon that Ho *et al.* refer to as "false tainting," where page faults occur when accessing non-shadowed data on pages that contain some shadowed data. Because this can happen in tools other than taint analyses, this dissertation will instead refer to this problem as "false faults." Because virtual memory pages usually range from 1KB to 8KB and shadowed data is often 1 to 8 bytes in length, it is possible for thousands of shadowed and non-shadowed values to share the same page.

If any single byte on a page is shadowed, accessing every other byte will be extremely slow, as it involves: 1) taking an interrupt to the kernel's page fault handler, 2) checking the shadowed status of the data being accessed, 3) marking the page available, 4) returning to user-mode to single-step the instruction, 5) returning to the kernel after single-stepping the instruction, 6) once again setting the page unavailable, 7) returning to user-mode execution. We measured pathological cases of this on x86 Solaris that showed slowdowns of over 10,000×, though such heinous overheads are unlikely to be seen in the common case. For their demand-driven taint analysis tool, Ho *et al.* claimed that they "anticipate that using [finer-granularity] techniques would greatly improve performance by reducing the amount of false tainting" [93]. In a dataflow sampling system, improving this overhead would result in finding more errors under a particular overhead and reduce some of the unrecoverable overheads discussed in the previous paragraph.

Finally, virtual memory is difficult to use when trying to analyze multi-threaded programs. In most current operating systems, threads in parallel programs are contained within the same process and associated virtual memory space. In this case, if any byte on a page is shadowed *for any thread*, the page will be marked unavailable *in every thread*. This effect will increase the rate of false faults, as some threads may take page faults on pages where they have no tainted data. Additionally, analyses that wish to observe data movement between threads cannot use virtual memory to catch these events. While it is possible to split each thread into a separate process, this is often avoided. For performance reasons, operating systems such as Linux leave multiple threads in the same process to reduce TLB flushes. Additionally, multi-threaded programs that were transitioned to run as multiple processes would need to catch changes to the memory maps of any process (through system calls such as `mmap` and `sbrk`) so that they do not desynchronize. These problems make it difficult to change multi-threaded programs into multi-process programs, further increasing the difficulty of analyzing them with VM-based demand-driven analyses.

In summary, while these software-based analyses can increase the performance of some dynamic analyses and can be used to build some dataflow sampling systems, there is room for improvement in this area.

3.2 Hardware-Based Dynamic Dataflow Analysis

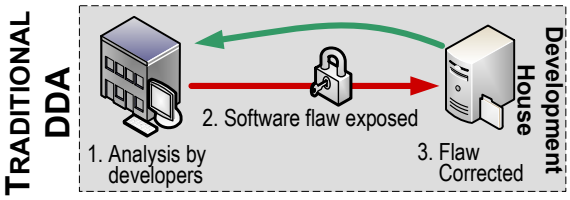
In an effort to mitigate the slowdowns caused by current software-based dynamic dataflow analyses, some researchers have proposed hardware-based dataflow analysis devices. In general, these techniques add storage space in the memory system for the shadow values and

extend the processor pipeline to compute, propagate, and check shadow values in parallel with the normal operation of the processor. For example, Minos [50], Suh *et al.* [210] and Chen *et al.* [38] each implemented hardware-accelerated taint analysis systems in such a manner. As Dalton *et al.* pointed out, these systems built their taint propagation rules directly into the pipeline, making them difficult to update and potentially insecure [52]. RIFLE, on the other hand, makes the information flow mechanisms architecturally visible, allowing binary translation utilities to track implicit information flows that simple taint analyses cannot find [218]. Dalton *et al.* also discussed methods of bypassing RIFLE's security mechanisms.

Dalton *et al.* therefore proposed Raksha as a solution to correctness problems in existing hardware taint systems [53]. It utilizes a form of microcoding for its propagation and checking rules, which allows multiple different variations on a basic dataflow analysis to run at any point in time. In order to allow some of these dataflow checks to be liberal in declaring faults, Raksha also takes advantage of user-level security exception handlers to verify any potential fault using complex software routines, rather than requiring hardware designers to make such algorithms at design time. In a similar manner, Venkataramani *et al.* focus on the flexibility of the taint analysis algorithm in their FlexiTaint system [219]. This design puts the taint propagation and checking logic entirely in software and then caches the results of propagation calculations in a hardware lookup table to reduce overheads. Venkataramani *et al.* also designed MemTracker, a more generalized hardware dataflow analysis system that uses a programmable hardware state transition table to perform different types of shadow analyses [221].

In general, these systems have very little runtime overhead. MemTracker, for instance, introduced less than 3% runtime overhead across a number of complicated checker mechanisms. Similarly, Raksha averaged a 37% slowdown across a number of SPEC benchmarks when running two different taint analysis schemes in parallel.

If we look at other types of dynamic analyses, Lam and Chiueh showed a method for finding array bounds violations using the segmentation features of the x86 instruction set [115]. This method has reasonable overhead, but it is limited in its power and flexibility by the relatively restrictive nature of the now-deprecated x86 segmentation logic. Chiueh then did a similar project that used the x86 hardware watchpoint registers to perform bounds checking [42]. The extremely limited number of debug registers in x86 meant that this required a complex system to effectively use them, resulting in a technique that is difficult to use in the general case.



	Traditional DDA		SW DDA Sampling
	In HW	In SW	
Runtime Overhead	**Low**	High	**Low**
Memory Overhead	Moderate	High	**Low**
Hardware Overhead	High	**None**	**None**
Shadow Accuracy	**Byte**	**Byte**	Page
Multi-threaded	**Yes**	**Yes**	No

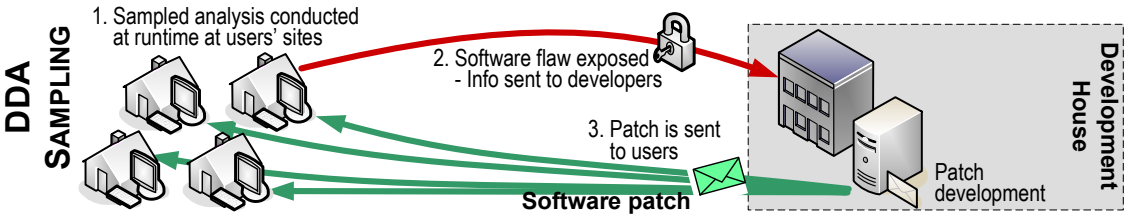


Figure 3.1 Comparison of Dynamic Dataflow Analysis Methods. Traditional software-based DDAs are carried out by the developer and incur high performance and memory overheads. Developers rely on it to identify and resolve bugs before an application is released. Hardware-based DDAs could be carried out by either the developer or by end-users, but require a large amount of extra hardware and take up a significant amount of memory. Software-based analysis sampling combines the results of many partial analyses by end-users. Hardware and performance overheads are minimal, but current systems require virtual-memory based shadow values that can cause extraneous false negatives and do not work well with multi-threaded programs.

3.2.1 Limitations of Hardware DDA

One of the major benefits of these hardware-based dynamic dataflow analyses over the previously discussed solutions is their ability to perform these tests with little runtime overhead *while maintaining high accuracy*. This is especially true because these systems keep track of shadow variables on a per-byte or per-word basis. Unfortunately, this requires storage space for these shadow values. For taint analysis systems, this can lead to overheads of up to one bit per byte, while more complex analyses can go even higher.

This memory overhead can yield significant design costs. Raksha, for instance, extends the size of each memory word by four bits to hold its taint analysis information. This requires modifications to a large number of components in the design, including caches, memory controllers, buses, and memories, which must all be expanded to accommodate the extra bits. Hardware designers outside of the research community may be wary of the increased design complexities of such a scheme, as it affects far more than just the pipeline of the processor. Additionally, it is unlikely that this approach could be extended to more complex dataflow analyses, such as dataflow tomography, which can require up to four bytes of shadow data for every byte of memory [148].

Other designs store shadow values in the existing memory system. FlexiTaint, in particular, stores the values into a packed array within the virtual memory of any process under analysis. It then caches these values in on-chip structures, much like any other data. This

has the benefit of requiring no changes to the hardware beyond the processor pipeline, but does cause unrecoverable overheads whenever the on-chip caches miss or overflow. The extra virtual memory taken up by the shadow values can crowd shared last level caches and physical memory, adding additional overheads.

The limitations of hardware and software dynamic dataflow analysis are summarized in Figure 3.1, along with a comparison to software-based dataflow sampling. Each has its benefits, such as the low runtime and memory overheads of software-based sampling. However, each has its limitations as well. Although hardware-based analysis is more accurate than software-based sampling, it comes at the cost of complex and expensive hardware additions, and higher memory overheads.

This chapter will focus on using sampling to reduce some of the complexities of hardware dataflow analysis systems. Rather than requiring expensive changes to the memory system, or slow shadow storage in main memory, this chapter will describe how hardware can stochastically remove some shadow values from the on-chip caches when they overflow. This removes any need to store data off-chip while the program is running, effectively performing sampling in hardware. Additionally, it should be possible to reduce the hardware complexity needed to perform difficult dataflow analyses. By using sampling to reduce performance overheads, these analyses can be performed by software handlers, rather than with purpose-built hardware.

3.2.2 Contributions of this Chapter

The remainder of this chapter describes Testudo[1], a hardware-based dynamic dataflow analysis sampling technique. Much like the software-based technique in Chapter 2, the goal of this design is to leverage large user populations to find software flaws which have slipped through in-house testing.

Testudo analyzes dataflows at runtime, but looks at only a small, randomly selected subset of the potential shadow variables. It utilizes additional logic in the pipeline that is similar to other hardware dataflow systems, allowing it to accurately perform taint analysis at no overhead. However, rather than storing the byte-accurate shadow values in an extended physical memory system or within virtual memory, Testudo incorporates a small on-chip *sample cache* to hold the shadow values of the dynamic analysis. When this on-chip cache fills, further shadow values are only allowed to enter using a feedback-controlled lottery system, after which they displace randomly chosen shadow variables that already exist in

[1]This terminology comes from the name of the ancient Roman military formation, where legionaries gathered their shields together to provide protection for the group as a whole.

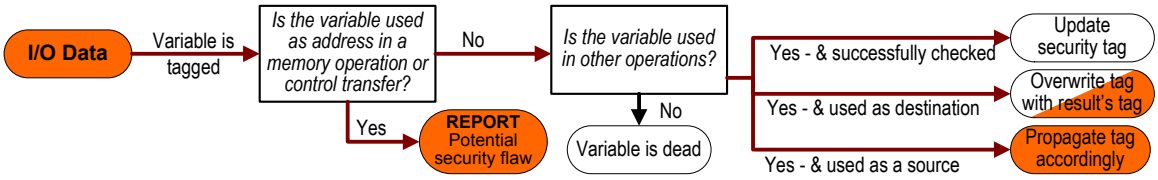


Figure 3.2 The Life of a Tainted Variable. Data brought in from an external source is tagged as potentially unsafe. Subsequent operations on that data remove the tag or propagate it to other variables based on analysis-specific rules. When a variable is overwritten by an operation's result, it inherits the tag of the operation's sources. If potentially dangerous operations are performed on tagged data, a warning is flagged.

the cache. The hardware performs dataflow sampling by stochastically removing shadow variables from the analysis.

Additionally, by taking faults to a software handler, it is possible to use this hardware to perform dynamic analyses that are not completely supported by the hardware shadow propagation system. This would normally have very high overheads, but by reducing the number of shadow values tracked during each execution, it is possible to arbitrarily limit the subsequent memory and performance overheads at the expense of needing more runs to achieve full-coverage analysis.

The remainder of this chapter introduces Testudo and analyzes its cost, performance, and analysis coverage. Section 3.3 introduces the hardware additions and shows how they can be used to accelerate dynamic analysis sampling. Section 3.4 details a theoretical model for the number of users needed to observe all of the dataflows within a particular program when using the sampling system. Section 3.5 details the experimental framework and uses it to show the overheads and accuracy of this system. Finally, Section 3.6 concludes the chapter. The majority of this work was originally published in the 2008 paper, "Testudo: Heavyweight Security Analysis via Statistical Sampling" [80].

3.3 Hardware for Dataflow Sampling

The overall goal of dynamic dataflow analysis is to tag certain important data, track its movement through the system, and find potentially erroneous operations performed on this data. This section details Testudo, a distributed dynamic analysis system that has byte-accurate shadow values and hardware support for dataflow sampling.

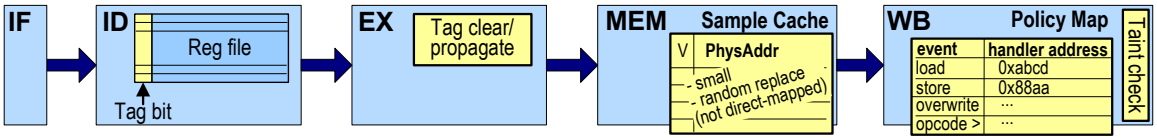


Figure 3.3 Testudo Hardware Implementation. To support sampled dataflow analysis, Testudo adds a small amount of hardware to the existing processor, as highlighted in the schematic. A tag bit is added to each entry in the register file to track shadowed data. Simple tag propagation logic is placed along the pipeline to handle setting and clearing of taint analysis tags. The sample cache is a small on-chip cache used to monitor a handful of physical memory addresses that are currently shadowed. The policy map enables a broader range of user-specified dataflow analysis semantics through the use of service handlers, invoked when particular operations touch tainted values.

3.3.1 Baseline Support for Dataflow Analysis

To implement the taint analysis studied in this chapter, the semantics of a program's execution must be extended to support the capability to: 1) tag program values that originate from program inputs, 2) monitor the dataflows that derive from these values in order to propagate tags, 3) detect checks on externally derived values for the purpose of validating external inputs, 4) detect when a tagged variable is used to perform a potentially dangerous operation such as a memory access or control transfer. An illustration of this approach is shown in Figure 3.2.

Variables are initially tagged when an external input is loaded into a memory location or register. Testudo utilizes a "tagged load" instruction to detect these loads, which device drivers employ to mark incoming data from an I/O device or DMA buffer. Use of the tagged load indicates to the system that the input value is from an external source and that it should undergo taint analysis.

All data storage locations conceptually support a tag bit, indicating that the value contained in that location is tainted. As shown in Figure 3.3, Testudo enhances each entry of the register file with an extra tag bit; tagged load instructions and regular loads of previously tagged data set this bit in the corresponding register file entry. Tag bits in the register file are propagated down the pipeline so that instruction results can also be tagged.

If a tagged value is used in a potentially dangerous operation (e.g., as the base register of a load or as the destination address for an indirect jump), Testudo will declare a security violation trap. Once the security trap handler is invoked, it has great flexibility: it can validate the action based on its input, send error-report information to the software developer, or even terminate the program to prevent malicious operations.

In previously proposed hardware tagging schemes, every memory location is potentially extended to accommodate the tag bit associated with the data it contains. In Raksha, for

instance, this is done by extending the memory bus and each word in the on-chip caches from 32 bits to 36 [53]. FlexiTaint, on the other hand, utilizes an on-chip cache dedicated to holding tags and stores these bits into a packed array in virtual memory whenever this cache overflows. As detailed in Section 3.3.2, Testudo consolidates all tag information into a *sample cache* that is similar to FlexiTaint's taint cache. However, rather than storing tags that overflow this cache into virtual memory, Testudo instead chooses to simply remove random tags from the cache and "forget" that they are tagged.

The baseline taint analysis semantics simply propagate binary tags to the results of certain instructions using a set of soft-coded rules, similar to the mechanisms presented in Raksha. To adapt to a broad range of software analyses, Testudo offers a policy enhancement mechanism that can be tailored to many dataflow analysis rules. These tests can be implemented through the use of the *policy map*, also shown in Figure 3.3. This mechanisms associates fault handlers with particular instruction types, allowing the hardware to interrupt execution when these particular instructions are encountered along with tagged data. System software can install handler routines, making it possible to set or clears tags, inspect arbitrary program state, and access private bookkeeping data in a non-invasive manner.

Symbolic bounds checking is one example of a dataflow analysis that can be implemented using this policy map [118]. This requires the creation of a symbolic interval constraint expression for each externally derived variable. Rather than simply propagating the trust status of these variables, it is possible to accurately compute their constraints by installing service handlers that are called when tagged values are loaded, stored, or operated upon. An additional handler then calls a constraint solver on potentially dangerous operations to determine if the tagged value belongs to an interval that could allow an illegal operation. Unfortunately, such checking can be much slower than the hardware-only taint analysis, as approximately 50 instructions of service handler code must be called for each instruction that accesses a tagged value. The original software-based implementation of this approach resulted in a 220$\times$ slowdown for programs with frequently accessed untrusted values. To tackle these situations, Testudo can control the slowdown experienced by a user through a dataflow sampling technique that is similar to the software-only system described in Chapter 2.

3.3.2 Limiting Overheads with a Sample Cache

The overarching goal of this work is to build a system that can perform dynamic dataflow analyses at extremely low hardware and performance costs by distributing the work over a large population of machines. To minimize the impact to users and designers, it is important

to devise techniques that limit both memory and performance overheads. It is also important to note that the hardware modifications that Testudo requires are limited to the processor pipeline. No changes or costs are incurred in existing caches, the memory system, or buses.

Traditional hardware-based dataflow analyses add one tag bit per memory byte to track shadow variables through memory. This memory overhead can result in considerable design costs, if physical devices such as caches, memory interfaces, and interconnect buses must be extended to support tag bits. However, in the most restrictive case, the analysis of a single path in a dataflow *can be implemented with storage space for a single shadow value*. If this storage space precisely held the shadow value that lead to, it could be replaced by the next entry in the erroneous dataflow once that value is created.

It follows from this observation that if we execute a program many times and select different paths each time, it is possible to eventually analyze the full program while strictly limiting memory and design costs. This observation is a more restricted version of the concept of dataflow sampling discussed in Chapter 2. If additional tag storage entries are available, it becomes possible to examine multiple paths and dataflows in a single execution, meaning that fewer executions will be required to explore every dataflow. Looking back on the previous software-based sampling scheme, it is also possible to limit the performance impact of software dataflow analyses by limiting the number of paths and shadow variables in the system. Therefore, there exists a convenient trade-off between dataflow analysis costs (both performance and memory) and the number of executions required to completely analyze the software.

As shown in Figure 3.3, Testudo limits the cost of tag storage by using a small storage unit called a *sample cache*. This cache holds the physical addresses of currently shadowed memory variables. For the most part, the sample cache behaves as any other cache, except that: 1) it does not contain data storage, since valid bits and physical tags are sufficient to denote that a memory variable is being tracked, 2) it uses randomized replacement and insertion policies to ensure that different dataflows are seen each time a program is run.

To ensure that users see as much of each dataflow as possible, Testudo only inserts *one* dataflow into the sample cache at a time. This policy makes it more likely to see all of a large, long-lived dataflow which would otherwise be quickly displaced by new shadow variables from other dataflows that would fight for the limited storage space. Limiting the sample cache to one dataflow at a time can be done by flushing the sample cache when a new dataflow is introduced. In Testudo, this happens when a value is loaded with the "tagged load" instruction. This means that Testudo implements two different replacement policies: that which chooses replacements within a single dataflow (intra-flow) and that which chooses between the different dataflows that occur within a program execution (inter-flow).

3.3.3 Intra-flow Selection Policy

While only one tagged dataflow resides in the sample cache at a time, it still may be the case that the current dataflow is larger than what can be stored in the sample cache. Consequently, the intra-flow policy must implement a replacement scheme that ensures high coverage analysis of large dataflows across multiple executions. Note that any traditional deterministic cache replacement mechanism, such as least-recently-used (LRU), would not work for the sample cache. In fact, any deterministic cache replacement policy would result in the same tagged dataflow values being examined in subsequent runs of the program (given the same program inputs). Consequently, Testudo uses a random replacement policy, whereby for each candidate replacement, all the resident cache entries as well as the newly arrived tagged address have equal probability of being evicted from the cache.

Moreover, for more complicated analyses, the chance of entering the sample cache can be reduced more sharply. For these tools, Testudo assigns a lower probability to variables that are not already in the cache, rather than a uniform random probability of eviction. There is a higher chance that new variables have not been analyzed before, meaning they could trigger more events that call to the slow software handler. This biasing is adjusted dynamically throughout a program's execution: Testudo monitors the frequency of analysis invocations as well as the average analysis cost per invocation, altering the eviction probabilities to keep the performance penalty at a constant, user-acceptable level. Thus, if the service handlers begin to be triggered more often or begin to incur higher overheads, new variables are disregarded with higher probability. In hardware, this dynamically adjusted random selection can be implemented with a weighted random number generator in as little as $2 * \log(cache\_size)$ LFSR bits and $\log(cache\_size) + 1$ weight bits [29].

3.3.4 Inter-flow Selection Policy

As previously discussed, the sample cache includes the shadow variables of only one dataflow at a time. The final important algorithmic decision is therefore how to select which newly created dataflow roots to track. This inter-flow selection policy is dynamically adjusted based on the arrival rate of new dataflows and the constraint on perceived performance impact. Each new dataflow, created with a "tagged load" instruction, polls the interflow selection mechanism to determine whether it can overwrite the dataflow currently being analyzed by the sample cache. To ensure high coverage of the dataflow currently in the sample cache, the probability of a flow replacement is set quite low. Conceptually, the execution of a program is partitioned into epochs. For each epoch, the number of challenges

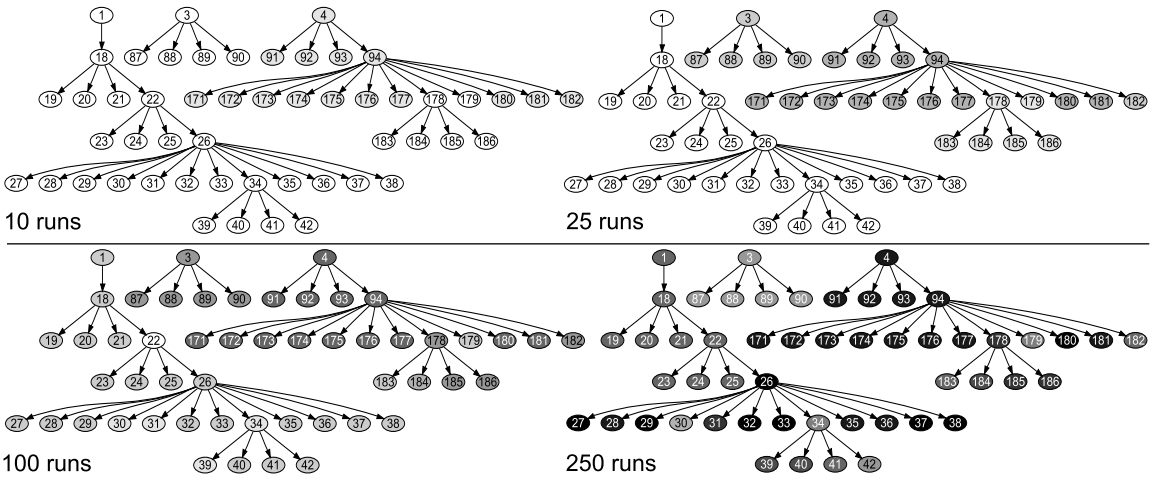


Figure 3.4 Testudo's Dataflow Sampling Coverage on Three Dataflows. These example dataflows are taken from the tiff.bad benchmark. Node labels indicate the chronological order in which each tainted variable arrives from the pipeline to the sample cache. Coverage results are shown after 10, 25, 100 and 250 runs on a system with a 4-entry fully-associative sample cache, where different shades of gray indicate how many times each node has been covered cumulatively (0=white, 10+=black).

to replace the dataflow under analysis is fixed (say, n). Two rules are then enforced: 1) each challenger has a small probability to overwrite the current dataflow ($0.1 \times 1/n$ in our experiments), 2) at most one challenge can be won during an epoch. The chance of entering the sample cache can further decrease if the performance impact of analyzing the currently dataflow goes too high.

3.3.5 Testudo Sampling Example

Figure 3.4 shows the dataflow coverage achieved for one of our experimental benchmarks, *tiff.bad*, by using the replacement policies discussed above over multiple executions. In this example, Testudo performs simple taint analysis while tracking only three of the dataflows of this benchmark. Hence, the service handlers have virtually no performance impact, and the replacement probabilities remain fixed. The figure shows the coverage attained after 10, 25, 100 and 250 executions of the program. Each node in the dataflow is colored with a different shade of gray, based on how many times it has been covered cumulatively (0=white, 10+=black). In this example, the sample cache is four entries and fully associative. Testudo was able to see every variable through to its final use, at least once, after 231 executions.

3.4 An Analytical Model of Dataflow Coverage

The small storage available in the sample cache limits the number of variables that can be analyzed during any particular execution of a program. One important question to ask is how many executions of a program (assuming it has the same inputs and is executed in a deterministic manner) are required to analyze each shadowed variable from its creation until its last use. In a taint analysis system, for example, this asks how many execution are required to see every value derived from I/O from when it is first tainted until it is last used (or until it is finally untainted).

Since the sample cache's stochastic replacement policy encourages covering different dataflows (and portions of dataflows) in subsequent executions, we are probabilistically likely to see every variable through to completion *eventually*. This section derives an analytical model that delineates an upper bound for the number of executions required to achieve this "full coverage." The model is parameterized with the size of the sample cache and the frequency at which tagged data are analyzed. This frequency is a function of the number of dataflows in the program, the dataflows' shapes, and the specific cache replacement policies selected in the analysis (see Section 3.3.2). To reiterate, this model is an *upper bound* on the number of executions required to achieve full coverage with a given probability. Section 3.5.3 will compare this bound against actual empirical measurements, and shows that Testudo performs much better in practice.

3.4.1 Analytical Model Overview

This analytical model computes the probability that a particular dataflow word has not been covered after a program execution. To reiterate an important point, a "covered" variable is defined as a variable that has been sent through the dataflow analysis engine (either hardware or software) at every point in the program that the variable is both accessed and shadowed. Note that each of these points need not take place within a single execution. If the use of a variable leads to an error, and if the dynamic analysis engine can catch that error, then the error is caught at least once if the variable is covered.

Using probability theory concepts, it is then possible to compute the chance that there exists *any* word in *any* dataflow that has not yet been covered after a program execution. This expression can be used to extend the model to estimate the probability of having any uncovered data after N executions.

3.4.2 Analytical Model Derivation

The first step of the model requires characterization of the program's dataflows to compute the probability, P_i, that any given word i resides in the sample cache at the end of an execution (or has been removed from it only after the word's last use in the program). While this value could be computed analytically for small sample cache configurations by fully enumerating all possible caching scenarios, it is computationally infeasible to perform this step of the analysis analytically for larger sample caches. Consequently, we measure these probabilities empirically by running Monte Carlo simulations on the externally derived dataflows of a program for a given sample cache configuration. Once we reach the desired confidence level, the P_i are computed as the fraction of times each tagged word is covered during the total number of simulated executions.

Given P_i for an individual word being covered in one program execution, we can compute the probability that this word has not yet been covered after N runs as $(1-P_i)^N$. From analytical calculus, this latter expression is always less than or equal to $e^{-P_i \cdot N}$, for P_i between 0 and 1. Hence, we can use the exponential as a conservative approximation of the original expression. That is, the probability that the tagged word i has *not* been covered after N runs is always less than or equal to $e^{-P_i \cdot N}$.

Finally, this inequality can be extended to include *any* tagged word in *any* dataflow by using the union bound: the probability that any tagged word in any dataflow has not yet been seen after N program executions is less than or equal to the sum of the individual events' probabilities. Thus, the probability $\rho(N)$ of having any uncovered tagged word after N runs is:

$$\rho(N) \leq \sum_i e^{-P_i \cdot N}$$

Thus, given the individual probability of covering each shadowed datum i, we can compute the upper bound probability of having any uncovered data after N runs. By plotting this result for increasing values of N, we can find the first value of N for which $\rho(N)$ is less than $1 - desired\_confidence\_level$. For example, if we want a 90% confidence of covering all tagged words in a program, we can sweep over N until we calculate $\rho \leq 0.1$. Note that the union bound provides a good approximation if the events are independent; when the events are correlated, the approximation becomes coarser, but is still conservative.

It is important to note that the model computes a worst-case estimate to achieve the desired confidence. As shown in Section 3.5.3, Testudo often requires fewer executions to accomplish the same goal. Any significant difference between the analytical model and the experiments is due to dependencies between the events (that is, covering shadowed datum i correlates with covering datum j). The more dependent the events, the fewer

executions will be required in practice to achieve full coverage. Nonetheless, the result of the analytical model is valuable because it equips developers with an approximation of how many executions will be required, in the worst case, to achieve full program coverage.

3.4.3 On the Relation to the Coupon Collector's Problem

It is also worth noting that the probability of a program being fully covered after N executions is a generalized case of the coupon collector's problem. This problem, in its simplest version, can be defined by the question:

> Suppose there is a group of N different coupons. Some brand of cereal contains a random coupon in every box, and each coupon appears with an equal probability. How many boxes of cereal must you buy to get at least one every each coupon?

If the sample cache had a single entry, and the probability of covering any variable was equal to the probability of covering any other variable, it would be possible to reword this question to request the number of program executions to cover all variables. Because this is a randomized process, the answer cannot be given as a concrete number (after all, one could feasibly obtain the same coupon every time). Instead, the answer can be given as a probability distribution, showing the chance of seeing all coupons after N trials. This distribution was originally described by de Moivre [56, 134], and the expected number of trials, ε, was categorized by Baum and Billingsley to be $\varepsilon = N \cdot H_N$, where H_N is the harmonic number, equal to $\sum_{k=1}^{N} \frac{1}{k}$ [18].

Because Testudo's sample cache can potentially cover more than one variable per execution, the categorization of the coupon collector's problem with singular draws (e.g. one coupon per box of cereal) is pessimistic. Laplace originally studied the probability distribution of the collector's problem with group drawings [117], while both Stadje as well as Adler and Ross studied the case where the goal it to collect specific subsets of the coupons [2, 209].

There is a wide range of literature in this area, but most authors focus on group drawings, where the probability of drawing any individual group is equal to the probability of drawing any other group. This is not the case for Testudo's sample cache. The act of removing a shadow variable earlier in a dataflow can make it impossible to see shadow variables later in the dataflow. In this case, later propagations may not occur, as the propagation logic would be unaware that the source variable was previously tainted. This means that the probability of covering one shadow variable is not equal to the probability of covering another. This

is why the probability of observing a single variable was computed using Monte Carlo simulations in Section 3.4.2.

The case of the coupon collector's problem with group draws and non-equal probabilities is, unfortunately, a poorly studied area. Thus, the rest of this chapter continues to use the pessimistic analytical model derived above.

3.5 Experimental Evaluation

Section 3.5.1 details the framework and benchmarks that were used for the experimental evaluation of Testudo. Section 3.5.2 then details the results of using Testudo as a taint analysis framework by reporting its performance and the user population required to cover 99% of each program's dataflows. These latter numbers are then compared to the analytical upper bound in Section 3.5.3. Next, Section 3.5.4 looks at applications other than taint analysis and evaluates the performance overhead of Testudo for a number of different software fault handlers. Finally, Section 3.5.5 reports the area, power and access time of the sample cache sizes studied in this chapter.

3.5.1 System Simulation Framework

To evaluate the performance of a taint analysis system based on Testudo, a data tracking simulator, which generates dataflows for each of the benchmarks, and a sample cache simulator, which analyzes these dataflows for coverages, were implemented. The data tracking simulator was implemented in Virtutech Simics [129], a full-system functional simulator able to run unmodified operating systems and binaries for a number of target architectures. The benchmarks in this paper were run on a model of an Intel Pentium 4 system running Fedora 5 Linux.

This setup also included a Simics module to track tagged memory addresses, general purpose registers, status/control flags, and segment registers. This module was implemented using code from the Bochs IA-32 emulator [119], and uses tag propagation and clearing techniques similar to those used by Chen *et al.* [38] and Suh *et al.* [210]. Tagged variables are tracked with 32-bit word granularity, as is done with Raksha [53], and all external data coming from disk, network and keyboard is tagged on arrival. The simulator only tracks dataflows relevant to the benchmark under study, by enabling the dataflow module when the Linux kernel or a process with the relevant process ID is scheduled. The traces produced by the tag tracker are then analyzed by the sample cache simulator.

Table 3.1 Benchmarks for Testudo. The table reports the number of execution cycles, tracked words and unique addresses, size of the largest dataflow and total number of dataflows for each benchmark.

	tiff.good	tiff.bad	xpdf.good	xpdf.bad	eggdrop.good
Total Cycles	2,600,000	3,500,000	15,000,000	15,000,000	114,000,000
Tainted Words	987	214	195,853	73,613	3,854
Unique Addresses	299	68	5,729	3,771	1,232
Largest DF # Words	116	59	185,465	64,015	678
Largest DF # Addresses	29	15	2,817	316	25
Number of Dataflows	71	17	540	960	150
	eggdrop.bad	telnet.server	telnet.client	lynx	httpd
Total Cycles	245,000,000	90,000,000	50,000,000	30,000,000	14,000,000
Tainted Words	486	75,323	27,529	518	47,745
Tainted Addresses	123	1,600	917	141	4,440
Largest DF # Words	154	71,345	20,663	173	46,214
Largest DF # Addresses	26	1,119	315	46	3,945
Number of Dataflows	10	29	50	10	46

Testudo was tested with a set of ten benchmarks consisting of popular Linux applications with known exploits. Table 3.1 summarizes the relevant statistics of these benchmarks, including program execution length (in cycles), number of tainted dataflows (i.e., external variables from untrusted sources), and the total number of tainted words to be tracked. It also lists the total number of unique addresses spanned by the tracked variables (two variables may share an address if they have non-intersecting lifespans, a common situation for stack variables). Finally, the table reports the size of the largest dataflow. Note that this does not count single-node dataflows. These cannot occur in Testudo, since a dataflow's root will be explicitly tainted by tagged loads from I/O spaces in memory. Though the values in memory-mapped I/O are tainted, they are not entered into the sample cache in order to save space. Any access to them comes from a tagged load, and only subsequent tagged values are stored into the sample cache.

The benchmarks, in no particular order, are:

- **Libtiff** - (*tiff.good, tiff.bad*). A library providing TIFF image manipulation services. Versions 3.6.1 and earlier fail to check bounds on RLE decoding [154]. *tiff.good* parses a valid image, while *tiff.bad* parses an invalid image in an attempt to exploit this problem.
- **Xpdf** - (*xpdf.good, xpdf.bad*). PDF document handling library. Versions 3.0.0 and earlier are vulnerable to a PDF format issue [157]: a malicious PDF file could contain

invalid tree node references that would never be checked. The two variants parse a benign and malicious PDF, respectively.

- **Eggdrop IRC bot** - (*eggdrop.good, eggdrop.bad*). Open source IRC bot. Version 1.6.18 fails to check bounds on data supplied by an IRC daemon, leaving it vulnerable to buffer overflow attacks [159]. This benchmarks receive benign and malicious data from an IRC daemon, respectively.
- **Telnet** - (*telnet.server, telnet.client*). A popular but insecure remote connection server. The benchmark client connects to a benchmark server and logs in to a user account.
- **Lynx web browser** - (*lynx*). Text-based web browser for command line environments. Versions 2.8.6 and earlier fail to properly check bounds on NNTP article headers [155]. The benchmark sends a malicious article header from a simulated NNTP server to the browser.
- **Apache server with PHP** - (*httpd*): The Apache web server and PHP have experienced numerous public exploits in the past. This benchmark is a simulated SQL injection attack where data from an HTTP request is used unsafely to generate a would-be SQL query. It returns the compromised query as an HTTP response.

3.5.2 Taint Analysis Coverage and Performance

This section details the accuracy and performance of Testudo when it is used as a taint analysis tool. The sample cache simulator was used to emulate the sampling of dataflows obtained from the data tracking simulator and to record the average number of runs required to cover 99% of all tainted data in the program. This was performed using a broad set of sample cache configurations: fully associative caches with 32, 64 and 128 entries, and 4-way set associative caches with 256, 512 and 1024 entries. The intra-flow eviction policy for the sample cache was uniformly random, while the inter-flow policy used an epoch length of 10 tainted dataflows and a 10% probability of winning at each challenge.

The results of these experiments are shown in Figure 3.5. Each graph plots the average coverage achieved for the total number of tainted words versus the number of runs required to achieve that coverage. Most benchmarks were evaluated with more than one sample cache size; however, this figure only shows a subset of the results due to space limitations. In all cases, high coverage can be attained by with a reasonable number of runs, even with very limited cache sizes.

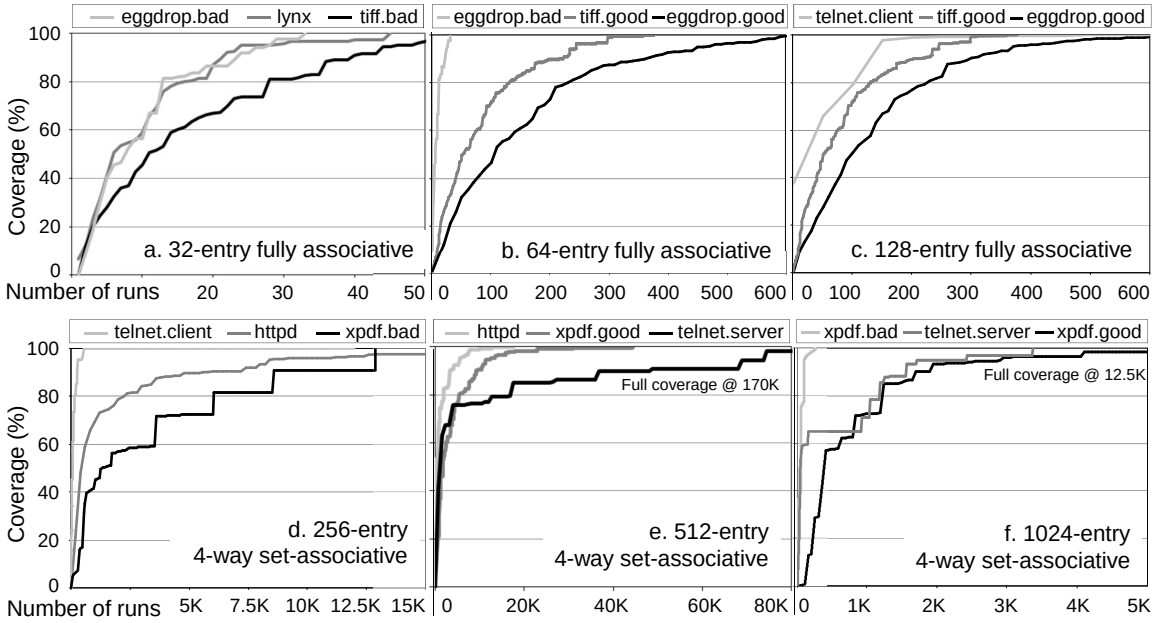


Figure 3.5 Dataflow Coverage vs. Execution Count for Various Sample Cache Sizes. The figures plot the number of runs required to achieve levels of dataflow analysis coverage. Increasing sample cache sizes reduce the runs required to achieve full coverage, up to a cache size where the largest dataflow can be fully stored (e.g., *telnet.server* in **e** and **f**). Beyond this threshold, no additional benefit is provided by larger caches (e.g., *tiff.good* in **b** and **c**).

As intuition suggests, increasing the size of the cache increases the coverage for a given benchmark at a particular number of runs. An alternate way to view this is that it reduces the number of runs required to reach a desired coverage target (e.g., *telnet.server* in Figures 3.5e and 3.5f). Once the cache size is sufficiently large to fully store the largest dataflow in a program, further increases do not provide any additional benefit (e.g., *tiff.good* in Figures 3.5b and 3.5c).

Note that these numbers are different than those measured in Chapter 2. Because the experiments in that chapter were not run in simulation, it was difficult to observe the "total dataflow coverage." Instead, those numbers show the number of runs needed to observe particular security flaws. The numbers in Figure 3.5 instead show the number of runs needed to see every point on every tainted dataflow for a particular input. This means that if an error existed at any point in any dataflow for this input, this number of executions would be adequate to find it.

The access time for the sample caches used in these experiments is smaller than the CPU clock cycle and L1 data-cache access time of the systems, as will be discussed in Section 3.5.5. Moreover, the propagation and checking of taint bits can be implemented efficiently within the pipeline, requiring no support from the policy map and the service

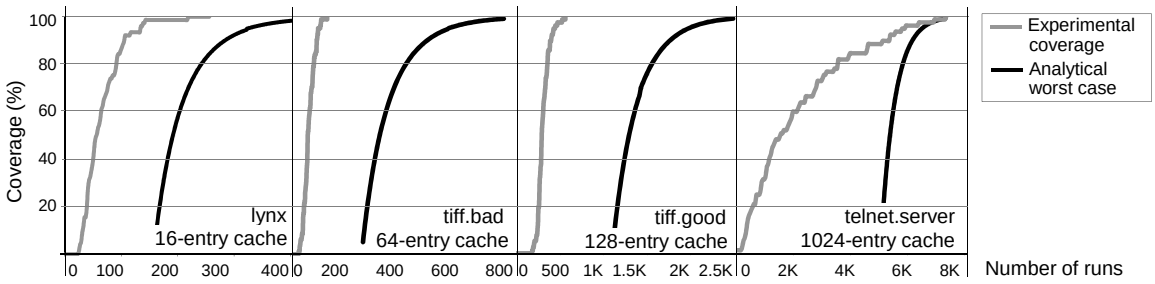


Figure 3.6 Experimental Dataflow Coverage and Analytical Upper Bound. These plots report coverage vs. number of runs for both our experimental results and the analytically computed upper bounds. The analytical upper bound is the probability of achieving full coverage after a particular number of runs. As an example, *tiff.bad* theoretically has an 80% probability of achieving full coverage in 400 runs. In practice, Testudo achieves full coverage in many fewer runs.

handlers. Therefore, Testudo can perform taint analysis without incurring any performance penalty and without affecting program runtime. Thus, for this application, Testudo provides a flexible trade-off between the sample cache size and the number of executions required to achieve high coverage. It performs analyses quickly, with none of the off-chip overheads of previous systems.

3.5.3 Worst Case Analytical Bounds

The analytical upper bound for a subset of the benchmarks listed above was computed to compare the performance of Testudo to the worst-case analytical model outlined in Section 3.4. Note that to conduct this analysis one must first know the coverage probabilities for each tainted value in a benchmark (as many as 200,000 words for *xpdf.good*). The straightforward solution to computing the coverage probability for each word is based on a Markov chain analysis of the dataflow. However, this approach becomes computationally intractable quickly with growing dataflows and sample cache sizes. To cope with this challenge, a Monte Carlo simulation was used with each fixed cache configuration, and the benchmarks were simulated until each tagged word in the program's dataflows was covered at least a few times. Each coverage probability, P_i, was then obtained by dividing the number of times each word was covered by the number of Monte Carlo runs.

Figure 3.6 shows the comparison between the experimental coverage of Section 3.5.2 and the analytical model derived in Section 3.4. The gray lines report the probability that a trial achieved full coverage during a particular execution, while the black plots indicate the probability of achieving full coverage in a given number of runs using the analytical model. Note that in this case, a trial means running the sample cache simulation until reaching full coverage.

As an example of the data this graph presents, the analytical model states that full coverage of *tiff.bad* can be achieved with 80% probability after 400 runs when using a 64-entry sample cache. As Figure 3.6 demonstrates, Testudo can in practice achieve this coverage up to 5× faster than the predicted worse-case number of executions. This is primarily because the analytical model is a bound that assumes that, as maximum, 64 entries would be able to be observed during any particular run. However, when actually performing dataflow sampling with Testudo, it is possible to replace variables in the sample cache after they have been completely analyzed.

3.5.4 Beyond Taint Analysis

This section investigates the performance overhead of conducting software analyses with higher overheads than taint analysis. In particular, while the previous experiments have assumed that the analysis itself causes no overhead, this section will assume that each time an instruction with tainted data is committed, it invokes a software handler.

As a consequence, to retain acceptable overall performance, we must limit the number of tagged variables sampled by Testudo in each program execution. This down-sampling, in turn, increases the number of runs required to provide adequate coverage of the program's dataflows. To gauge this impact, Figure 3.7 plots acceptable performance impact as a function of runs required to achieve 95% coverage of tainted words. The plots show the slowdown versus runs trade-off for several handler overheads, ranging from 100 to 1000 cycles for each call to a service handler. As a point of reference, Larson and Austin's input bounds analysis technique incurs approximately 50 instructions per instruction accessing a tagged value, though there would also be overheads caused by squashing the pipeline to jump to the handler. Even with fairly intensive service handler routines, Testudo can maintain a small level of perceived performance degradation by spreading the analysis over more executions.

The benefit of using Testudo, instead of a software-based system built on top of virtual memory, is its ability to track shadow variables on a per-byte basis. This, as well as the ability for hardware to remove the shadow values without software intervention, allows the system to avoid any unrecoverable overheads from problems such as false faults.

3.5.5 Sample Cache Hardware Overheads

This section analyzes the performance, area and power requirements of Testudo's sample cache hardware. These experiments used CACTI v5.0 [215] at 65nm/1.1V technology node.

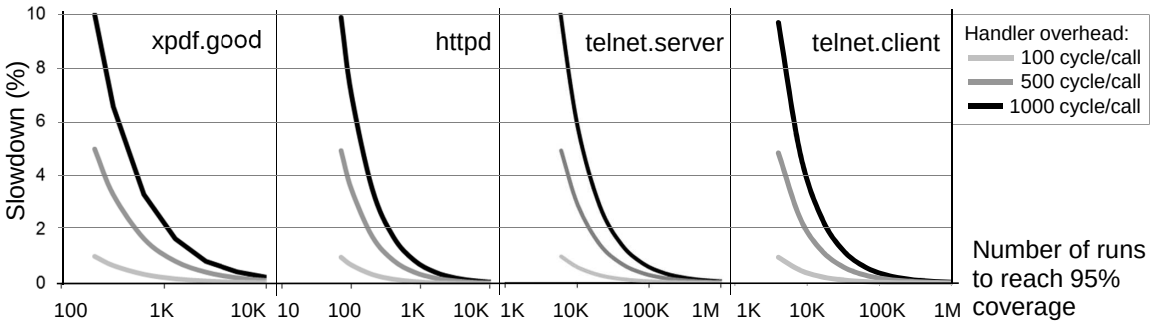
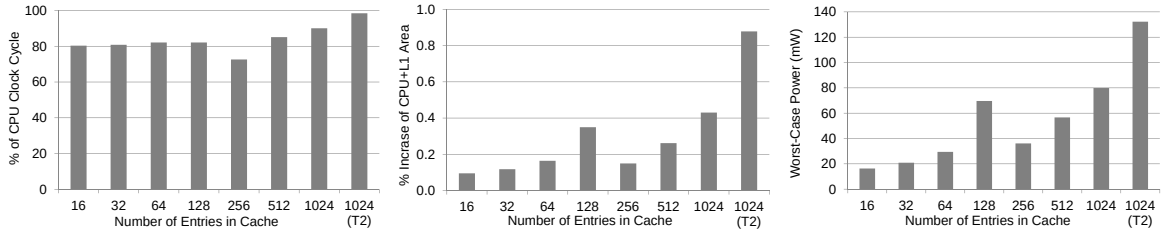


Figure 3.7 Performance Overhead for Software-Handled Analyses. More difficult software analyses are carried out through software interrupt handler routines linked in the policy map. These plots compare perceived execution slowdown vs. number of runs to achieve 95% coverage for three overhead levels incurred through the handler routines. This shows how limiting the amount of data in the sample cache can allow much tighter bounds on performance overheads than was shown in the sampling system based on virtual memory.

Fully associative caches with 16 to 128 entries and 4-way set associative caches with 256 to 1024 entries are evaluated. Each entry is five bytes wide, a typical physical address width in modern processors [100]. The impacts of the sample cache are compared against a complex modern processor, a 2.5GHz AMD Phenom [225], and a simpler core, a 1.4GHz UltraSPARC T2 [161].

Figure 3.8 plots access latency as a fraction of the CPU clock cycle, area and power overheads for the sample cache configurations previously studied. The first seven bars in each graph compare the sample cache against the L1 cache of the AMD Phenom. In the Phenom processor, L1 cache accesses are pipelined and take three cycles to return. This data thus show sample cache statistics as if they were pipelined across multiple cycles. The final bar shows the largest sample cache size, 1024 entries, versus the UltraSPARC T2. The L1 caches in the T2 cores are much smaller than those in the Phenom, but they are also non-pipelined and are accessed in a single cycle.

Figure 3.8a shows the access time to the sample cache as a percentage of the respective processor's clock cycle. Note that in all cases, the access time to the sample cache is smaller than the L1 cache access time. Thus, no performance impact is experienced when the system is augmented with Testudo's sample cache hardware. Figure 3.8b shows that the area impact of the sample cache is minimal, less than 1% in all configurations. The graph compares the sample cache's area against the core and L1 cache area of the Phenom processor ($25.5mm^2$ as reported in [57]), and against the Niagara's core plus cache combination ($12.5mm^2$, as manually derived from die photos) in the last bar. Finally, Figure 3.8c shows worst-case power requirements. To model a worst case scenario for the sample cache, we assumed that each instruction is a memory operation accessing the sample cache. As indicated by the



(a) Clock Cycle **(b)** Area Increase **(c)** Power Usage

Figure 3.8 Delay, Area, and Power Overheads of the Sample Cache. From the left to right, the charts show access latency, area, and worst-case power estimates for several sizes and configurations of the sample cache. The first four bars in each graph are for fully-associative caches, while the last four correspond to four-way set associative configurations. The first seven bars in each graph are compared against a 2.5GHz AMD Phenom, while the last is against a 1.4GHz Sun UltraSPARC T2.

figure, even worst-case power requirements are quite modest. In practice, one would expect these power requirements to be much less than this value since not all instructions access the sample cache, making the total power overhead very small. Note that the Phenom and UltraSPARC T2 dissipate 125W and 84W in total, respectively.

This section does not analyze the overhead incurred due to the pipeline modifications needed to include the tag bits in the register file and propagate them through the pipeline. This overhead is common to all hardware taint analysis systems, and it is unlikely that this hardware, which is operated in parallel to the rest of the pipeline, would affect the core's critical path.

3.6 Chapter Conclusion

This chapter described Testudo, a hardware-based programmable dataflow analysis sampling mechanism. The approach is based on the same distributed analysis technique described in the previous chapter, where a random selection of a program's dataflows are analyzed during any particular execution. Testudo does this by performing dataflow analysis in hardware and keeping word-accurate shadow values in an on-chip sample cache. This cache utilizes a random replacement strategy that ensures that successive runs of the program are likely to see different externally derived variables. To minimize the performance impact of heavyweight analyses, the random selection mechanism dynamically monitors performance overheads and limits the introduction of new dataflows into the sample cache if overheads encroach the budgeted performance.

To understand the nature of this approach, its performance was empirically analyzed through full-system simulation, and an analytical model that estimates the number of pro-

gram runs necessary to achieve high-coverage analysis was developed. These experiments showed that Testudo is low cost, and it achieves high program coverage, often within thousands of executions in the worst case. Additionally, the area and power costs of the sample cache hardware only increase system costs by a few percentages and cause no extra pipeline latency.

Testudo focuses on sampling dataflow analyses, but many other useful dynamic software tests need better performance as well. The next chapter therefore looks at a mechanism for accelerating data race detection, an analysis of increasing importance as parallel programming enters the mainstream. In particular, it will show a novel way of using simple, existing hardware on commodity processors to enable demand-driven data race detection.

Chapter 4
Demand-Driven Data Race Detection

> Technical innovations are often understood as the creation of new technology—new materials, new chip designs, new algorithms. But technical innovation can also mean using existing technical constraints in new ways, something that produces interesting results when combined with creative goals.
>
> *Racing the Beam: The Atari Video Computer System*
> Nick Montfort and Ian Bogost

Multi-threaded programs are becoming more important as parallel processors become ever more popular. Software performance improvements now hinge on programmers' abilities to find and exploit explicit parallelism in their code. This paradigm shift has opened up a veritable Pandora's Box of new problems, chief among them concurrency bugs such as data races. This means that tools to find these bugs, such as data race detectors, are also of increasing importance.

Data race detection analyzes the movement of data between threads and attempts to decipher whether these accesses are appropriately locked. Performing this analysis in hardware, or having hardware alert software that such a test is needed, requires observing not just the actions taking place on a single processor, but the *interactions* between multiple threads and/or processors.

This chapter will explore methods for performing hardware-assisted demand-driven data race detection. Unfortunately, the virtual memory method of demand-driven analysis discussed in Chapter 2 cannot easily be used for data race detection. Because individual threads in a program share the same virtual memory space on most modern operating systems, this type of demand-driven analysis would require splitting the multi-threaded program into a multi-process program. This is difficult, in and of itself, and can also result in high overheads as each process attempts to communicate its shadow values between all other processes in the program. Instead, this chapter will show how to use different hardware that

exists on some modern commercial microprocessors to keep a software data race detector disabled until it is likely that two threads are actively sharing data.

The previous chapters have focused on oblivious sampling methods, where the analysis is stopped at random intervals regardless of the dynamic state of the program under test. This chapter instead looks at a slightly different way of accelerating data race detection, where the analysis is only enabled when it is likely (though not guaranteed) that data races are not occurring. There have been previous works on sampling software data race detection [26, 65, 131], and each uses a different oblivious method for deciding when to disable the tool. This chapter presents a new data race sampling technique that approximates a best-effort demand-driven analysis. By performing this best-effort demand-driven analysis *in conjunction* with previous sampling systems, it may be possible to find more errors under a particular overhead threshold. In other words, this chapter explores a mechanism that can make previous sampling more systems effective by utilizing information about the data usage of the program.

4.1 Introduction

Parallel programming has quickly entered into the mainstream. Consumer processors are already well into the multicore domain, and it does not appear that the trend will abate. Software performance increases currently hinge on programmers taking advantage of more processing cores, but parallel programming is marred with difficulties. Chief among these problems are concurrency bugs such as data races [125].

Like other software correctness problems, significant efforts in the research community have yielded powerful tools to help create more robust software. This includes both tools for automatically finding data races that occur during execution. Dynamic data race detectors can help a developer locate code that contains improperly synchronized memory accesses without requiring tedious code reviews or long debugging sessions. Without automated testing tools, these errors can be difficult to detect (until a crash occurs!) and even harder to locate.

Dynamic data race detectors observe when two threads access a shared memory location and calculate if it is possible for the threads to do this in an unordered way. Happens-before race detectors, the style optimized in this chapter, indicate a race when two threads access a shared variable, at least one is a write, and there is no intervening synchronization event between them. This type of race can lead to nondeterministic output because external factors such as scheduling, bus contention, and processor speed will ultimately determine

the ordering of these accesses. This yields ultimate control of the program to these factors, which is rarely the programmer's intent.

Like other dynamic analysis tools, dynamic data race detectors can be powerful, but also slow. It is not uncommon to see overheads of up to 300× in commercial race detectors. As mentioned previously, such overheads are especially troublesome for dynamic analysis tools because they reduce the degree to which programs can be tested within a reasonable amount of time. Beyond that, high overheads slow debugging efforts, as repeated runs of the program to hunt for root causes and verify fixes also suffer these slowdowns.

There have been numerous methods proposed to accelerate dynamic data race detectors. Some software-based mechanisms, for example, attempt to filter out accesses that cannot possibly cause races in order to reduce the number of calls to the detector [190]. These mechanisms can mitigate race detection slowdowns to a certain degree, but the software filtering mechanism itself will still contribute large overheads even if it filters most memory operations. Other researchers have proposed embedding race detectors in hardware [181, 236]. Though these increase performance, they incur large hardware overheads for non-upgradable, single-purpose designs. CORD, for instance, requires extra storage equaling 19% of the last-level cache in addition to the hardware used to calculate data races [181]. Other researchers have proposed sampling mechanisms that randomly disable the data race detector a certain percentage of the time in order to keep overheads below a user-defined threshold [26, 65, 131]. These oblivious sampling systems can effectively control overheads, but may miss a large number of data races because many analyses that have a high probability of finding a data race might be skipped.

4.1.1 Contributions of this Chapter

This chapter presents a novel approach to optimizing the performance of software-based race detectors. This chapter first describes a *demand-driven data race detector* – a data race detector that is only enabled when it is dealing with potential races. This demand-driven detector turns on when data is shared between two threads and turns off when no sharing is observed. To enable this capability, an ideal *hardware sharing monitor* is described that informs the software detector when sharing happens, allowing the detector to watch for data sharing with no runtime overhead. A non-ideal version of this sharing monitor is then built using hardware performance counters that interrupt execution on inter-processor cache sharing events, an indicator of inter-thread data sharing. Because this sharing detector is not ideal, it may miss some sharing events, leaving the detector disabled when a race actually happens. In a sense, then, this indicator is used to build a sampling data race detector

that takes samples when it is likely that races could occur. While normal software race detectors must *check if the race detection algorithm should analyze each memory access*, this best-effort demand-driven analysis is able to utilize hardware to perform these checks in parallel with the execution of the original application.

This can yield a large increase in performance over a continuous-analysis software race detector when little data-sharing occurs. The race detector in Intel Inspector XE, a commercial dynamic analysis tool, was modified to perform demand-driven data race detection, and the performance increased by 3× and 10× for two benchmark suites. Because the cache events that existing performance counters can see may miss some racy accesses (due to limited cache capacity, for example), this chapter also studies the accuracy of this best-effort demand-driven race detector. In experiments, this design finds most data races seen by the slow continuous-analysis tool. Mitigation techniques for the few remaining situations are also studied.

This chapter, the majority of which was previously published in the 2011 paper "Demand-Driven Software Race Detection using Hardware Performance Counters," [79] makes the following contributions to this thesis:

- It **introduces a demand-driven approach to race detection** that only invokes expensive analyses when it is possible for races to occur.
- It **develops an efficient sharing indicator** that detects, at little runtime overhead, when expensive data race detection analyses should run.
- It **describes and evaluate a commercial-grade implementation of the technology** and shows that it is possible to find new data races in a popular parallel benchmark suite while reducing analysis overheads by 3-10×.

This chapter is organized as follows: background works are reviewed in Section 4.2. Section 4.3 presents a design for a demand-driven data race detector that uses idealized hardware, while Section 4.4 discusses methods of monitoring data sharing in order to build the demand-driven detector. Because the hardware that currently exists is less than ideal, Section 4.5 describes a demand-driven tool that uses currently existing performance counters. The setup of the experiments and the results of these tests are detailed in Section 4.6. Finally, related works are discussed in Section 4.7, and the chapter is concluded in Section 4.8.

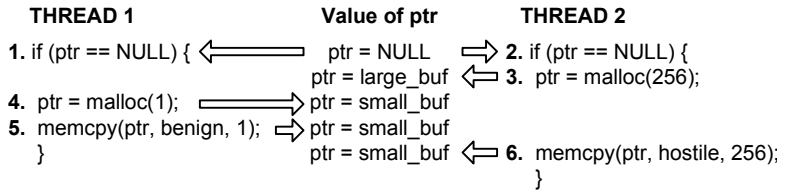


Figure 4.1 A Data Race Resulting in a Security Flaw. This example, also shown in Chapter 1, shows a security flaw caused by a data race. The numbers represent dynamic execution order. Both threads attempt to initialize the shared pointer (1, 2). Due to the lack of synchronization, thread 1 allocates a small buffer (4), but thread 2 copies a large amount of data into it. This particular ordering can cause a buffer overflow, exposing a potential security flaw. If (4) occurred before (2), this code would run correctly; (3) and (6) would not execute due to the conditional check at (2).

4.2 Data Race Detection Background

Concurrent execution can lead to bugs that would not occur in sequential code. *Data races*, for instance, occur when two parallel threads of execution access a single memory location without a guaranteed order of operations between them. More formally, a data race occurs when two or more threads access a single memory location, at least one access is a write, and there is no enforced order between the accesses [168]. These situations can introduce unintended, and sometimes dangerous, values into the program when they arise, as demonstrated in Figure 4.1.

This example is drawn from CVE-2010-3864, a security flaw found in some versions of the OpenSSL TLS library [160]. This flaw allowed two threads to access a shared pointer without synchronization. In some circumstances, each thread may attempt to initialize it to a different heap object and subsequently attempt to copy data into the object. If, however, the accesses to the pointer are interleaved (as shown in the figure), one thread may attempt to copy a large amount of data into a small heap object, causing a potentially dangerous heap overflow [180]. No overflow would occur if the memory accesses were performed in a different order. This data race could allow a heap-smashing attack to take place, potentially giving a remote attacker the ability to run arbitrary and dangerous code on the machine.

There are three types of data races that can affect the output of a program:

- W→W (a.k.a. Write After Write, or WAW): one thread writes a variable and another thread then writes to the same location.
- R→W (a.k.a. Write After Read, or WAR): one thread reads a variable and another thread then writes to the same location.
- W→R (a.k.a. Read After Write, or RAW): one thread writes a variable and another thread then reads from the same location.

R→W and W→R races are related; the difference between the two is the observed order of the accesses. For example, if the reading thread in an R→W race happened later, the race could become a W→R race.

Data races can be mitigated by ensuring that there is some enforced order on the accesses to a variable. For instance, putting mutual exclusion boundaries around each thread's access to a variable will ensure that no other thread will operate on that variable at the same time, forcing a partial order on the program.

It is difficult to manually find data races because they can cause the program to behave contrary to the intuitive sequential ordering implied by the programming language. Their side effects may also occur infrequently, making them difficult to trace. To combat this, dynamic race detectors observe the execution of the program and automatically detect when the program is performing a racy access.

The happens-before race detector utilized in this paper uses Lamport clocks to divide programs into segments based on thread synchronization operations [116]. On memory accesses and synchronization operations, it updates and analyzes the values of these clocks to determine if two segments that access the same variable could execute in parallel, indicating that a data race could occur [13, 14].

While dynamic race detectors are useful when searching for concurrency bugs, they come at a high runtime cost. They must check each access to memory in some manner, yielding high overheads. It is not uncommon to see overheads ranging from 20× [199], to 200× [214], to upwards of 1000× [126] in commonly used tools. Besides making these tools more difficult to use, these slowdowns negatively affect their capabilities. As previously discussed, dynamic analysis tools can only observe errors in portions of programs reached during a particular execution, and slow tools limit the number of runtime states that can be observed before a software product must be shipped.

4.3 Demand-Driven Data Race Detection

Software-based dynamic race detectors have extremely high overheads because they conservatively perform sharing analysis on every memory access. As Sack *et al.* pointed out, many memory operations need not go through the entire race detection algorithm because they could not possibly cause data races [190]. However, the very act of checking whether a memory access should be sent to the race detector will cause slowdowns in a software-based analysis tool. This section describes a *demand-driven data race detector* that is only active when the software-under-test is executing instructions that should be sent through the race

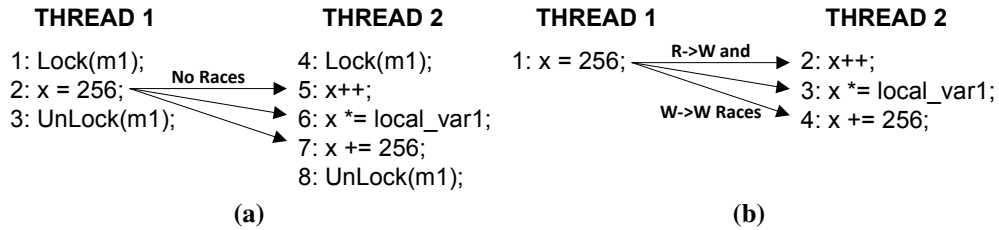


Figure 4.2 Multiple Race Checks Yielding the Same Answers. The numbers represent dynamic execution order. **(a)** The race checking on instructions 6 and 7 will always yield the same "no races" answer as instruction 5, because they are all protected by the same lock. **(b)** Each check in thread 2 will show the same data races on x as no other threads are interleaved.

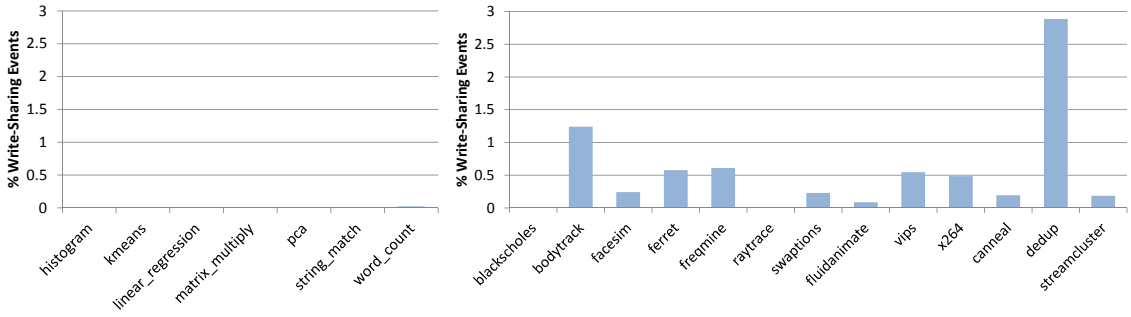
detection algorithm. A hardware model is then described that enables the design of a race detector that causes no runtime slowdowns when no races could possibly occur.

4.3.1 Unsynchronized Sharing Causes Races

In the most general sense, race detection is only important when data sharing is occurring between threads or processes. As Netzer and Miller described, apparent data races are "failures in programs that access and update shared data in critical sections" [168]. For a dynamic race detector, this means that accesses to unshared data need not be sent through the race detection algorithm. For instance, if two threads are writing to and reading from thread-local data, a happens-before race detector could not possibly observe a data race – each variable is accessed by a single thread whose Lamport clock is monotonically increasing.

Additionally, shared variables need not be sent through the race detector on every access. A happens-before race detector will not return a different result if a variable it is checking has not been shared since the last time it was checked. Figure 4.2 shows two examples of this. For these dynamic orderings, the subsequent accesses to the variable x by thread 2 *cannot* result in a different answer in either case. This means that the only important accesses for this data race detector are those that are participating in a dynamic write-sharing event. In other words, the first access to a variable that was last touched by another thread is important, but all subsequent accesses (until another sharing event) are not.

It is possible to significantly reduce the number of memory accesses that must be sent through a data race detector by utilizing these two insights. Figure 4.3 shows the number of memory accesses that are actually participating in write-sharing events across two parallel benchmark suites. On the whole, less than 3% of dynamic memory operations need to be sent through the software race detection algorithm, and for data-parallel benchmarks like those found in the Phoenix suite, the number is much smaller still.



(a) Phoenix Suite **(b)** PARSEC Suite

Figure 4.3 Dynamic Write-Sharing Events in Two Parallel Benchmark Suites. The percentage of dynamic memory accesses where the location was last touched by another thread and at least one of the two accesses is a write. These are the only memory operations that can yield new results from a data race detector. **(a)** The Phoenix benchmark suite is a collection of data-parallel programs, and so has very little sharing. Often, only hundreds of accesses out of the billions in the program are participating in sharing. **(b)** The PARSEC suite has much more sharing, on average, but less than 3% of the accesses need to be sent through a race detector.

Sack *et al.* took advantage of this by using software filters to reduce race detection overheads [190]. Their FSM filter utilizes the first insight to filter all accesses to a variable until multiple threads touch it. Their duplicate filter uses the second insight to remove checks on variables that have previously been analyzed since the last time they were shared. While this software-based filtering improves the total race detection performance, it is, unfortunately, still quite slow. Even if the program-under-test has little data sharing, and thus most operations are not analyzed, each memory access must still be analyzed by the sharing filter.

4.3.2 Performing Race Detection When Sharing Data

We designed a software race detector that is only enabled when a thread is operating on shared data in order to take advantage of the fact that races are tied to inter-thread data sharing. This detector relies on the hardware to inform it of data sharing events; when none are occurring, it need not attempt to find data races. The hardware utilized in this section is idealized and performs the checks illustrated in Figure 4.4 on each memory access.

On each memory operation, it first checks if the accessed location is thread-local or at least not currently participating in a write-share. If either case is true, the access need not be sent through the race detector, as explained in Section 4.3.1. This is accomplished through a form of eager conflict detection, like those implemented in hardware transactional memory [144] or race recording systems [97]. This detection mechanism compares the ID of the

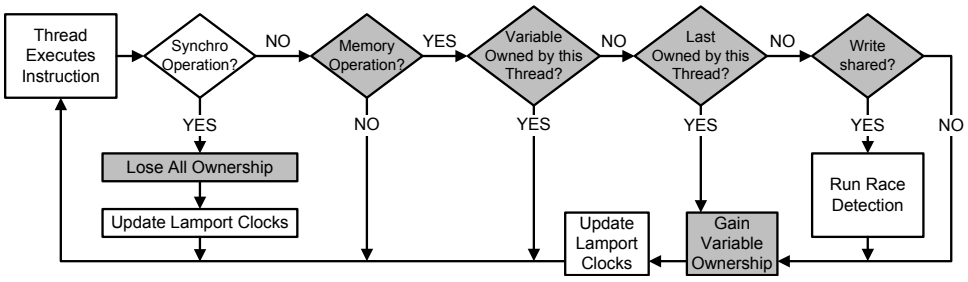


Figure 4.4 Ideal Hardware-assisted Demand-Driven Software Data Race Detector. This algorithm details an ideal demand-driven race detector. The dark boxes are performed within the hardware, which tells the software race detector to run only when it detects data sharing that could cause races. The data race detection itself (and the race detector's updates to the associated Lamport clocks) are executed in software.

thread that previously touched this location with the current thread ID; if they are different and either is a write, the system should check if a data race has occurred. Afterwards, the ID variable associated with this location is changed to the current thread. To this end, a thread gains *ownership* of a variable when accessing it.

This can be done with ownership records similar to those described by Damron *et al.* [55] or by keeping meta-data alongside each byte in memory. Regardless, inter-thread data sharing is detected using the ownership of a variable. If thread 1 owns a variable and thread 2 accesses that location, there is data sharing. The hardware must now check if either access was a write, and give thread 2 ownership of the variable for this type of access, as it is now the most recent thread to touch it. If the data *is* write-shared, then the software race detector should be run on this instruction. For example, if thread 1 write-owned a variable, and thread 2 attempted to write to it, there is W$\rightarrow$W data sharing. Thread 2 is granted write-ownership of the variable and the access is sent through the software race detector.

This demand-driven analysis system begins by performing no race detection, instead only tracking the Lamport clocks associated with each thread and lock (as, for instance, LiteRace does when its race logging is disabled [131]). When the software attempts to execute a synchronization operation (or other operation that increments its Lamport clock), it asks the hardware to revoke all of this thread's ownerships. This is done so that the next time a variable is accessed by this thread, the hardware will cause the software race detector to update the location's Lamport clock, even if no other thread has touched it.

While the program is running, it checks the ownership of every memory access. Unlike a transactional memory system, this occurs at *every* access in a multi-threaded program because we cannot rely on races to exist only within demarcated regions. In a sense, we must use a system like the debug transactions described by Gupta *et al.* [84]. If this thread owns the memory location, the access will complete without intervention. If not, the hardware

must then check if some other thread currently owns (or most recently owned, as described above) this variable. If this is not the case, this thread gains ownership of the variable and the hardware signals to the race detector to updates the variable's clock appropriately. Subsequent accesses by this thread to that variable will not cause any hardware events, and the program can run at full speed.

If the variable is or was most recently owned by another thread, however, this access causes data sharing between two or more threads. The hardware will detect this and, if one or both accesses to this variable were writes, will inform the race detector that it must check this access for data races. This system ensures that the software data race detector does not perform any actions unless they are required. The race detector does not perform the slow analysis algorithm when the threads access unshared variables. When data sharing occurs, the signals from the hardware precisely inform the race detector when it should activate, ensuring that no time spent in analysis is wasted.

4.4 Monitoring Data Sharing in Hardware

As described in the previous section, the hardware to enable a demand-driven race detector needs a number of capabilities to perform the checks for inter-thread data sharing. Beyond the logic previously described, the architecture must be able to:

1. Set thread-ownership of individual memory locations.
2. Keep separate read/write ownerships.
3. Invoke fast faults or interrupts to the race detector.

Without the first two, it would be impossible for the hardware to properly perform sharing detection. If it had no concept of thread-ownership, the hardware would be unable to detect inter-thread sharing. Similarly, separate read/write ownership is needed to avoid performing race detection during read-only sharing, a common occurrence in parallel programs.

Without the third capability – fast interrupts – it is unlikely that this system would be any faster than performing the race detection in software. If each fault needed to go through the kernel in order to run race detection on a single memory operation, sections of parallel software with large amounts of data sharing would become unbearably slow.

No hardware with these abilities currently exists, though some researchers have described designs that nearly fit the bill. Witchel's Mondriaan Memory Protection system [227] and UFO as described by both Baugh *et al.* [17] and Neelakantam and Zilles [163] are examples of fine-grained memory protection systems that could be repurposed to pro-

vide thread-ownership of memory regions. Even then, application-specific hardware (or a programmable state machine such as described in MemTracker [220]) would need to be added to perform the sharing analysis.

These systems would require significant changes to the architecture of existing processors, so it is unlikely that they will be built to accelerate a single application. Instead, this section looks elsewhere for hardware mechanisms that, while not perfect, may be useful in building a "best effort" demand-driven data race detector.

4.4.1 Cache Events

It is possible to observe data sharing by watching for particular cache events. A W→R memory sharing event occurs when two threads write to the same location. Assuming that each thread is in its own core, and each core has its own MESI-style private, coherent cache, the first write transitions the cache line that contains the variable into a modified state. When the second thread reads the line, the data is fetched from the remote cache that was in the modified state. Similarly, W→W data sharing occurs whenever one cache contains a line in the modified state and another cache attempts to read the modified line with a read-for-ownership (RFO) operation. R→W sharing is slightly different, as it does not begin with a cache line in the modified state. Nonetheless, it can be seen when a cache line reads-for-ownership from an exclusive or shared line in another cache.

Because of limitations inherent to caches, it is not readily possible for a demand-driven data race detector to observe every data sharing event by using cache information because:

1. Caches have limited capacity. If, for instance, thread 1 wrote to a location, the line was evicted, and then thread 2 read the location, no cache events would signal this sharing.
2. Cache events are performed on a process***or***, not process or thread, basis. Two threads that share data may be scheduled on the same processor at different times, allowing data sharing through the common cache.
3. It is possible to have more threads than processors. Cache sharing events may overlook data that was shared between active and inactive threads.
4. Simultaneously multithreaded (SMT) systems have multiple threads share the same cache, so some data sharing is not visible with inter-cache events.
5. The cache events must be observable by software if a race detector is to use them.

6. False sharing can cause races to be missed. If thread 1 writes variable *x*, putting its cache line into the modified state, thread 2 may read variable y, moving the same line to shared. This would make it difficult to observe a later access to *x* by thread 2.

The first problem has been addressed by researchers of hardware-based race detectors and transactional memory systems. For instance, Prvulovic measured the differences in races seen within the L2 cache versus a theoretical infinite-size cache in CORD and found little difference in the total accuracy [181]. Intuitively, racy accesses that happen closer together in time (and are thus still contained within the cache) may be the most important races because they are more likely to have their orders changed by timing perturbations. Zhou *et al.* summarize this intuition by in their report on HARD by stating that "this [cache-sized] detection window is usually enough because most races occur within a short window of execution" [236].

Problems 2, 3, and 4 can usually be mitigated through software mechanisms such as pinning threads to individual processors (2), reducing the maximum number of threads to match the number of cores (3), and disabling SMT (4) when performing race detection. Only the latter two are performed in this chapter. It is also important to note that caveat 2 was found to occur infrequently enough that, despite the potential problem, it was never observed in practice. The next step in building the demand-driven race detector is therefore a method of informing the software of these events.

Previous research by Horowitz *et al.* proposed hardware structures that solve this problem by making memory events available to the user [95, 96]. More recently, Nagarajan and Gupta proposed ECMon, a hardware mechanism that can take user-level interrupts when a program causes particular cache events [149]. ECMon also solves the first problem mentioned above by causing an interrupt whenever a cache line is evicted. In order to quickly take the subsequently large number of events, ECMon requires user-level interrupts.

While making every cache event available to the user is a fruitful area of work, it would also be nice to have a less intrusive hardware design. To this end, this chapter abandons the requirement of signaling the software on every cache event. Instead, as will be detailed in the next section, this work settles for observing a subset of cache events. Instead of enabling the tests for individual instructions, this system will also enable the detector for long periods of time, under the hypothesis that inter-thread sharing is often bursty. This has the added benefit of solving the last problem in the common case, as the detector will remain on after the false sharing occurs, often observing many subsequent true sharing events. Rather than a pure demand-driven data race detector, this design results in a best-effort demand-driven analysis that works much like a sampling analysis. The tool is enabled when it appears that sharing is happening and disabled when it is unlikely that races are occurring.

4.4.2 Performance Counters

Performance counters, a class of hardware devices contained in most modern microprocessors, can be programmed to count the number of times specified dynamic events occur within a processor. One of the most common counters, for example, is the number of instructions committed since the counter was enabled. When these counts are made available to software, they can be used to find and remove software inefficiencies [208].

While performance counters can easily track the number of events that occur within a processor, it is more difficult to use them to find which instruction causes a particular event. One common method is to set the initial value of a counter to a negative number, take an interrupt to the kernel when the counter overflows, and observe the state of the machine from within the interrupt handler. This does not always work for finding the exact state of the machine, as this interrupt may not be precise, and may thus show the state of the machine after an instruction that did not cause the event.

Intel processors since the Pentium 4 include a hardware facility called Precise Event Based Sampling (PEBS) that allows the hardware to automatically and precisely save information about instructions that cause performance events. For instance, PEBS can automatically store into a physical memory buffer the instruction pointer and register values of the operation that causes an event. When this memory buffer fills, the hardware will raise a precise interrupt, allowing the kernel to read out the data and make it available to analysis tools.

Recent Intel processors have also introduced a number of advanced performance counter events [58]; the most interesting for this work are the events that describe cache state. For example, in processors based on the Nehalem microarchitecture, the event MEM_UNCORE_RETIRED.OTHER_CORE_L2_HITM (which will be referred to as HITM for the remainder of this work) allows a core to count the number of times it loads a cache line that was stored in the modified state within another core's private L2 cache.

To build a dynamic sharing monitor that can inform software of inter-thread sharing events, it is possible to set up a performance counter to follow HITM events, and set the PEBS hardware to trigger when this counter overflows. This makes it possible to obtain information about the thread that caused the data sharing. By taking a PEBS interrupt when the HITM event occurs, a core is able to inform the software that a W$\rightarrow$R data sharing event has happened. This can be used, then, as the mechanism for informing our demand-driven race detector that data is being shared.

There are, unfortunately, further impediments to using the HITM performance counters as the signal to begin performing race detection.

1. Current HITM counters do not count reads-for-ownership that cause HITMs. This means it is impossible to directly observe W→W data sharing.
2. There is currently no performance counter event to directly observe R→W data sharing.
3. Current performance counters only count cache events that are caused by instructions. They cannot count when the hardware prefetchers load a cache line that is in the modified state in another processor's cache.

The effects of problems 1 and 2 on the accuracy of the demand-driven data race detector are quantified in Sections 4.6.3 and 4.6.4. Initial methods for mitigating these problems are also tested in that section. It is often possible to disable the hardware prefetcher with BIOS or UEFI settings, eliminating the third issue. This was not possible when performing the tests for the chapter, but no adverse effects were noticed in any of the benchmarks. The third problem is therefore not studied further in this work.

4.5 Demand-Driven Race Detector Design

The goal of creating a demand-driven race detector requires that we have hardware events that will inform the software of data sharing events. Though the cache-based performance counter events described in the previous section allow this by taking faults to the kernel whenever certain types of data sharing occur, these interrupts are too slow to be used to signal every memory operation that must be sent to through the race detector. Instead, these events are used to infer that the program under analysis has entered a region of code that is participating in data sharing. Working under the hypothesis that data sharing often happens in bursts, this is a signal to enable the software-based race detector for a period of time. The demand-driven analysis algorithm for this system, shown in Figure 4.5, is therefore:

1. Run the software under test with race detection disabled (but paying attention to synchronization operations) and HITM performance counters enabled.
2. Observe HITM performance events. If these occur while the processor is within the program's instruction space, enable the software race detector and disable the HITM performance counters. Synchronization functions such as pthread_mutex_lock that cause HITMs are observed but do not cause the race detector to start.
3. Run the software with data race detection enabled and keep track (with software

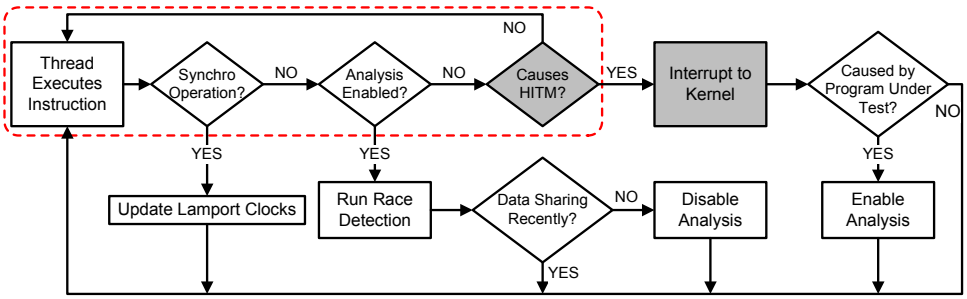


Figure 4.5 HITM-based Demand-Driven Data Race Detector. This algorithm uses hardware events (dark boxes) to watch for data sharing. If sharing is detected, the slow race detection is enabled until the software race detector finds no more data sharing. With no sharing, the system quickly goes through the steps within the outlined box.

mechanisms) of data sharing. If the race detector finds that the program is no longer in a section of code with data sharing, disable the race detector and return to step 1.

Enabling race detection for a length of time whenever the hardware indicates that the software is in a region with shared memory helps alleviate two of the limitations of the hardware sharing indicator. First, it reduces the number of interrupts, which can devastate the performance of a system if they occur too often. Second, it increases the chance of detecting W→W and R→W data races in the vicinity of detected W→R sharing, the two race types that the current hardware sharing indicator cannot detect. Because these races are likely to happen in regions of code that have other data sharing, they may be caught by the software race detector after it is enabled by a W→R sharing event.

When the performance counters are enabled in this design, a HITM event increments PMC0, the first performance counter. When this counter overflows, it arms the PEBS hardware, which will fire on (roughly) the next HITM event. When the PEBS hardware fires, it writes the current architectural state into a physical memory buffer and causes an interrupt to the kernel. The interrupt handler first checks that the software-under-test (SUT) was scheduled when taking the interrupt (as other processes or the kernel could have caused the event) before sending a signal to the race detector and resetting PMC0. The signal causes the race detector to access the performance counter driver to read the information that the PEBS hardware wrote into physical memory. It uses this information to decide whether the HITM was caused by the SUT or the detector itself. Finally, if the interrupt was caused by the SUT, the detector requests that the driver disable the counter and enable full race detection.

The race detector, when enabled, keeps track of inter-thread data sharing using software mechanisms such as those described by Sack *et al.* [190]. It does this in order to enable the previously detailed filters, where instructions working on thread-local data are not sent

through the race detection algorithm. If no inter-thread data sharing has taken place within some user-defined period of time, the software race detector will disable itself. Before allowing any further instructions to run, it also asks the driver to again enable the HITM performance counters. This puts the software back into the initial state where it runs at nearly full speed, with no race detection occurring. The particular timeout used in this work is dependent on the internal workings of the race detector. Broadly, the sharing check occurred every few thousand instructions.

As mentioned in Sections 4.4.1 and 4.4.2, this system is not guaranteed to find all races, as the caches may miss some data sharing events. However, this work agrees with the assertion of other researchers that it is better to have a race detector that is usable but slightly inaccurate than one that is unusable [190]. For some developers, the performance offered by a best-effort demand-driven race detector may be worth the chance of missing some races. At other times, they may choose to disable the demand-driven aspects and run the continuous-analysis detector. Both options are possible with this system.

4.6 Experimental Evaluation

This section details a series of experiments performed on a demand-driven data race detector built in a similar fashion to that described in the previous section. The two important questions addressed by these tests are the potential performance gains of a demand-driven data race detector, and the accuracy attainable using the less-than-ideal detector built using current hardware.

4.6.1 Experimental Setup

This system was built on top of the Intel Inspector XE race detector, a software-based happens-before race detector built using Pin [127]. The Inspector XE pintool performs few actions (such as updating Lamport clocks on synchronization points) when race detection is disabled, but there is still some slowdown due to the instrumentation overhead. When race detection is enabled, the tool checks every memory operation in some way and disables the performance counters to avoid unnecessary interrupts.

The experiments were performed on an Intel Core i7-870, a 2.93 GHz processor with 256KB of private L2 cache per core. This is normally a four-core processor, and SMT was disabled for these experiments. The test system was equipped with 8 GB of DDR3 DRAM and ran 64-bit SUSE Linux Enterprise Server 11.0, kernel version 2.6.27.19.

Table 4.1 The Concurrency Bugs from RADBench Used in this Work. These bugs were used to test the accuracy of our demand-driven race detector versus a system that analyzes every memory accesses. SpiderMonkey-0 is not described by Jalbert *et al.*, as it was removed in the final version of RADBench.

SpiderMonkey-0	A data race in the Mozilla SpiderMonkey JavaScript Engine, Mozilla bug 515403
SpiderMonkey-1	A data race and atomicity violation in the Mozilla SpiderMonkey JavaScript Engine, Mozilla bug 476934
SpiderMonkey-2	A data race in the Mozilla SpiderMonkey JavaScript Engine, Mozilla bug 478336
NSPR-1	Mutexes being incorrectly allocated (resulting in improper locking) in Mozilla NSPR, Mozilla bug 354593
Memcached-1	Thread-unsafe incrementing of global variables in *memcached*, Memcached bug 127
Apache-1	List in Apache *httpd* incorrectly locked, causing crashes. Apache bug 44402

Two benchmarks suites were used to ascertain the performance of this demand-driven data race detection system. The first, the Phoenix shared memory map-reduce benchmarks, represents programs that, while parallel, have little data sharing [186]. The other suite, PARSEC version 2.1, is a collection of parallel applications with a wide range of data access patterns, concurrency paradigms, and data sharing [24]. All benchmarks were run with four threads, and all except for *freqmine* were compiled with GCC 4.3 using pthreads. *Freqmine* used ICC 10.1 and was parallelized with OpenMP. The Phoenix benchmarks were given default inputs, while the PARSEC benchmarks used the simlarge input set. Each benchmark was run multiple times to increase statistical confidence, and the first run of every test was discarded in order to preload data from the hard disk.

The Phoenix and PARSEC suites were also run to compare the race detection accuracy of the demand-driven analysis tool versus the original continuous-analysis Inspector XE. In addition, the accuracy of this system was tested with a pre-release version of RADBench by Jalbert *et al.* [103]. The particular benchmarks from that suite that were used are listed in Table 4.1. Because these tests were only used to test the accuracy of the demand-driven data race detector, the two deadlock benchmarks (NSPR-2 and NSPR-3) were not used. Similarly, Apache-2 was not tested because it was an atomicity violation only, and the race detector in Inspector XE would not find it. A SpiderMonkey bug that was not included in the final version of the suite, SpiderMonkey-0, was tested because these benchmarks were run before the final version of RADBench was released. Finally, the Chrome benchmarks were not run, as they did not correctly compile within the particular version of Linux used for these tests.

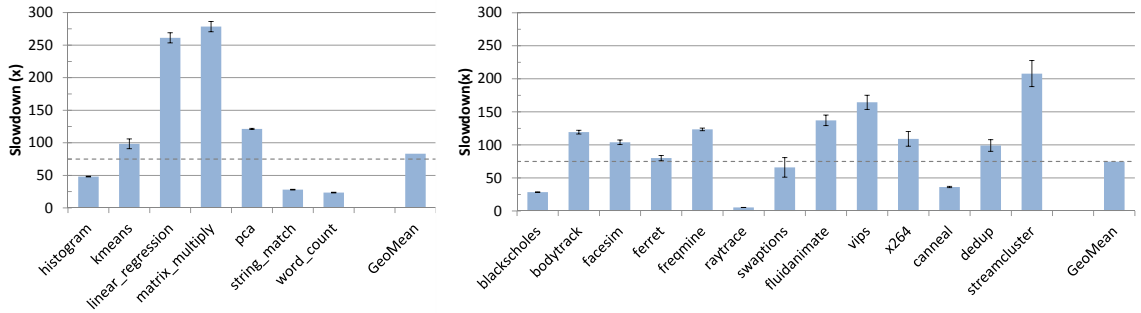
In all cases, for both the continuous and demand-driven tools, the data race detector was configured to search for races but not cross-thread stack accesses or deadlocks. The software filtering available within Inspector XE was enabled, meaning that it would not run the full data race detection algorithm on a variable until it was accessed by at least two threads, and one of those threads performed a write. No call stack information was kept for debugging purposes for any of the performance-related benchmarks.

4.6.2 Performance Improvement

The initial tests quantify the performance improvements that a demand-driven data race detector could yield over a continuous-analysis detector. Each benchmark was first run with the original race detector, configured as previously described, which yielded the baseline slowdowns for continuous-analysis as shown in Figure 4.6. The slowdowns between the benchmarks range from 5.3× for *raytrace* in the PARSEC suite to 278× for *matrix_multiply* in the Phoenix suite. The former spends most of its time in single-threaded code where the race detector causes little slowdown, while the latter is slowed heavily due to the software filtering algorithm.

On the whole, the Phoenix benchmark ran 83× slower while inside the race detector than on the bare hardware, as shown in Figure 4.6a. While all of these algorithmic kernels utilized similar forms of data parallelism in their concurrent sections, the slowdowns caused by the race detector can differ by an order of magnitude. Because these benchmarks are written in a map-reduce manner (where one thread assigns independent work to all the others), the software filtering stops most of the memory accesses from going through the slow race detection algorithm. The benefit of this can be seen in benchmarks such as *string_match* and *word_count*, where overheads are relatively low (28× and 23.5×, respectively). Unfortunately, the memory access filtering can itself cause high overheads, as can be seen in *linear_regression* and *matrix_multiply*. These benchmarks also have few shared memory locations, but their particular memory access patterns bring out inefficiencies in the filtering algorithm. As such, even though the memory operations are not sent through the slow race detector, the filtering mechanism itself causes overheads of 261× and 278×, respectively.

The PARSEC benchmark suite contains a collection of large programs with many different styles of parallelism. These differences in the methods of concurrency can be inferred from Figure 4.6b. While benchmarks with little communication, such as *blackscholes* and *raytrace* have little overhead (28.4× and 5.3×, respectively), the program *dedup* has a significant amount of sharing and runs much slower (99× slowdown). Many of these benchmarks spend a large amount of their time either running in the race detection algorithm



(a) Phoenix Suite

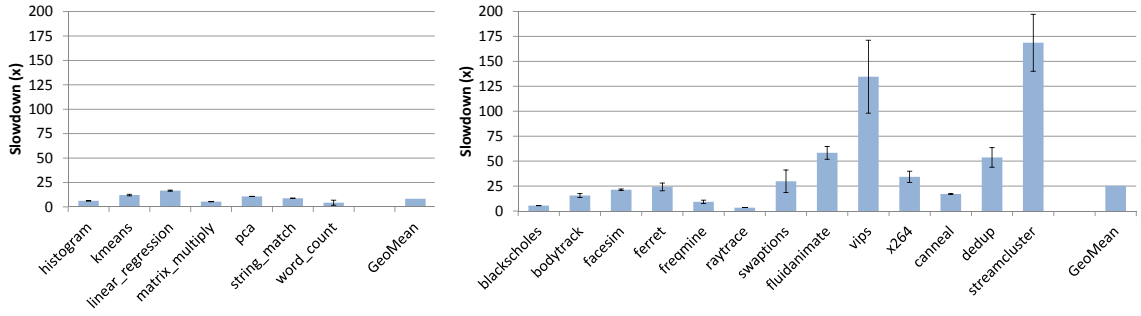
(b) PARSEC Suite

Figure 4.6 Continuous-Analysis Data Race Detection Slowdowns. Bars represent the mean slowdown; error bars represent 95% confidence intervals. The dashed line is illustrative of a 75× slowdown. These benchmarks see a range of slowdowns, averaging **(a)** 83× and **(b)** 75×. Much of the time in the high-overhead benchmarks is spent within the software filtering mechanism or the Lamport clock algorithm.

or going through the software filtering mechanism. The geometric mean of the slowdowns for the PARSEC suite is 74.7×.

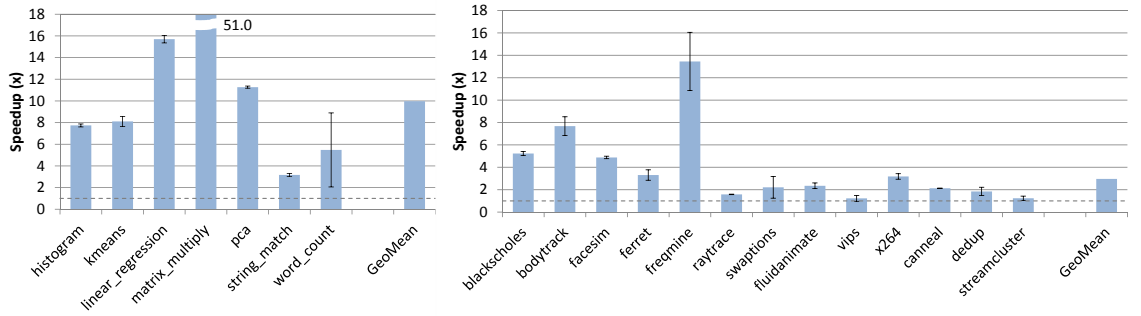
The same benchmarks were run under the HITM-based demand-driven detection system. For these benchmarks, HITMs that occurred in some library calls (such as malloc and free) were ignored because they had needless sharing that should not cause data races. The Performance differences between these experiments and the baseline are due to the demand-driven system disabling the race detector and software filters when they are not needed. The results of these experiments are shown in Figures 4.7 and 4.8. The former shows the slowdowns that the demand-driven data race detector causes for each benchmark. This is the same data as shown in Figure 4.6 for the continuous-analysis detector. The latter figure, 4.8 instead shows the performance improvements that the demand-driven race detector has over the continuous-analysis detector.

Figure 4.8a shows the performance improvements yielded by the HITM-based race detector in the Phoenix suite. The average slowdown dropped from 83× to 8.3×, a 10× performance improvement. This can be seen most dramatically in the *matrix_multiply* benchmark. In this case, the hardware is able to observe that very little data sharing actually occurs, and it keeps the race detector disabled for the majority of the benchmark. With the race detector disabled, no memory operations are sent through the software filter, yielding a 51× performance improvement. This reduction of the filter overhead is also seen throughout the other benchmarks, including those that originally had relatively little slowdown. The slowdown of the *word_count* benchmark, for instance, is reduced to 4.3×.



(a) Phoenix Suite **(b)** PARSEC Suite

Figure 4.7 Total Overhead of the Demand-Driven Data Race Detector. Bars represent the mean slowdown (note that the Y-Axis is different than the Y-Axis on Figure 4.6; error bars represent 95% confidence intervals. **(a)** The Phoenix benchmark suite sees a significant performance improvement, with the geometric mean of the slowdowns falling to 8.3×. **(b)** Because the PARSEC suite has more data sharing in some of its applications, its slowdowns only fall to 25×.



(a) Phoenix Suite **(b)** PARSEC Suite

Figure 4.8 Speedups of Demand-Driven over Continuous-Analysis Data Race Detection. Bars represent the mean performance gain; error bars represent 95% confidence intervals. The dashed line is the baseline performance of the continuous race detector. **(a)** The Phoenix map-reduce benchmarks see large speedups because they contain little data sharing between the worker threads. **(b)** The PARSEC suite contains more data sharing, but the average speedup is still 3×.

The concurrency model of the Phoenix benchmark suite is well suited for our method of demand-driven race detection. Each benchmark shares very little data, so the vast majority of memory accesses need not be sent through the race detector. Without the hardware sharing detector, these memory operations would still be sent through some type of analysis to determine if they should be checked for races.

Figure 4.8b details the performance improvements of the PARSEC benchmark suite. The geometric mean of the speedups in these benchmarks is 3×, reducing the slowdown from 74.7× to 25.2×. *Freqmine* sees the highest performance gains (13.5×), as it is an OpenMP data parallel program, similar to the benchmarks in the Phoenix suite. Other benchmarks,

such as *x264* and *canneal* see less performance gain (3.2× and 2.1×) because they have a significant amount of inter-thread communication that must be sent through the race detector. A few benchmarks (e.g. *vips* and *streamcluster*) do not contain a large amount of sharing, but still do not gain much performance (22% and 23%, respectively). This appears to occur primarily because what little sharing the programs perform causes the race detector to run for a period of time, exit, and quickly reenter on the next sharing event or falsely shared cache line. The speedups of all the benchmarks are dampened by the overheads of going to the kernel to deal with performance monitor interrupts, but it is especially apparent in these benchmarks that have a small amount of data sharing.

Taken together, these benchmarks verify that a demand-based race detector can yield higher performance than a continuous-analysis detector, even when the latter uses advanced software filtering techniques. Programs that contain a high amount of data sharing gain some performance benefits, as there are bound to be sections of code that are not operating on shared data. Meanwhile, programs with large sections of code that do not share data can be analyzed much faster.

4.6.3 Accuracy of Demand-Driven Data Race Detection

The next question in regards to the demand-driven data race detector is its accuracy compared to the slower continuous-analysis data race detector. If demand-driven analysis was faster but found no errors, it would be of little use.

The data races found in the Phoenix, PARSEC, and RADBench suites using the demand-driven race detector were compared against the data races found with the continuous-detection system. These races include both benign races (such as threads checking a variable in a racy way before attempting to acquire a lock and rechecking the variable) and actual data races. Every observed race is reported because the difference is meaningless to the tool; it is up to the programmer to designate a race either benign or real, and some have even argued that there is no such thing as a benign race [25].

The accuracy results are presented in Table 4.2, which only lists benchmarks with data races detected by the continuous-analysis tool. Demand-driven analysis misses no W→R races, the type of sharing that the hardware performance counters can observe. It also finds most W→W and R→W races, verifying the theory that, quite often, W→R sharing is an indicator of shared sections of the program. In total, the demand-driven data race detector observes nearly 80% of the races visible to the continuous analysis race detector.

It is also of note that among the races observed with the demand-driven race detector, two were true data races that had not been previously reported. *Facesim* contained a W→R data

Table 4.2 Accuracy of the Demand-Driven Data Race Detector. The demand-driven race detector finds most of the races observed by the continuous race detector (including previously unknown races in *facesim* and *freqmine*). The W→W races that are missed are due to the lack of HITMs on reads-for-ownership, while cache size limitations cause the missed races in *facesim*.

	kmeans	**facesim**	**ferret**	**freqmine**	**vips**
W→W	1/1 (100%)	0/1 (0%)	-	-	1/1 (100%)
R→W	-	0/1 (0%)	2/2 (100%)	2/2 (100%)	1/1 (100%)
W→R	-	2/2 (100%)	1/1 (100%)	2/2 (100%)	1/1 (100%)

	x264	**streamcluster**	**SpiderMonkey-0**	**SpiderMonkey-1**
W→W	-	0/1 (0%)	0/9 (0%)	0/1 (0%)
R→W	3/3 (100%)	1/1 (100%)	3/3 (100%)	-
W→R	3/3 (100%)	1/1 (100%)	8/8 (100%)	1/1 (100%)
	SpiderMonkey-2	**NSPR-1**	**Memcached-1**	**Apache-1**
W→W	0/1 (0%)	3/3 (100%)	-	1/1 (100%)
R→W	1/1 (100%)	1/1 (100%)	1/1 (100%)	7/7 (100%)
W→R	2/2 (100%)	4/4 (100%)	-	2/2 (100%)
Total	55/69 (79.7%)			

race due to a mistake in the code that did not appropriately create thread-local copies of the working data. *Freqmine* (the OpenMP benchmark with a high speedup) contained a W→R error due to a missing OMP critical statement. Both bugs were reported to the PARSEC developers and confirmed, and patches were developed to be deployed in a future version of PARSEC. One of the data races in *x264* was also a true data race that had previously been reported to the *x264* developers and patched.

Anecdotally, debugging these races was easier than normal thanks to the increased speed of the demand-driven data race detector. The multiple runs performed to pinpoint the locations and call-stacks of the error in *freqmine*, for instance, were completed before the continuous-detection run completed for the first time.

4.6.4 Races Missed by the Demand-Driven Data Race Detector

Despite these benefits, the demand-driven tool misses 20% of the data races that the continuous-analysis tool finds. The first reason for these inaccuracies is because executing a read-for-ownership (RFO) on a cache line stored in the modified state within another cache does not increment the HITM performance counter. This is why the W→W race in *streamcluster* is not observed, for instance; there is no W→R sharing in the section of code that causes this race.

Additionally, the limited size of the L2 cache makes it difficult to observe the remaining races in *facesim*. The initial write and subsequent racy accesses are far apart (with a large

Table 4.3 Change in Detection Accuracy with HITMs on RFOs. By taking HITM events on the innocuous loads before the stores to shared variables, we are able to observe the previously missed W→W races, though we still miss the races in *facesim* due to cache size limitations. This table lists only the benchmarks that previously had less than 100% accuracy.

	facesim	**streamcluster**	**SpiderMonkey-0**	**SpiderMonkey-1**	**SpiderMonkey-2**
W→W	0/1 (0%)	1/1 (100%)	9/9 (100%)	1/1 (100%)	1/1 (100%)
R→W	0/1 (0%)	1/1 (100%)	3/3 (100%)	-	1/1 (100%)
Total	67/69 (97%)				

amount of data written by all threads in between), and thus the modified line is evicted from the cache before it can cause a HITM event.

4.6.5 Observing more W→W Data Sharing

If it were possible to detect RFO accesses that cause HITM events, it should be possible to detect some W→W races that the current demand-driven tool misses. One way to do this would be to modify the HITM performance counter event in future processors to also count stores that cause RFO HITMs.

In order to test this solution on the existing system, the race detection pintool was modified to insert an innocuous load before every store. This load should cause a HITM performance event at the same location that would cause an event on a processor that counted RFO HITMs. The accuracy tests were rerun and it was observed that this indeed made it possible to detect the missing W→W races. The total detection accuracy is listed in Table 4.3.

In total, this modification raises the accuracy of our demand-driven tool from 80% to 97% in these tests. The only extra overhead would come from the additional time spent in the race detector from W→W sharing events if it were possible to detect these events in hardware. Instead, inserting the extra loads using binary instrumentation slows the program an additional 5–130% of its original speed. This reduces the performance gains seen by the demand-driven race detector by less than 5% in all cases (*e.g.* the speedup seen by *freqmine* is reduced from 13.5× to 12.9×).

4.6.6 Negating the Limited Cache Size

Even when watching RFO HITM events, there are still unobserved races that remain in *facesim*. As mentioned, these data races are missed because the data from the first write

is evicted from the L2 cache before the second thread's read or write. This is an inherent limitation of the cache-event-based race detection architecture. There are a number of ways that this shortcoming might be mitigated.

1. Ignore the problem, as races that take place far apart (such that the cache has enough time to evict the modified line) may be less likely to affect the program's output.
2. Increase the size of the monitored L2 cache.
3. Take events when data is evicted from the cache, as ECMon does [149].
4. Perturb the schedules of the threads such that the sections that may race are put closer to one another by, for instance, placing `sleep()` calls throughout the code to perturb the regular schedules [177]. This is an active area of research in concurrency bug analysis [64].

This work focuses on method 4. It was found that by modifying the scheduling of the threads, it was possible to cause the racy regions to overlap, making it possible to detect both remaining races in *facesim*. This process, however, was labor-intensive in its current form, and a more systematic method (such as those in the literature) may help.

In general, through a combination of techniques, the demand-driven data race detector was able to observe all data races that the continuous-analysis Inspector XE race detector observed.

4.7 Related Work

4.7.1 Data Race Detection

Data races are common concurrency bugs that affect many parallel programs. Netzer and Miller formalized the definitions of both feasible data races (those that could happen in some possible program ordering) and apparent data races (those that can be inferred based on the explicitly observed synchronization of the program) [168]. This work focused on the latter.

There have been numerous works dedicated to building tools that find data races. This chapter focused on dynamic techniques, which can be divided into post-mortem and online (or on-the-fly) debugging. The former will record what data sharing it observes, and if an error occurs, this information can be used to help find the bug [139]. The latter will find bugs in while running the original program [195]. Once again, this paper focused on the latter.

This paper also focuses on Intel Inspector XE [13, 14], but there are also other commercial and open-source race detectors in use. Examples include Valgrind's Helgrind tool [104], Google ThreadSanitizer [199], IBM Multicore SDK [182], and Oracle (née Sun) Thread Analyzer [214]. In general, these tools all suffer from high overheads. For example, we took measurements that put Helgrind at about the same order of magnitude as the continuous-analysis Inspector XE. The demand-driven analysis technique presented in this chapter should be amenable to many, if not all, of these race detectors.

4.7.2 Software Acceleration Methods

Dynamic data race detectors can have extremely high overheads, so researchers have looked at software mechanisms for improving their performance. Choi *et al.* use static analysis methods to find memory operations that are subsumed by more important accesses [43], while Sack *et al.* looked into ways of filtering superfluous race detection in software [190]. FastTrack shows that by carefully using scalar clocks instead of vector clocks, a happens-before race detector can be made much faster without significant loss of accuracy [69]. In a different vein, previously discussed techniques for accelerating dynamic analyses, such as Umbra [234], would also work to accelerate data race detection.

Though these works show large performance improvements, current dynamic data race detectors are still quite slow. As shown in this chapter, a demand-driven data race detector is able to achieve much higher performance than a commercial race detection tool that uses a number of the mechanisms present in the literature.

As has been discussed throughout this dissertation, one novel way of increasing dynamic analysis performance is to perform sampling. Section 2.1.3 discussed methods of performing sampling to accelerate concurrency analyses. In summary, LiteRace [131], PACER [26], and DataCollider [65] present methods of performing data race detection on a user-controllable subset of the dynamic memory accesses of a program. In doing so, they are able to make the overheads user-controlled at the expense of missing some data races. The choice to perform sampling is orthogonal to the work presented in this chapter, however, as both techniques could be used at the same time. The higher performance brought about through demand-driven analysis would allow a sampling data race detector to see fewer false negatives under a particular performance threshold. In essence, this chapter would allow a demand-driven sampling data race detector, much like the demand-driven sampling taint analysis tool studied in Chapter 2.

4.7.3 Hardware Race Detection

Hardware race detection systems take advantage of the fact that cache events are related to data sharing by tracking these events and using them as signals to other hardware mechanisms. One popular area of research in hardware-based data race detection is hardware race recorders, which are used for post-mortem data race debugging [12]. Systems such as FDR record enough information about inter-processor cache events (and other metrics) to backing store that it is possible to deterministically replay the multi-threaded software execution afterwards to observe the bug occurring again [97, 151, 230]. These systems are unable to perform online race detection.

There has also been work in performing dynamic race detection within hardware. Min and Choi described one of the earliest works in this area; their system performs happens-before analysis on the cache line level, and uses inter-processor cache events to detect sharing [142]. Their system came with a number of limitations, however, as they assume that cache evictions will be caught by the OS and virtualized (a slow proposition), and that shared variables will be contained on their own cache line. More recently, CORD and HARD described hardware happens-before and lockset race detectors, respectively [181, 236]. The former uses inter-processor cache events to detect data sharing and studies the accuracy loss of ignoring races that happen outside of the cache. The latter uses bloom filters to detect when a variable has associated lockset data that is stored alongside the cache (and in main memory). Muzahid *et al.* describe a signature-based mechanism, SigRace, where accesses that conflict in a lossy filter will cause the hardware to roll back execution and enable hardware race detection [147].

Despite their ability to find data races while running the original application at (close to) full speed, these hardware race detectors require application-specific additions to the processor's pipeline and cache system. Unlike the solution proposed in this chapter, these systems do not currently exist in processors, though some of their additions may be useful in making a more accurate demand-driven software race detector.

4.7.4 Observing Hardware Events

Other researchers have proposed hardware structures that make memory events available to the user. Horowitz *et al.* described Informing Memory Operations, or memory operations that cause execution to branch to a user-defined handler whenever a cache miss occurs [95, 96]. Nagarajan and Gupta later proposed ECMon, a hardware mechanism that can take user-level interrupts when a program causes particular cache events [149]. They mention

that these events could be used to perform lockset-based race detection, though they do not give any details of an implementation. Regardless, both IMO and ECMon require user-level interrupts, as their designers expect to perform actions on every cache or memory event. In comparison, this work describes a way to perform demand-driven detection while requiring many fewer hardware changes and taking many fewer interrupts.

This demand-driven data race detector was built using the performance counter system that already exists on commercial microprocessors. Post-silicon verification of hardware operation and offline software optimization are the most common uses of these performance counters [208], but a number of researchers have looked into new ways of using these devices. Because processor activity can be monitored using these counters, Singh *et al.* use them to estimate power usage in a processor [205]. Additionally, Singh *et al.* and Chen *et al.* both propose using performance counters as input to the process scheduler in an effort to better utilize processor resources [36, 205].

4.8 Chapter Conclusion

This chapter described a method of building a demand-driven data race detector using commodity hardware. Because happens-before race detectors find apparent data races by observing data sharing events that are not appropriately protected, there is no need to execute the race detection analyses if there is no data sharing. This demand-driven tool therefore does not perform race detection until a hardware-based indicator informs it that the program has entered a section that is potentially sharing data. This sharing indicator was then built using the performance counter facilities on current consumer processors and was used to build a best-effort demand-driven data race detector.

Demand-driven race detection can significantly increase the performance of software race detectors. The above-described system was able to achieve a mean performance improvement of 10× in the Phoenix benchmark suite because its programs have little data sharing, and a 3× performance improvement in the PARSEC suite. While using cache events to perform demand-driven analysis can cause the race detector to miss some races, a number of solutions exist to combat these inaccuracies. The demand-driven detector was able to catch 97% of the races observed using a continuous-analysis detector, two of which were new bugs in the PARSEC suite that have been reported to the developers.

The demand-driven analysis tool in this chapter is similar to that used as the baseline for the software-based dataflow sampling system described in Chapter 2. The primary difference lies in the hardware mechanism used to enable the analysis tool on demand. The

sharing indicator described in this chapter searches for a different type of event than the virtual memory watchpoints used for the previous tool. Rather than looking for accesses to a particular subset of memory locations, the performance counters used to build the demand-driven data race detector instead directly looked for inter-thread sharing of data. As an added benefit, the detection is done on a finer granularity (64-byte cache lines vs. 4KB pages), leading to less false faults that superfluously enable the analysis tool.

This sharing indicator has some similarities to the Testudo hardware discussed in Chapter 3. Both use on-chip caches to store some semblance of meta-data, which can then be used to enable software detection algorithms. While Testudo uses a dedicated cache for this, the performance counters piggyback on the cache coherency protocols that already exist on the host processor. Testudo uses a randomized replacement policy to deal with the inability of the small on-chip cache to hold all interesting meta-values. This system instead limits itself to enabling the software analysis only when limited on-chip storage *can* see it. The software analysis tool, which is enabled for long periods of time after the performance counter system turns it on, allows the analysis to see more meta-values than can be stored on-chip.

The next chapter details a generalized hardware mechanism that can be used to accelerate a wide range of analyses. The tools studied in this dissertation (and others not yet discussed) share a need for fine-grained memory monitoring. To that end, the next chapter describes hardware that can make a virtually unlimited number of fine-grained hardware watchpoints available to software. In this way, it is possible to speed up numerous software tools without needing specialized hardware for each one. This, when combined with the other techniques discussed in this dissertation, can greatly increase the amount and quality of analyses available to software developers.

Chapter 5

Hardware Support for Unlimited Watchpoints

> You can observe a lot by watching.
>
> Yogi Berra

5.1 Introduction

We have studied numerous software analysis tools over the course of this dissertation. Chapters 2 and 3 focused on dynamic dataflow analyses, while Chapter 4 moved into data race detection. This is but a small subset of the types of tools that software developers use to improve their applications. Beyond even the online correctness tests discussed in Section 1.4.2 exists mechanisms such as speculative multithreading [109] and deterministic execution [21]. These systems have different goals than dynamic software analyses, but they are still related to the other dynamic analyses since they observe a program's actions as it runs.

While dynamic software analysis mechanisms are powerful, the constant refrain of this dissertation holds true: they are slow. So slow, in fact, that the tests that we've studied in this dissertation see little adoption outside of the development lab. Even worse, while the systems like speculative multithreading and deterministic execution are designed to make software faster and more robust when running on user's computers, the overhead of doing this work relegates them primarily to the research community [31].

Hardware acceleration is one way of solving this problem. There have been proposals for application-specific accelerators for basically every tool discussed in this dissertation: bounds checkers [59], taint analyses [38, 50, 53, 219], data race detection [142, 181, 236], speculative multithreading [237], and deterministic execution engines [60]. This list shows

that it's possible to make nearly every dynamic analysis tool fast, given appropriate hardware support.

It also hints at the reason why no modern processors contain any of these accelerators: none offer a comprehensive way of accelerating many tools. As a consequence, an architect is left with the choice between adding accelerators for every analysis (leading to a Franken-chip with a nightmare of backwards compatibility issues), choosing some subset of "important" analyses (and potentially guessing wrong), or throwing her hands into the air in frustration and letting software people worry about it.

In summary, these accelerators are too specific to meet the stringent requirements needed to be integrated into a modern commercial microprocessor. Their benefits are too narrowly defined in light of their area, design, and verification costs, relegating most of these ideas to the research literature. The only device to make its way into silicon is transactional memory [46, 101, 87], and even then, it is as stripped down best-effort mechanisms instead of the robust, complex designed described by researchers.

Commercial processors tend to favor generic solutions, where costs can be amortized across multiple uses. As an example, the performance counters available on modern microprocessors are used to find performance-degrading hotspots in software [208], but they are also used during hardware bring-up to identify correctness issues [233]. While Sun added hardware into their prototype Rock processor in order to support transactional memory [34], the same mechanisms were also used to support runahead execution and speculative multithreading [35]. Intel's engineers have also been looking for other uses for their upcoming transactional memory support, as evidenced by recently released patent applications [185].

The work already presented in this dissertation has been a mix of application-specific and general techniques. The sampling system described in Chapter 2 required no additional hardware support and worked for a number of analyses, but as the subsequent chapters detailed, it has its limitations. The dataflow sampling hardware additions shown in Chapter 3 work for multiple software analyses (as described in Section 3.5.4), but require application-specific hardware changes in the pipeline. That hardware only worked on dataflow analyses and *required* sampling in order to function. The data race detection accelerator in Chapter 4 needed no extra hardware (after all, performance counters already exist on every modern microprocessor), but the technique itself was application-specific. HITM events are useful for data race detection, but do little to speed up other tests.

In contrast, this chapter will describe the design of a generic hardware accelerator that should work on many software tools. In essence, this is the culmination of the work done in the previous chapters. The end goal is a piece of hardware that is simple enough to implement in hardware but powerful enough to accelerate many important software tools. If

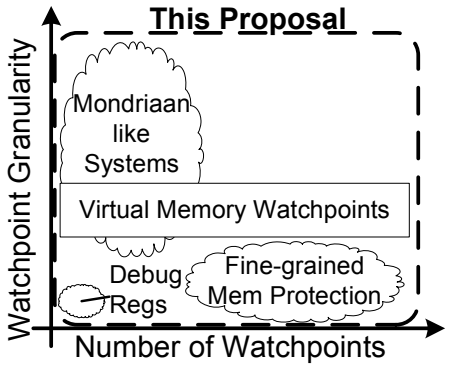


Figure 5.1 Qualitative Comparison of Existing Watchpoint Systems. Watchpoint registers are too few in number. VM's granularity is too coarse. Mondriaan-like systems cannot quickly change many watchpoints, and other systems are often only useful for small regions because they rely on holding each watched byte in hardware.

this is the case, a similar design may find its way into future consumer processors, making it easier to distribute software analyses to users at low runtime overheads.

5.1.1 Contributions of This Chapter

Many runtime analysis systems perform actions on shadow values while the program executes. In taint analysis, for example, each location in memory has a shadow value that marks it as trusted or untrusted. Ho *et al.* showed that using the virtual memory system to look for tainted values could greatly accelerate their taint analysis tool, and Chapter 4 demonstrated that checking read/write sets using software routines was 3-10$\times$ slower than using hardware to do the same [79].

Perhaps, then, a general software acceleration mechanism should alert programmers when specified locations in memory are being accessed. In other words, having the ability to set a large number of data watchpoints may benefit a wide range of analyses.

This chapter makes a case for the hardware-supported ability to set a virtually unlimited number of fine-grained data watchpoints. It will show that watchpoints can accelerate numerous software tools and that they are more general than the application-specific hardware often touted in the literature.

This is an improvement over existing memory-watching and fine-grained protection mechanisms, as qualitatively illustrated in Figure 5.1. Hardware watchpoint registers are too limited in number for the advanced systems we wish to accelerate, while virtual memory watchpoints, which are not constrained by hardware resources, are limited by the coarse granularity of pages. Finally, a number of proposals for fine-grained hardware memory pro-

tection and tagged memory systems exist in the literature, but they too are not general enough. Systems such as Mondriaan Memory Protection [229] (referred to as Mondriaan-like systems by Witchel [228]) are not designed for many watchpoints that frequently change. Other fine-grained protection designed, such as iWatcher [235] and MemTracker [221], make it easier to frequently change watchpoints, but suffer from the inability to set watchpoints of vastly different granularities. Watchpoints covering gigabytes of memory require setting fine-grained protection on all the composite data, potentially causing immense slowdowns.

This chapter presents a mechanism that avoids these limitations. This hardware/software hybrid watchpoint system stores per-thread virtual address watchpoints in main memory, avoiding hardware limitations on the total number of watchpoints. It then makes use of two on-chip caches to hold these watchpoints closer to the pipeline. The first is a modified version of the range cache proposed by Tiwari *et al.* [216], which efficiently encodes continuous regions of watched memory. The second is a bitmapped lookaside cache, which increases the number of cached watchpoint ranges when they are small, as the range cache can potentially have a very small reach in this case.

By combining these mechanisms with the abilities to take fast watchpoint faults and set watchpoints with user-level instructions, we can greatly accelerate tools that work on shadow data while remaining general enough to be useful for other memory-watching tasks. Simulated results show, for instance, that a data race detector built using this technique can check read/write sets up to 9× faster than one built entirely with binary instrumentation and 3× faster than one using other fine-grained memory protection systems. These performance numbers are similar to those shown in Chapter 4, but unlike that chapter's performance counter mechanism, watchpoints are also useful for other analyses. This chapter also demonstrates that watchpoints can accelerate tools such as taint analysis and deterministic execution engines.

This chapter, much of which was originally presented in the 2012 paper "A Case for Unlimited Watchpoints" [81], makes the following novel contributions:

- It describes hardware that allows software to set a **virtually unlimited number of byte-accurate watchpoints**.
- It studies numerous dynamic software analysis tools and show how they can utilize watchpoints to **run more accurately and much faster**.
- It demonstrates that **this design performs better on a wide range of tools and applications** than other state-of-the-art memory monitoring technologies.

Section 5.2 details the design of this watchpoint hardware, while Section 5.3 discusses

Table 5.1 Hardware Watchpoint Support in Modern ISAs. Most ISAs support a small number of hardware-assisted watchpoints. While they can reduce debugger overheads, their limited numbers and small reach are usually inadequate for more complex tools.

ISA	#	Known As	Possible Size
x86 [-64]	4	Debug Register	1, 2, 4, [8] bytes
ARMv7	16	Debug Register	1-8 bytes or up to 2GB using low-order masking
Embedded Power ISA (Book III-E)	2	Data Address Compare	1 byte **or** a 64-bit address with any bit masked **or** a range up to 2^{64} bytes (requires both registers)
Itanium	4	Data Breakpoint Register	1 byte to 64PB using low-order masking
MIPS32/64	8	WatchLo/Hi	8 bytes, naturally aligned, 8B-2KB using low-order masking
Power ISA (Book III-S)	1	Data Address Breakpoint Register	1-8 bytes
SPARC (T2 Supplement)	2	Watchpoint Register	Any number of bytes within an 8-byte naturally aligned word
z/Architecture	1	Program Event Recording	Read-only Range up to 2^{64} bytes

software systems that could be built with it. Section 5.4 compares this system to previous fine-grain memory protection works while running a variety of software analysis tasks. Finally, Section 5.5 reviews related works, and the chapter is concluded in Section 5.6.

5.2 Fast Unlimited Watchpoints

This section describes a system that allows software to set a virtually unlimited number of byte-accurate watchpoints. We first review existing watchpoint hardware and list what properties are needed to effectively accelerate a variety of software. We then present a design that meets these needs.

5.2.1 Existing HW Watchpoint Support

Watchpoints, also known as data breakpoints, are debugging mechanisms that allow a developer to demarcate memory regions and take interrupts whenever they are accessed [106, 136]. This ability is extremely useful when attempting to hunt down software errors and is therefore included in most debuggers. Using software to check each memory access can cause severe slowdowns, so most processors include some form of hardware support for watchpoints.

As Wahbe discussed, existing support can be broken down into specialized hardware watchpoint mechanisms and virtual memory [223]. The first commonly takes the form of watchpoint registers that hold individual addresses and raise exceptions when these addresses

are touched. Unfortunately, as Table 5.1 shows, no modern ISA offers more than a small handful of these registers. This limited number and small reach makes them difficult (if not impossible) to use for many analyses [42].

The second method marks pages containing watched data as unavailable or read-only in the virtual memory system. The kernel then checks the offending address against a list of byte- (or word-) accurate watchpoints during each page fault. Though this has been implemented on existing processors [19, 196], it has a number of restrictions that limit its usefulness, as this dissertation has previously alluded to.

For instance, individual threads within a process cannot easily have different VM watchpoints. This makes this type of watchpoint difficult to use for multi-threaded analyses [174]. The large size of pages also reduces the effectiveness of virtual memory watchpoints. Faults taken when accessing unwatched data on pages that also contain watched data can result in unacceptable performance overheads.

ECC memory can be co-opted to set watchpoints at a finer granularity than virtual memory pages [183, 194]. By setting a value in memory with ECC enabled, then disabling ECC, writing a scrambled version of the data into the same location, and finally re-enabling error correction, it is possible to take a fault whenever a memory line with a watched value is accessed. However, changing watchpoints is extremely slow in such a system, as it taking watchpoint faults. Because of the high overheads and limited availability of this technique, it is not explored further in this dissertation.

In sum total, existing hardware is inadequate to support the varied needs of a wide range of dynamic analysis tools.

5.2.2 Unlimited Watchpoint Requirements

While this tells us what current systems do *not* offer, we must still answer the question of what a watchpoint system *should* offer. Section 5.3 will detail watchpoint-based algorithms for numerous dynamic analysis systems, but in the vein of Appel and Li's paper on virtual memory primitives [7], this section first lists the properties which hardware should make available to these tools:

- **Large number**: Some systems watch gigabytes of data to observe program behavior. If there are a limited number of watchpoints, as is the case with existing watchpoint registers or the Testudo hardware of Chapter 3, it may not be possible to perform complete watchpoint-assisted analyses.

- **Byte/word granularity**: Many tools use watchpoints at a very fine granularity to reduce false faults. As granularity coarsens, software tools must make efforts to deal with this unwanted overhead. As an example of this, the sampling system from Chapter 2 used page-grained watchpoints, but used sampling to deal with overheads. Some of this overhead was impossible to remove, as shown in Figure 2.14a.
- **Fast fault handler**: Some applications take many faults, so this would greatly increase their performance. The HITM-based race detector from Chapter 4 had to deal with slow kernel-level faults. The tool remained enabled (and thus running slowly) when it technically did not need to be in order to amortize kernel fault overheads.
- **Fast watchpoint changes**: Numerous tools frequently change watchpoints in response to the program's actions. If every modification took hundreds of cycles, the tool's operation would need to be modified (perhaps weakening the analysis) to reduce the number of changes.
- **Per-thread**: Separate watchpoints on threads within a single process would allow tools to use watchpoints in parallel programs without taking false faults.
- **Set ranges**: Many tools require the ability to watch large ranges of addresses without needing to mark every component byte individually. This can significantly reduce the overhead of changing watchpoints, even when the raw number of changes is low.
- **Break ranges**: Similar to, yet distinct from, setting ranges, it is important to be able to quickly remove sections in the middle of ranges without rewriting every byte's watchpoint. Some existing hardware memory protection systems make it easy to set ranges, but breaking them apart devolves into changing each internal memory location.

It is important to note that these are the ideal abilities of a hardware watchpoint system. As the other chapters in this dissertation (and many other dynamic analysis works) show, it is entirely possible to utilize non-ideal hardware to make some tools better. However, because we wish to show that hardware watchpoint systems have large future potential, this chapter focuses on building an ideal design. The tradeoffs required for putting such a design into a modern microprocessor are left for future work.

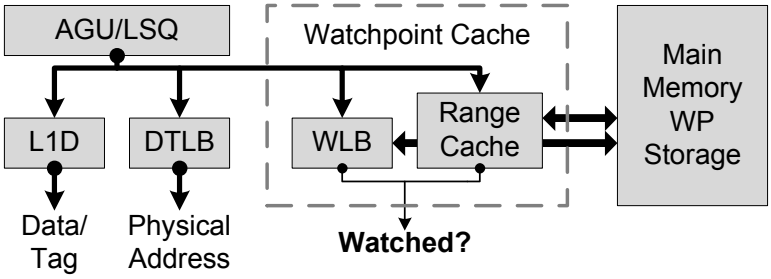


Figure 5.2 Watchpoint Unit in the Pipeline. Because watchpoints are set on virtual addresses, the watchpoint system is accessed in parallel with the DTLB. This also allows watchpoints to be set with user-level instructions while not affecting the CPU's critical path.

5.2.3 Efficient Watchpoint Hardware

This section describes a hardware mechanism that operates in parallel with the virtual memory system in order to deliver on the requirements described in the previous section. Each watchpoint is defined by two bits that indicate whether it is read- and/or write-watched. To allow a ***large number*** of these watchpoints, the full list of watched addresses is stored in main memory. Accessing memory for each check would be prohibitively slow, so virtual addresses are first sent from the address generation unit (AGU) or load/store queue (LSQ) to an on-chip watchpoint cache that is accessed in parallel to the data translation lookaside buffer (DTLB), as shown in Figure 5.2.

To deliver these watchpoints at ***byte granularity***, the cache compares each virtual address that the instruction touches and outputs a logical OR of their watched statuses. This check need not complete until the instruction attempts to commit, and so it may be pipelined.

If the watchpoint unit indicates that the address is watched (this is referred to as a watchpoint fault), a precise exception is raised upon attempting to commit the offending instruction. This could either be a user-level fault, which is treated as a mispredicted branch, or a traditional kernel-level fault. We assume the former in order provide a ***fast fault handler***. Using such handlers for kernel-controlled watchpoints, or for cross-process watchpoints, is beyond the scope of this work.

In order to yield ***fast watchpoint changes***, it is important that the on-chip caches only write back to main memory on dirty evictions. This, and the fact that the watchpoints are stored as virtual addresses, means that watchpoints can be changed with simple user-level instructions instead of system calls. The instruction can hold the exact virtual addresses it wishes to watch (without needing a virtual→physical translation from the kernel), and because the cache can hold dirty data, the change need not wait on main memory.

Mechanisms for dealing with virtual memory aliasing are out of the scope of this chapter, but it is worth noting that the tools we're trying to accelerate are run at user level. In their

traditional, non-accelerated forms, they also cannot deal with virtual aliasing, as they do not know about physical memory. In such a case, it may be reasonable for the hardware to also miss a watchpoint on a virtual alias.

In order to support ***per-thread*** (rather than per-core) watchpoints, any dirty state in the watchpoint cache will necessarily be part of process state. This can be switched lazily, only being saved if another process utilizing watchpoints is loaded, to increase performance. Having per-core meta-data was not an impediment to the accuracy of the data race detector in Chapter 4, but this ability helps maintain the ideal nature of our system. Additionally, each core will require a thread ID register so that individual threads can be targeted with watchpoint changes by the user-level instructions.

In order to support the many different watchpoint usage models, watchpoints are stored on-chip in three different forms. They are first stored in a range cache (RC), which holds the start and end addresses of each cached watchpoint and the status values that hold each range's 2-bit watchpoint. This system helps support ***setting and breaking ranges***.

Small ranges can negatively impact the reach, or the number of addresses covered by all entries, of the RC. In this case, the watchpoints within a contiguous region can be held in a single bitmap stored in main memory, rather than as multiple small ranges. The RC then stores the boundary addresses for the entire "bitmapped" region, and the status bits associated with that entry will point to the base of the bitmap. Accessing this bitmap would normally require a load from main memory, so we also include a Watchpoint Lookaside Buffer (WLB) that is searched in parallel with the RC.

Finally, because creating bitmaps can be slow (requiring kilobytes of data to be written to main memory), it can be useful to store small bitmaps directly on chip. By using the storage that normally holds a pointer to a bitmap, it is possible to also make an On-Chip Bitmap (OCBM) that stores the watchpoints for a region of memory that is larger than a single byte but smaller than a main memory bitmap.

Range Cache

The first mechanism for storing watchpoints within the core is a modified version of the range cache proposed by Tiwari *et al.* [216]. They made the observation that "many dataflow tracking applications exhibit very significant range locality, where long blocks of memory addresses all store the same tag value," and we have found this statement even more accurate when dealing with 2-bit watchpoints rather than many-bit tags. This cache, shown as part of Figure 5.3, stores the boundary addresses for numerous ranges within the virtual memory space of a thread and the 2-bit watchpoint status associated with each of these regions.

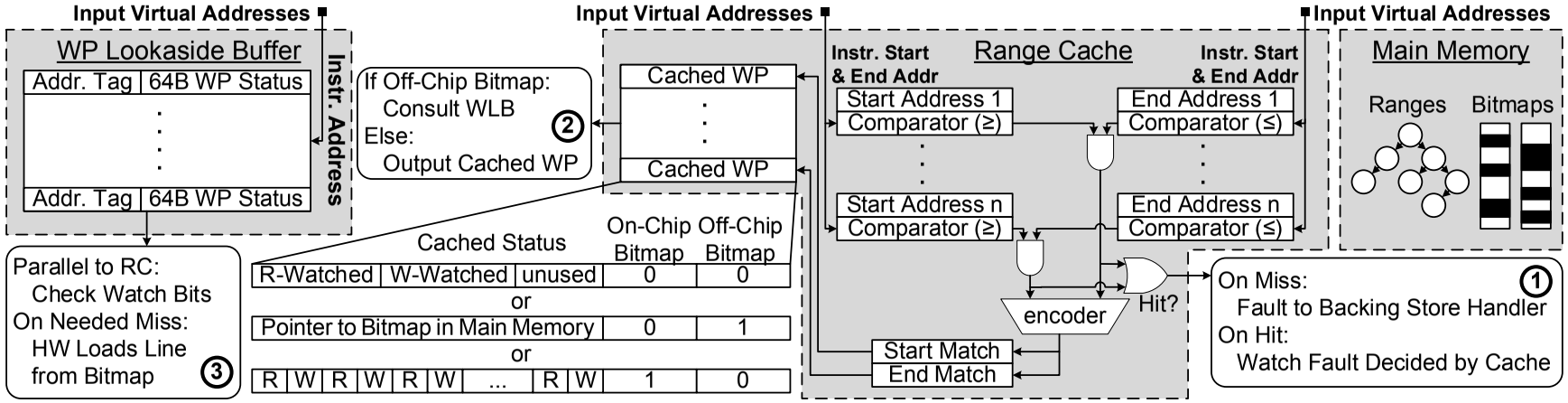


Figure 5.3 Unlimited Watchpoint Architecture. This system uses a combination of a range cache, center, with bitmaps and a lookaside buffer, left, to accelerate accesses to watchpoints that are stored into main memory, right. **(1)** On a RC access miss, a software handler loads new ranges from main memory. A hit causes the watchpoint system to check the associated status entry. **(2)** This range could be a uniform watchpoint, an off-chip bitmap, or an on-chip bitmap. In the case of an off-chip bitmap, the output of the lookaside buffer, which is accessed in parallel to the RC, is consulted. **(3)** If the WLB misses, the pointer from the RC status entry is used by the hardware to load a line in from the main memory bitmap.

Virtual addresses are sent to the RC in parallel with DTLB lookups; the boundary addresses of each access are compared with those of the cached ranges. When any address that a memory instruction is attempting to access overlaps with a range stored in the RC, the hardware checks that range's watchpoint bits. If an overlapping range is marked as watched for this type of access, the instruction will cause a watchpoint fault when it commits. This causes execution to jump to a software-based watchpoint fault handler.

If there are no watchpoints set on a region of memory, an overlapping entry must still exist somewhere. This entry will have both R- and W-watched bits set to '0', indicating that neither type of access will cause a fault. This means that if the range cache *misses* on any lookup, it should attempt to retrieve that range from main memory, since this *is not* an indication that the accessed locations are not watched.

The original range cache design by Tiwari *et al.* brought in 64 byte chunks from a two-level trie whenever it missed. This method was inefficient for the software tools studied in this work, as it required a large number of writes to save non-aligned ranges into the trie. Instead, the RC modeled in this chapter causes a fault to a software handler on a miss. This handler then loads an entire range from the main memory storage system. We implemented this backing store handler such that it kept a balanced tree of non-overlapping ranges in memory. This structure was modeled after the watchpoint data structure used in the page fault handler of the OpenSolaris kernel.

Programs set watchpoints on their own memory space using range-based instructions, which are described later in this section. These instructions can set or remove ranges by directly inserting the new watchpoint tag into the range cache, which uses a dirty-bit write-back policy to avoid taking a fault to the backing store handler on every watchpoint change. The RC employs a pseudo-LRU replacement policy, which keeps track of the most-likely candidate for eviction. If the range cache overflows and the LRU entry is dirty, the cache will cause a user-level fault that reroutes execution to the backing store handler, which will save that entry out to memory and overwrite it with the needed data.

Updating a watchpoint range is more complicated than setting or removing, as it may require loading in ranges from the backing store to find the value that is to be modified.

Bitmap and Watchpoint Lookaside Buffer

The RC is designed to quickly handle large ranges of watchpoints, and its write-back policy can significantly reduce the number of writes to main memory. However, its reach can be severely limited if it contains many short ranges. In the worst case, a 128-entry RC, which takes up an area roughly equal to 4KB of L1D, may only have a reach of 128 bytes. We

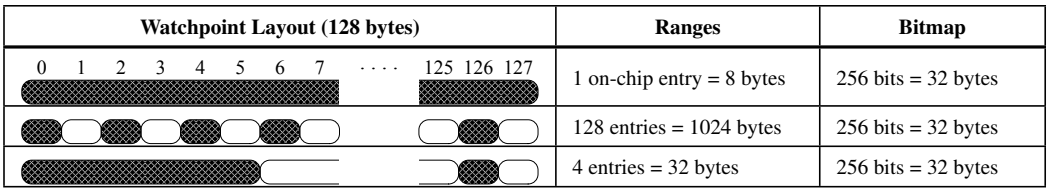


Figure 5.4 Different Watchpoint Storage Methods. This shows three examples of watched ranges. In the first, a range cache can hold the entire region in a single on-chip entry. The second, however, would take 1KB of on-chip storage if it were held in a range cache, while a bitmap method would only take 32 bytes. The final entry shows a break-even point between the two methods.

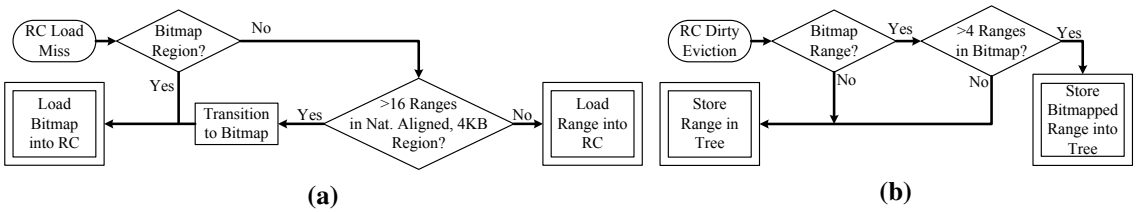


Figure 5.5 Software Algorithm for Bitmapped Ranges. Because hardware has a myopic view of the global status of watchpoints, we rely on the software fault handlers to choose when a region of watchpoints should toggle between a bitmap and a range.

utilize a second hardware structure to handle these cases.

Bitmaps are a compact way of storing small watchpoint regions, because they only take 2 bits per byte within the region, as demonstrated in Figure 5.4. Instead of storing the start and end address for each small range, we therefore choose to sometimes store into the range cache the first and last address of a large region whose numerous small watchpoints are contained in a bitmap in memory. This increases the required amount of status storage in the range cache, which originally only held the read-watched and write-watched bits. It instead requires 32 or 64 bits of storage to hold the pointer to the bitmap and one bit to denote whether an entry is a bitmap pointer or a range. The two watched bits can be mapped onto the pointer bits to save space.

We found it difficult to design a hardware-based algorithm to decide when to change a collection of ranges into a bitmap. Doing so in an intelligent manner requires knowing how many watchpoints are contained within a particular area of memory. This knowledge may best be gathered by the backing store software, as it can see the entire state of a process's watchpoints. We therefore leave it to the software backing store handler to decide when to toggle a region of memory between a bitmap and ranges. The algorithm modeled in this paper moves a naturally aligned 4KB region from a range to a bitmap if the number of internal ranges exceeds an upper threshold of 16. The dirty eviction handler changes a bitmap back to ranges if this number falls below 4. This is illustrated in Figure 5.5.

While this mechanism increases the reach of the range cache, it could adversely affect performance if every access to a bitmapped region required checking main memory. This is especially true because the location of the bitmap is held as a virtual address, meaning that each access to it would wait on the TLB. In order to accelerate this process, our system includes a Watchpoint Lookaside Buffer (WLB). This cache is accessed in parallel to the RC, and any watchpoint status it returns is consulted if the RC indicates that this location is stored in a bitmap. On a miss in the WLB, a 64-byte line of watchpoint bits is loaded by a hardware engine from the bitmap pointed to by the RC entry. Changes to the watchpoints in a bitmapped region cause a WLB eviction to avoid expensive hardware to translate the range-based instructions into multiple WLB modifications.

This WLB could be replaced with extended cache line bits, such as iWatcher uses to store its bitmapped watchpoints [235]. Sentry's power-saving method of only storing unwatched lines in the L1 (and thus only checking the WLB on cache misses) may also be useful [203]. We leave a more in-depth analysis of these tradeoffs for future work, and focus on a system with a separate watchpoint cache, much like MemTracker uses [221].

If a bitmapped range is evicted from the RC to main memory, the modeled software handler stores the entire range as a single entry in the balanced tree, along with the pointer. It then brings the entire bitmap range back into the range cache on the next miss, though it may also need to update the bitmap to handle changes that occurred in the range cache but that have not yet been evicted.

On-Chip Bitmap

There are also situations where a collection of small ranges within a large region prematurely displace useful data from the range cache. Though the backing store handler may later fix this by changing the region to a bitmapped one, we also devised a mechanism to better utilize the pointer bits in a range cache entry when it is not bitmapped. It is possible to store a small bitmap for a region $\leq$16 bytes (on a 32-bit machine) within these bits, allowing the RC to slightly increase its reach. This on-chip bitmap (OCBM) range is stored much like a normal range, with an arbitrary start and end address. However, unlike a uniformly tagged range, the watchpoint statuses of OCBM entries are stored as a bitmap in the range cache's status tag bits, which normally hold either a pointer to a main memory bitmap or the 2-bit tag of a range. The low-order bits of the address are used to index into this bitmap, allowing a single 32-bit range cache entry to hold up to 16 consecutive small watchpoints.

Unlike the bitmaps stored in main memory, the transition to OCBM requires little knowledge. Simply put, if a range becomes small enough to be held in a single OCBM entry,

Table 5.2 ISA Additions for Unlimited Watchpoints. Watchpoint modifications must be memory fences, and instructions working on remote cores cause a shootdown in the remote WLB. Instructions used by the backing store handler can set all bits in a RC entry, and are also used during task switching.

Watchpoint Modifications	
`set_local_wp` *start, end,* {*r, w, rw, 0*}	Adds a R/W/RW/not-watched range into this CPU's RC. Overwrites any overlapping ranges.
`add/rm_local_wp` *start, end,* {*r, w*}	Updates an entry in the RC of this CPU. If anything between *start* and *end* is not in the RC, it must be read from the backing store to properly update it.
`set_remote_wp` *start, end, tid,* {*r, w, rw, 0*}	Adds an entry into the RC of the CPU with TID register *tid*. May take many cycles waiting for other processors to respond. May also fault to software.
`add/rm_remote_wp` *start, end, tid,* {*r, w*}	Updates an entry in the RC of the CPU with TID register *tid*. May take many cycles waiting for other processors to respond. May also fault to software.
Backing Store Handler	
`read_rc_entry` *n*	Reads the n^{th} entry of the RC (LRU order). This allows the oldest entries to be sent to the backing store. A bulk-read instruction could be used for task-switching.
`store_rc_entry` *start, end, status_bits*	Allows writing an entire range (including bitmap and OCBM status bits) into a range cache entry. Used for reading in a watchpoint on a RC miss.
Watchpoint System Interface	
`enable/disable_wp`	Enable or disable the watchpoint system on this core. Kernel-level instruction.
`set_cpu_tid` *tid*	Set the TID register on this CPU at thread switch time.
`enable/disable_wp_thread` *tid*	Toggle watchpoint operation for any core with the same PID as this core's and TID equal to *tid*.
`set_handler_addr` *addr*	Set the address to jump to when a watchpoint-related fault occurs. If faults are user-level, then this instruction is user-level.
`get_cpu_tid`	Find the value in this processor's TID register.

it is converted to one by the hardware. Returning to a uniform range will occur when the hardware detects that all of the watchpoints within the OCBM are equal. If an OCBM is chosen to be written back to memory, our backing store handler will store it as individual ranges. If two OCBMs cover adjacent ranges, the hardware will merge them only if their total size would still result in an OCBM.

ISA Changes

Interfacing with this watchpoint system requires modifications to the ISA. For instance, we must add instructions that can set or remove ranges from the RC, as well as instructions that allow the backing store handler to directly talk to the RC hardware. Table 5.2 lists the instructions that must be added and describes what each does.

The most complicated instruction semantics relate to modifying watchpoints in multi-threaded programs. The WLB, for example, must be able to respond to shootdown requests

in order to remain synchronized across multiple threads. Most importantly, instructions must exist to add, remove, and update ranges of watchpoints for sibling processors. To do this quickly, our system must send these requests without going through the OS. One method of doing this involves broadcasting to all cores a process ID (e.g., the CR3 register in x86) and thread ID along with the request to perform a global update. Each core can then update its own cache entry if it matches the target process and thread ID. This instruction must be a memory fence, however, to maintain consistency between watchpoints and normal requests to watched memory locations. After the remote threads update their watchpoint cache, they must also send back acknowledgments. If any target thread ID does not return an acknowledgment within some timeout period, it may not be running, and its watchpoints may be saved off into memory. The source processor must then raise an interrupt and allow the OS to update the unscheduled process's watchpoints.

5.3 Watchpoint Applications

This section analyzes potential applications for this watchpoint system. It details how each requires some subset of the requirements listed in Section 5.2.1 and then develops watchpoint-based algorithms that can accelerate each analysis. We only detail the programs used as tests in our experiments, but Table 5.3 covers other tools that are also briefly discussed in Section 5.5. Two of our tools, dynamic dataflow analysis and data race detection, were discussed in detail in previous chapters. The third, deterministic execution, is used to show that unlimited watchpoints can accelerate not just the dynamic analyses we've already studied, but other advanced systems as well. We review all of these analyses in the following sections.

5.3.1 Dynamic Dataflow Analysis

Dynamic dataflow analyses associate shadow values with program data, propagate them alongside the execution of the program, and perform a variety of checks on them to find errors. This meta-data can represent myriad details about the associated memory location such as trustworthiness [38], symbolic limits [118], or identification tags [148]. Unfortunately, these systems suffer from high runtime overheads, as every memory access must first check its associated meta-data before calculating any propagation logic.

Ho *et al.* described a method for dynamically disabling a taint analysis tool when it is not operating on tainted variables, allowing the majority of memory operations to proceed

Table 5.3 Applications of Watchpoints. This is a sample of software systems that could utilize hardware-supported watchpoints to perform more efficiently and accurately. High-level algorithms that focus on how such systems would interact with the watchpoint hardware are given for each, as well as an overview of the watchpoint capabilities that would be useful or needed for each algorithm.

Software System	High Level Algorithm	Large Number	Byte Granular	Fast Handler	Fast Changes	Per Thread	Set Ranges	Break Ranges
Demand-Driven Dataflow Analysis	`Set shadowed values as RW watched.` `Enable analysis tool only on watchpoint faults.`	X	X	X	X			

Deterministic Execution	`Start with shared memory RW watched in all threads.`
	`On local access fault:`
	`  Check for write conflict between threads.  If so, serialize.`
	`  Unwatch (R or RW, depending on access) cache line locally.`
	`  Rewatch cache lines on other threads after serialization.`
		X				X	X	X

Demand-Driven Data Race Detection	`Start with shared memory RW watched in all threads.`
	`On local access fault:`
	`  Run software race detector on this access.`
	`  Unwatch (R or RW, depending on access) address locally.`
	`  Rewatch address on other threads.`
		X	X	X	X	X	X	X

Bounds Checking	`Set W-watchpoints on canaries, return addresses, and heap meta-data.`	X	X					

Speculative Program Optimization	`Mark data regions as W watched in speculative and normal threads.`
	`On faults:`
	`  Save list of modified regions (perhaps larger than pages).`
	`  Mark page available in this thread.`
	`Compare values in modified regions when verifying speculation.`
		X				X	X	X

Hybrid Transactional Memory	`At start of transaction, set local memory as RW watched.`
	`On local access fault:`
	`  Save original value for rollback.`
	`  Check for conflicts with other transactions.`
	`  Unwatch (R or RW, depending on access) address locally`
	`Unwatch memory for this thread on transaction commit.`
		X		X	X	X	X	X

Semi-space Garbage Collection	`During from-space/to-space switch:`
	`  Mark all memory in from-space as RW watched to executing threads.`
	`  Update dereferenced pointer to be consistent on faults.`
		X				X	X	

without any analysis overheads [93]. Pages that contain any tainted data are marked unavailable in the virtual memory system. Programs will execute unencumbered when operating on untainted data, but will cause a page fault if they access data on a tainted page. At this point, the program can be moved into the slow analysis tool.

Ho *et al.* discussed the problem of *false tainting*, where the relatively coarse granularity of pages causes unnecessary page faults when touching untainted data. Ideally, such a demand-driven taint analysis tool would utilize byte-accurate watchpoints. Additionally, although they partially mitigated the slowdowns caused by the lack of a fast fault handler by remaining inside their analysis tool for long periods of time, this can limit performance. Their tool conservatively remains enabled while performing no useful analysis.

A watchpoint-based algorithm for this type of system can be summarized as setting RW watchpoints on any data that is tainted and running until a watchpoint fault occurs. The fault indicates that a tainted value is either being written over or read from, and the propagation logic or instrumented code should be called from the watchpoint handler. The tool should also remain enabled while tainted data exists in registers, as those values cannot be covered by watchpoints.

5.3.2 Deterministic Concurrent Execution

Deterministic concurrent execution systems attempt to make more robust parallel programs by guaranteeing that the outputs of concurrent regions are the same each time a program is run with a particular input [21]. This can be accomplished by allowing parallel threads to run unhindered when they are not communicating, but only committing their memory speculatively. If one thread concurrently modifies the data being used by another, they must be serialized in some manner.

Grace, one example of this type of system, splits fork/join parallel programs into multiple processes and marks potentially shared heap regions as watched in the virtual memory system [22]. Any time the program reads or writes a new heap page, a watchpoint fault is taken, whereupon the page is put into that process's read or write set and marked as read-only or available, respectively. As the processes merge while joining, any conflicting write updates cause one of the threads to roll back and re-execute.

While Berger *et al.* demonstrated the effectiveness of this system for a collection of fork/join parallel programs, using virtual memory watchpoints in such a way limits the applicability of their system. First, it only works on programs that can easily be split into multiple processes, which can lead to performance and portability problems in some operating systems. Additionally, it can only look for conflicting accesses at the page granularity.

While many developers work to limit false cache line sharing between threads, it is much less likely that they care to limit false sharing at the page granularity. Such a mismatch can lead to many unnecessary rollbacks, again reducing performance.

Our watchpoint-based deterministic execution algorithm is similar to Grace, except that it works on a per-thread basis and sets watchpoints at a 64B cache line granularity. This could also be done at the byte granularity, but with higher overheads.

5.3.3 Data Race Detection

Unordered accesses to shared memory locations by multiple threads, or data races, can allow variables to be changed in undesired orders, potentially causing data corruption and crashes [168]. Dynamic data race detectors can help programmers build parallel programs by informing them whenever shared memory is accessed in a racy way [195].

Chapter 4 showed a data race detection mechanism that operated on similar principles to demand-driven taint analysis, though it required the careful usage of performance monitoring hardware to observe cache events. That system could run into performance problems due to false sharing and may miss some races due to cache size limitations, among other issues.

It is possible to build such a demand-driven analysis system using per-thread watchpoints. Similar to how Grace operates, all regions of shared memory are initially watched for each thread. As a thread executes, it will take a number of watchpoint faults in order to fill its read and write sets in a byte-accurate manner. In our implementation, reads that take a watchpoint fault remove the local read watchpoint and set a write watchpoint on all other threads (in order to catch WAR sharing), while writes completely remove the local watchpoint and set RW watchpoints on all other cores (to catch RAW and WAW sharing). Any time an instruction takes a watchpoint fault, the software race detector can assume that it was caused by inter-thread data sharing, and should send the access through its slow race detection algorithm. A similar watchpoint-based race detection method was recently demonstrated in DataCollider, though they are only able to concurrently watch four variables due to watchpoint register limitations [65].

This tool is a prime example of the need to break large ranges, as the program initially starts by watching large ranges and slowly splits them to form a local working set.

Table 5.4 Exposed Latency Values for Watchpoint Hardware Evaluation. These events are counted in our simulator, and each event is estimated to take the listed number of cycles, on average.

Event	Added Cycles	Symbol
Kernel fault	700	T_{kernel}
Syscall entry	400	$T_{syscall}$
Signal from kernel	3000	T_{signal}
User-level fault time (branch misprediction)	20	T_{user}

5.4 Experimental Evaluation

Any slowdowns caused by issues such as watchpoint cache misses should not outweigh the performance gains a watchpoint system offers software tools. To that end, this section presents experiments that evaluate the performance of our watchpoint system and a collection of other fine-grained memory protection systems when they are used to accelerate dynamic analysis tools.

5.4.1 Experimental Setup

We implemented a high-level simulation environment using Pin [127] running on x86-64 Linux hosts. The software analysis tools were implemented as pintools and were used to analyze 32-bit x86 benchmark applications. These pintools communicated with a simulator that modeled the watchpoint hardware. It also kept track of events that would cause slowdowns. The overheads of events that were fully exposed to the pipeline (e.g. faults to the kernel) were calculated by multiplying the event count by values derived from running lmbench [137] on a collection of x86 Linux systems. These values are listed in Table 5.4.

Events that are not fully exposed, such as the work done by software handlers, were logged and run through a trace-based timing tool on a 2.4GHz Intel Core 2 processor running Red Hat Enterprise Linux 6.1. These actions are listed in Table 5.5. The runtime of the timing tool is recorded and used to derive the cycle overhead of such events.

While this setup is likely to have inaccuracies (due to, for instance, caches and branch predictors being in different states than what they would be on a real system), it is still useful in giving rough estimates of the performance of numerous different systems across a multitude of tools and benchmarks. It's worth nothing that even "cycle-accurate" simulators have inaccuracies compared to real hardware [232]. The larger testing space available to our fast simulation can still lead to useful design decisions.

Table 5.5 Pipelined Events for Hardware Watchpoint Evaluation. These are pipelined events that cause overlapping slowdowns. They are logged and their overheads estimated using a trace-based timing simulator.

Event	Symbol
Load watchpoints using SW handler.	$T_{SWcheck}$
Store watchpoints using SW handler.	T_{SWset}
Load watchpoints using hardware	$T_{HWcheck}$
Change page table protection bits.	T_{VM}
Create MLPT entries in MMP.	T_{MLPT}
Write data into bitmaps.	T_{bitmap}

Hardware Designs Modeled

To compare our design to previous works that could also offer watchpoints, we built models for a number of other hardware memory protection mechanisms. We assume that every system except virtual memory has user-level faults so as not to bias our results away from other hardware designs.

Virtual Memory – This models the traditional way of offering large numbers of watchpoints [7]. Touching a page with watched data causes a page fault, whereupon the kernel looks through a list of byte-accurate watchpoints to decide if this was a true watchpoint fault. If so, a signal is sent to the user code. If not, the page is marked available, the next instruction single-stepped, and the page is then marked unavailable again after the subsequent return to the kernel. The overheads of this system can be estimated as: (# true faults $\times$ ($T_{kernel} + T_{signal}$)) + (# false faults $\times$ $T_{kernel} \times 2$) + (# watchpoint changes $\times$ $T_{syscall}$) + $T_{SWcheck}$ + T_{SWset} + T_{VM}.

MemTracker – This implements a design that has a lookaside buffer that is separate from the L1D cache (called the "Taint L1" in FlexiTaint [219] and "State L1" in MemTracker [221]). In these tests, the State L1 (SL1) was a 4KB, 4-way set associative cache with 64-byte lines. The original MemTracker did not have user-level faults, so their backing store was a bitmap in main memory. Our initial tests showed that the vast majority of time was spent writing data to this bitmap, so we changed the system to use a software-controlled backing store handler similar to our design. The overheads of this system can be estimated as: ((# of faults + # SL1 misses) $\times$ T_{user}) + $T_{SWcheck}$ + T_{SWset}.

Mondriaan Memory Protection – MMP utilizes a trie in main memory to store the watchpoints for individual bytes [229]. Upper levels of the table can be set to mark large, aligned regions as watched in one action. The protection lookaside buffer (PLB) is 256 entries and can hold these higher-level entries (using low order don't-care bits). We did

not utilize the mini-SST optimization because Witchel later described how such entries can yield significant slowdowns if permission changes are frequent [228]. The overheads for this system are: (# of faults + T_{user}) + $T_{HWcheck}$ + T_{MLPT}

Range Cache – This system is a 128-entry range cache, where each entry holds 2 bits of watchpoint data. Tiwari *et al.* estimated that a 128-entry range cache with 32-bit entries would take up nearly the same amount of space as 4KB of data cache [216]. The OH is: ((# of faults + # RC misses + # write-backs) $\times$ T_{user}) + (complex range updates $\times$ 64 cycles) + $T_{SWcheck}$ + T_{SWset}.

RC + Bitmap – Our technique adds bitmapped ranges, OCBMs, and a 2-way set-associative, 2KB WLB to the range cache design. The size of the RC is reduced to 64 entries because of the extra area needed for these features. This system can have further overhead-causing events besides those of the range cache: ($T_{HWcheck}$ on WLB miss) + (time to decide to switch to/from bitmap ranges) + T_{bitmap}.

Software Test Clients

The software analyses discussed in Section 5.3 are utilized as clients for the simulated hardware watchpoint system. The overheads caused by the tools themselves are common amongst all watchpoint designs and are not modeled. In other words, the reported performance differences are relative to the meta-data checks only, not the shadow propagation, serialization algorithms, or race detection logic.

Demand-Driven Taint Analysis – This tool performs taint analysis on target applications, marking data read from disk and the network as tainted. As a baseline, we compare against two state-of-the-art analysis systems. The first, MINEMU 0.3, is an extremely fast taint analysis tool that utilizes multiple non-portable techniques (such as using SSE registers to hold taint bits) in order to run as fast as possible [27]. Because this system is a full taint analysis tool, its measured overheads *do* include taint propagation and checks. The second baseline system is Umbra, a more general shadow memory system built on top of DynamoRio [234]. Its overheads come from calculating shadow value locations and accessing them on each memory operation. Similar to the tests done by Tiwari *et al.* [216], we test these systems using the SPEC CPU2000 integer benchmarks.

Deterministic Concurrent Execution – We test the deterministic execution tool in a similar manner to Grace by using the benchmark tests included in the Phoenix shared-memory MapReduce suite [186]. We also tested the SPEC OMP2001 benchmarks, another set of programs with the fork-join parallelism that Grace is designed to support.

Demand-Driven Data Race Detection – The default tool we compare against is a

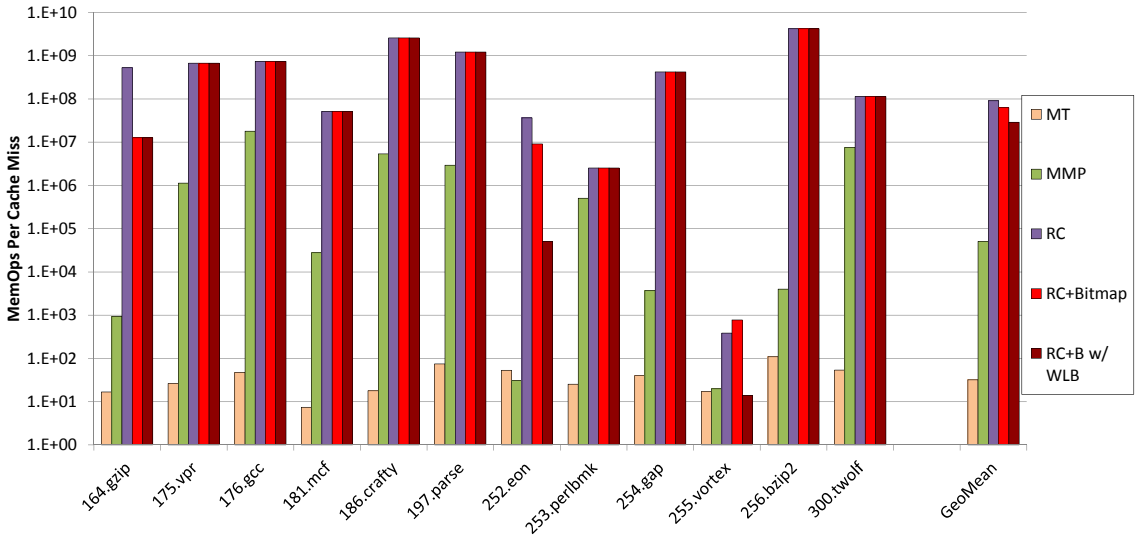


Figure 5.6 Memory Operations per Watchpoint Cache Miss for Taint Analysis. Most programs store a relatively small number of tainted regions, leading the range cache to have a very high hit rate (some of the benchmarks never miss). "RC+B w/ WLB" counts both range cache misses and the WLB misses, though the latter are filled much more quickly than the former.

commercial data race detector that performs this sharing analysis in software. We test this system in a similar manner to Chapter 4 by running the Phoenix [186] and PARSEC [24] suites.

5.4.2 Results for Taint Analysis

Figure 5.6 details the average number of memory operations between each watchpoint cache miss for all of the systems that use some cache mechanism (*i.e.*, not virtual memory). Because these programs have large regions of unwatched data, this tool shows the power of the range cache to cover nearly the entire working set of a program. Similarly, because the MMP PLB can hold aligned ranges, it has a larger reach than MemTracker's cache, which can only hold small sections of per-byte bitmaps. Nonetheless, Figure 5.7 shows that MMP has worse performance, on average, than MT. This is primarily because some benchmarks cause MMP to spend a great deal of time deciding whether to set or remove upper levels of the trie, in order to best utilize ranges in its PLB.

The taint analysis tool rarely causes our hybrid system to transition from using normal to bitmapped ranges. Because the size of the RC is reduced from 128 to 64 entries, its miss rate is slightly higher than the RC-only system's. The only counterexample is *255.vortex*, which taints a large number of small regions. However, when WLB misses are taken into

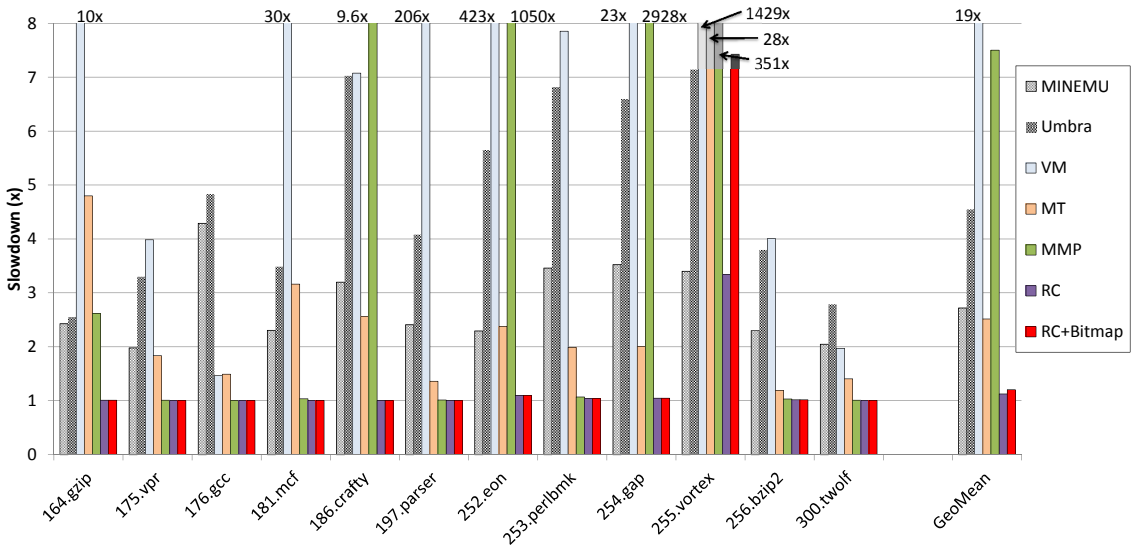
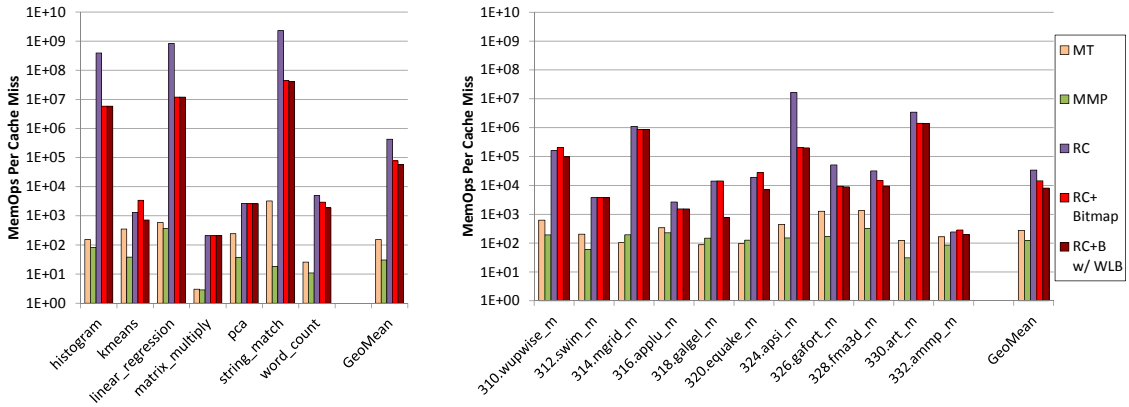


Figure 5.7 Performance of Demand-Driven Taint Analysis. The patterned are binary instrumentation systems that perform full taint analysis and shadow value lookup, respectively. The maximum slowdowns of the VM system are not shown, as they can go as high as 1400×. The RC + Bitmap solution performs poorly on 255.vortex because each watchpoint change to a bitmapped region causes an eviction from the WLB, and these are quite frequent. Nonetheless, this system easily outperforms manual shadow value lookups.

account (the line "RC+Bitmap" only accounts for range cache misses), the total hit rate drops precipitously. This is because the benchmark has a large number of watchpoint changes, and each change into a bitmapped region causes a WLB eviction.

Figure 5.7 compares the performance of the HW-assisted watchpoint systems against the two software-based shadow value analysis tools. The lower hit rate in the hybrid system means that it is about 7% slower than one with a larger RC. Nonetheless, the high hit rates of both systems mean that nearly every instruction that is not tainted suffers no slowdown, yielding large speedups over the software analysis tools. It is important to note, however, that MINEMU is also performing taint propagation, overheads which we do not analyze for our hardware systems. Nonetheless, using hardware-supported watchpoints can still result in large performance gains over an always-on tool like Umbra.

This performance benefit is dampened in *255.vortex* by its extremely low cache hit rates. All of the demand-driven analysis systems performed poorly for this benchmark, and would probably continue to do so even with high cache hit rates, since the software analysis tool would rarely be disabled – almost 10% of the memory operations in *255.vortex* operate on tainted values. Demand-driven tools are poor at accelerating applications such as this, and would need mechanisms such as sampling, as described in Chapter 2, to run faster.



(a) Phoenix Suite **(b)** SPEC OMP2001

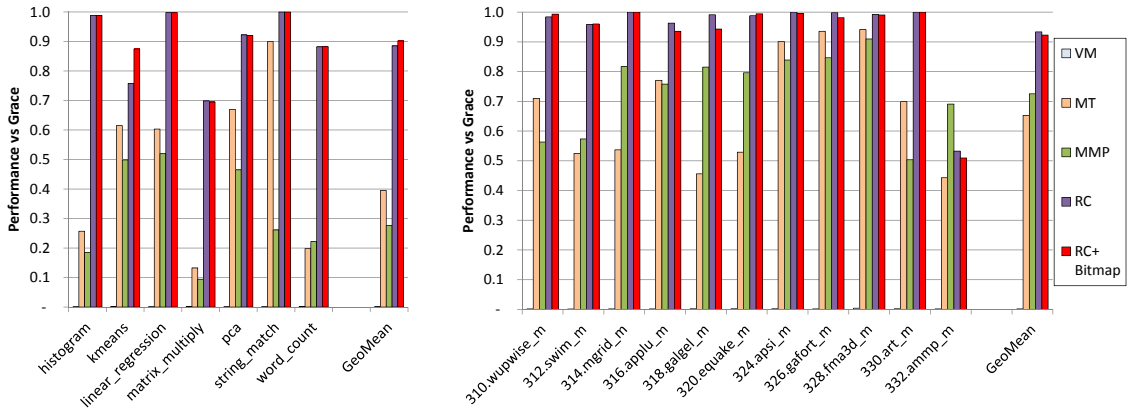
Figure 5.8 Memory Operations per Watchpoint Cache Miss for Deterministic Execution. The increased number of watchpoints causes the RC to have lower hit rates than it had for taint analysis. Because watchpoints are set and removed on cache line sized regions, the addition of a bitmap is only somewhat useful. It does not, however, increase the reach of the system as a whole since the size of the RC is reduced. Nonetheless, both systems take many fewer misses than other watchpoint hardware systems.

5.4.3 Results for Deterministic Execution

Figures 5.8 and 5.9 show the cache miss rates and performance, respectively, of the watchpoint-based deterministic execution system. The performance graph is normalized to a Grace-like system that removes watchpoints one page, rather than one cache line, at a time. The normalized performance of the more fine-grained mechanisms will be lower than 1.

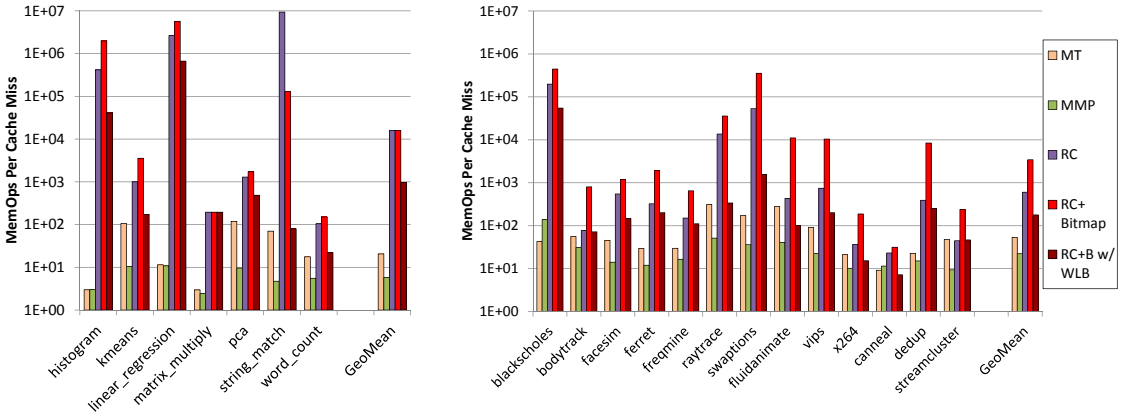
Because this system operates on more watchpoints than the taint analysis tool, the cache hit rates are lower. Very few of the watchpoints used in this tool can be held in higher levels of the trie, so MMP's PLB has a much worse hit rate. On average, MemTracker's hit rate is higher than MMP's due to its larger cache. Despite the increased number of watchpoints, the range cache maintains a high hit rate. This can partially be attributed to the tool using watchpoints of 64 bytes in length at minimum, which increases the reach of the range cache in highly fragmented cases.

As Figure 5.9 illustrates, attempting to use finer-grained watchpoints in the virtual memory system reduces performance significantly (it averages out to 670× slower than using 4KB watchpoints). All of the systems designed for fine-grained watchpoints handle this change much better. The systems that utilize ranges execute at about 90% of the speed of the VM-based system that uses 4KB watchpoints, with the bitmapped range system edging slightly higher. Because this tool primarily works on ranges of data, both range-based systems perform better than the other hardware systems that operate solely on bitmaps.



(a) Phoenix Suite **(b)** SPEC OMP2001

Figure 5.9 Deterministic Execution Performance: Cache Line vs. Page Granularity Watchpoints. The baseline system sets and removes watchpoints at the 4KB page granularity, while these systems operate on 64 byte lines. On average, both range-based systems operate at 90% of the baseline speed while maintaining more accurate checks.

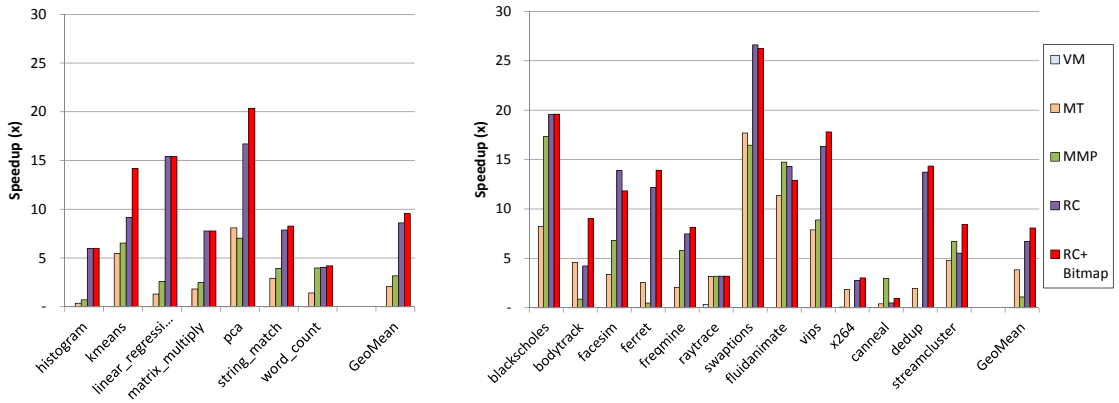


(a) Phoenix Suite **(b)** PARSEC

Figure 5.10 Memory Operations per Watchpoint Cache Miss for Data Race Detection. This system sets a large number of fine-grained watchpoints, making the bitmap addition especially useful. It increases the number of instructions between each RC miss by over 5× in PARSEC. The total miss rate with WLB included is higher, but those are filled much faster than RC misses.

5.4.4 Results for Data Race Detection

The cache miss rate for a demand-driven data race detector is shown in Figures 5.10. This tool deals with a much greater number of watchpoints, as it slowly unwatches data touched by individual threads at a byte granularity. The effect this has on the range cache can be easily observed in the Phoenix suite, as it falls from 100,000 memory operations between each range cache miss (in Grace) to 10,000 in this tool. The PARSEC benchmark *canneal* is particularly egregious, as none of the hardware caches go more than an average of 100



(a) Phoenix Suite **(b)** PARSEC

Figure 5.11 Performance of Demand-Driven Data Race Detection vs. Binary Instrumentation. Because the binary instrumentation tool is slow, most of the systems see significant speedups. The WLB miss handler is much faster than RC miss handler, so the bitmapped range system is noticeably faster than the RC system, even though it takes many WLB misses.

memory operations before missing. Because most of these benchmarks have a large number of relatively small ranges, however, this suite shows the benefit of the bitmapped ranges. In PARSEC, the geometric mean of the range cache hit rate is 5× higher when small ranges can be stored as bitmaps, even though the RC itself is half the size.

The performance of the hardware-assisted sharing detectors, shown in Figure 5.11, is almost always higher than when using software. In particular, the range based systems, even with low hit rates, can still outperform a software system that must check every memory access. The exception is *canneal*, where range-based hardware systems suffer high overheads from the backing store fault handler. In total, however, our hybrid system is able to demonstrate a 9× performance improvement, on average.

5.4.5 Experimental Takeaways

On the whole, the benefits of the range cache are pronounced. Its hit rate is significantly higher than caches that only store watchpoint bitmaps. Perhaps even more importantly, it acts as a write filter. Many of the slowdowns seen by MemTracker are due to writing to the backing store repeatedly. In a similar vein, the algorithm for filling out the trie in MMP can take up a great deal of time when there are many watchpoint changes, and breaking ranges apart can cause a large amount of memory traffic.

Nonetheless, we've found that the addition of a bitmap system to the range cache is beneficial when the watchpoints are small. There are numerous applications that create a

large number of small ranges. *dedup* within a race detector, for instance, is significantly helped by the increased reach of the RC when using bitmapped ranges. We found, however, that the WLB miss rate was quite high in many cases. This is partially because writes to bitmapped regions currently evict matching entries from the WLB. It is also the case that that bitmapped ranges are often enabled in sections of code that are difficult to cache. This warrants future research to perhaps find a better way to store these bitmapped ranges.

Though we did not illustrate the tests we did on the OCBM, we found that it was a net benefit, on average. However, its benefit was generally measured in single-digit percentages, rather than the larger gains we often saw with the full bitmap system.

Perhaps the most important area that could be improved in this system is the backing store handler. Range cache misses or overflows required hundreds to thousands of extra cycles to handle (depending on the size of backing store tree and the complexity of the insertion). With overheads this high, the range cache needs a particularly good hit rate to maintain high performance. One of the benefits we found with the bitmapped ranges was that the WLB miss handler was simpler and faster. It is probably possible to build more efficient backing store algorithms, such that they would switch between storage methods depending on the miss rate, eviction rate, and number of watchpoints in the system.

In summary, the hardware-assisted watchpoint systems allowed sizable performance improvements in demand-driven tools. For the deterministic execution system, the finer-grained watchpoints resulted in minor slowdowns, but allowed more accurate sharing analyses than can currently be performed at such a speed. This is the crux of the argument that hardware-assisted watchpoints are a generalized mechanism to improve dynamic software tools.

5.5 Related Work

This section explores works related to watchpoints and their uses. It first discusses hardware proposals that *could* be used to provide unlimited watchpoints, but which have limitations that keep them from being as general as we would like. It then lists other applications that could potentially utilize an unlimited watchpoint system.

5.5.1 Memory Protection Systems

There have been numerous proposals for hardware that provides watchpoint-like mechanisms. Capabilities, for instance, can be used to control software's access to particular

regions in memory [61]. Few capability-based systems were built, and even fewer still exist. Most recent publications focus instead on fine-grain memory protection.

One of the most widely cited works in this area is Mondriaan (also spelled Mondrian) Memory Protection [229]. MMP was designed with fine-grained inter-process protection mechanisms in mind, and is optimized for applications that do not perform frequent updates. The protection information is stored in main memory and is cached in a protection lookaside buffer (PLB). MMP utilized a ternary CAM for this cache, allowing naturally aligned ranges to be compactly stored. The first method Witchel proposed for storing protection regions in memory was a sorted segment table (SST), a list sorted by starting address. Though this allows O($ln\ n$) lookups and can efficiently store ranges, it is unsuitable for frequent updates. The second, a multi-level permission table (MLPT), is a trie that holds a bitmap of word-accurate permissions in its lowest level. It is beneficial to use upper levels of this table whenever possible in order to increase the PLB's reach. Checking ranges of permissions in order to "promote" a region can cause significant update slowdowns, however. In general, MMP is designed to work with tools that, while needing memory protection, do not perform frequent updates.

One common method of storing protection data is to put it alongside cache lines [163, 183, 235]. iWatcher, for example, stores per-word watchpoints alongside the cache lines that contain the watched data [235]. These bits are initially set by hardware and are temporarily stored into a victim table on cache evictions. The hardware falls back to virtual memory watchpoints if this table overflows. iWatcher can watch a small number of ranges, which must be pinned in physical memory. If this range hardware overflows, the system falls back to setting a large number of per-word watchpoints. In general, this system is inadequate for tools that require more than a small number of large ranges. Of note, Zhou *et al.* correctly state the usefulness of user-level faults in handling frequently-touched watchpoints. UFO, a similar system, used watchpoints to accelerate speculative software optimization [163]. Unfortunately, because it stores watchpoints in memory by updating the ECC bits of individual lines, it does not allow large ranges to be quickly set or removed. SafeMem uses a similar scheme [183].

MemTracker holds analysis meta-data (which is analogous to watchpoints) in a separate L1 cache, similar to our WLB. It also offers the option of using a hardware state machine to perform meta-data propagation [221]. This means that it can have even less overhead for some types of analyses than a system with a user-level fault handler. However, because its meta-data is stored as a packed array in main memory, it is time-consuming to set or remove watchpoints on large ranges. FlexiTaint, a follow-on work to MemTracker, has the same issues [219].

In order to reduce the number of accesses to the memory protection hardware, Sentry stores its protection data at the cache line granularity [203]. Every line in L1 is unwatched, and the on-chip protection structure is only checked on L1D misses. This can significantly improve the energy efficiency in programs that have few protection faults, since structures that hold protection data, such as the range cache, are often very power inefficient. By reducing the number of times these structures are checked, the dynamic power usage of the chip can be greatly reduced. Their paper also gives an excellent analysis of the type of hardware needed to perform watchpoints. However, their system has large overheads when performing frequent updates because it must go to a software handler on every change and evict cache lines accordingly.

5.5.2 Other Uses for Watchpoints

Watchpoints can also be used to accelerate software systems beyond the three examined in the experiments. Table 5.3 lists high-level algorithms for the tools that are discussed in this section.

Hill *et al.* previously made an argument for deconstructed hardware transactional memory (TM) systems [91]. They pushed for HTM additions that could be used for other things, like watchpoints; we instead argue that watchpoints can be used to make (among other things) faster TM systems. A watchpoint-based algorithm to accelerate software TM, as described by Baugh *et al.*, sets watchpoints on data touched by transactional code; any time another thread touches this data, a conflict will be caught [17]. If transactions are particularly long, forcibly setting watchpoints after every memory access may be slow. In this case, carving working sets out of large watched regions (as we demonstrated for deterministic execution and data race detection) may be faster, as the initial fault times are amortized across multiple memory operations.

Beyond correctness and debugging analyses, speculative parallelization systems allow some pieces of sequential code to run concurrently, and only later check if the result was indeed correct. In essence, watchpoints can be set on values created by the speculative code so that they are verified before being used elsewhere in the program. Fast Track performs these checks in software using virtual memory watchpoints [109].

Because they allow write faults to be taken on specified read-only regions of memory, watchpoints also enable security systems both as common as bounds checking [42] and as esoteric as kernel rootkit protection [226]. The latter requires little explanation, but it is useful to note that the authors lamented the "protection granularity gap," or the inability to set fine-grained watchpoints.

Watchpoints can even be used, in a fashion, for garbage collection. A semi-space collector will move all reachable objects from one memory space to another at collection time [39, 66]. To implement this efficiently, the program will continue to execute while data is moved, pausing if the program attempts to access data that has not yet been appropriately moved. Appel *et al.* did this by setting virtual memory watchpoints on each memory location that is to be moved [6]. This is similar to the mechanism that Lyu *et al.* use to perform online software updates [128].

5.6 Chapter Conclusion

This chapter presented a hardware design that allows software to utilize a virtually unlimited number of low-overhead watchpoints. By combining the ability of a range cache to store long watched regions with a bitmap's fast access and succinct encoding for small watchpoints, we were able to demonstrate an effective system for a collection of software tools. Unlike previous works, which were application specific, we demonstrated that watchpoints could accelerate tools ranging from taint analysis to deterministic execution. We also presented watchpoint-based algorithms for other advanced software tools. This chapter demonstrated that the software community can benefit from generic primitives provided by hardware, rather than needing application-specific mechanisms that are unlikely to be added to commercial microprocessors.

Though our results show that a design of this nature is promising, a number of avenues for future research remain open. Our range cache's software-controlled miss and eviction handler would be unusable without fast fault support. It may therefore be advantageous to look into hardware designs for operating on the watchpoints stored in main memory. There are also open questions as to the best algorithms for moving from ranges to bitmaps (and vice-versa). This chapter presented a simple algorithm that showed decent results, but it is probably not optimal either in its runtime or its decisions.

This chapter focused on the architectural design of an unlimited watchpoint system and the algorithmic design of software tools that would use it. The microarchitectural design of the watchpoint unit was left intentionally vague, because it depends heavily on the microarchitecture of the processor which will contain it. Nonetheless, there are interesting research questions that may arise when working out the low-level design details of an unlimited watchpoint mechanism. For example, the range cache system may be too power hungry for embedded parts (such as a smartphone processor), or a highly complex server processor may not be able to take user-level faults easily. The software that wants to utilize

hardware-supported watchpoints may need to be different from the algorithms presented in Table 5.3, as not all of the services presented in Section 5.2.2 may exist.

Finally, there are new types of software tools that could be built on top of watchpoint systems. Software developers and researchers have been valiantly extending the uses of the virtual memory system for decades. This ingenuity may very well yield novel uses for a byte-granularity watchpoint system in the future.

Chapter 6
Conclusion

> "Fortune disposes our affairs better than we ourselves could have desired; look yonder, friend Sancho Panza, where you may discover somewhat more than thirty monstrous giants, with whom I intend to fight, and take away all their lives: with whose spoils we will begin to enrich ourselves; for it is lawful war, and doing God good service to take away so wicked a generation from off the face of the earth."
>
> "What giants?" said Sancho Panza.
>
> "Those you see yonder," answered his master, "with those long arms; for some of them are wont to have them almost of the length of two leagues."
>
> "Consider, Sir," answered Sancho, "that those which appear yonder are not giants, but windmills; and what seem to be arms are the sails, which whirled about by the wind, make the millstone go."
>
> ---
>
> *The Ingenious Gentleman Don Quixote of La Mancha*
> Miguel de Cervantes

Software is a major component in the modern world's economy and in our daily lives. Despite this, many programs are error-prone and slow, owing partially to the complexity of modern computer systems. Performance scaling now relies heavily on concurrent programming, which is notoriously difficult. However, even traditional serial programs are often fraught with errors.

As discussed in Chapter 1, one of the most important mechanisms used to improve software is dynamic analysis. These tools observe software as it runs and attempt to reason about the program's correctness, security, or accuracy. There are numerous types of dynamic analysis tools in use within the software industry, and even more have been proposed in the literature.

To find as many errors as possible, developers should run their software under many different analyses, as each tool can only find a particular subset of errors. Similarly, dynamic analyses can only find their particular errors in sections of the program that are executed while under analysis. Ideally, then, users would run these tests while they run the software, since they can execute many more parts of the program than a limited number of developers.

Unfortunately, powerful analyses are also slow analyses. Tools such as data race detectors and atomicity violation locaters, which are invaluable in finding problems in the ever-more-important world of concurrent programming, slow the execution of the program under test by hundreds of times. This stifles the ability to run many of these tests over numerous inputs, limiting their efficacy.

6.1 Thesis Summary

This dissertation has discussed a collection of novel techniques for accelerating a wide range of dynamic software analyses. The end goal is to make these tools fast enough for end-users, as they represent a much larger collection of possible tester than the development teams that currently run all tests.

In a broad sense, the two mechanisms used to enable these speedups are 1) sampling and 2) hardware support. By combining these in unique ways, it is possible to speed up tests, distribute them to users, and achieve high quality software by eliminating bugs.

The solution presented in Chapter 2 allows demand-driven sampling of dataflow analyses (taint analysis in particular). It utilized virtual memory watchpoints to permit the underlying demand-driven analysis and stochastically removed watchpoints when attempting to reduce overheads. Using virtual memory to set these watchpoints had a number of limitations that curtailed such a system's usage for other analyses. The following chapters looked at methods of solving some of these problems.

Chapter 3 described Testudo, a hardware mechanism for allowing word-accurate metadata to be stored alongside the original memory values. These word-accurate watchpoints had advantages over page-granularity watchpoints, in that they curtailed false faults and allowed hardware to directly perform analyses only when needed (rather than for long periods of time, as was done in the virtual memory-based system). In order to reduce off-chip costs, Testudo took advantage of the concept of sampling and threw away any values that would normally be stored off-chip.

Chapter 4 solved another shortcoming of the virtual memory watchpoint system. Because virtual memory spaces are shared between the threads of a process, it is difficult to

use such a system to set different watchpoints for each thread. This means that it is difficult, if not impossible, to perform demand-driven multi-threaded analyses (or sampling) using virtual memory watchpoints. That chapter looked at a different type of hardware to enable a multi-threaded analysis only when it was needed. By using performance counters to take interrupts to the kernel on specific cache coherency events, it was possible to enable a data race detector only when inter-thread sharing was occurring.

Chapter 5 presented a generic mechanism for accelerating many different software tools: unlimited watchpoints. It described a hardware mechanism that allows software to set an unlimited number of byte granularity, per-thread watchpoints. By allowing hardware to cache this data in multiple ways, it was possible for many different types of software analyses to utilize these watchpoints. Many of these tools wait for the hardware watchpoint system to tell them when it is a good time to perform some analysis action, greatly accelerating their operation. Dynamic dataflow analysis and data race detection, the two mechanisms studied in the other chapters of this dissertation, can greatly benefit from this generic hardware, as can other advanced techniques, such as deterministic execution.

6.2 Future Research Directions

Some of the techniques explored in this dissertation are preliminary or prototypes, which point to new research questions. As with most research, new questions do not cease to appear simply because some answers have been found.

In reference to the software-based dataflow sampling discussed in Chapter 2, an underlying question raised by this technique is how to perform sampling without making very long dataflows impossible to observe. For example, a system that keeps the analysis enabled for short periods of time (to better favor interactivity) will find it very difficult to observe the entirety of a large dataflow. The system *must* be disabled at some point to reduce overheads, which can break long chains of dependent meta-data. Is there a fundamental trade-off between interactivity and analysis accuracy?

It is also the case that this style of analysis can only find errors on paths that are executed. Many security flaws, however, are located on paths that are rarely, if ever, run. In many cases, the first time the erroneous control flow path is taken is when the software is exploited. In this case, dynamic analyses offer no help in finding these errors before trouble arises. Perhaps there is some method of using the information gathered from a sampled analysis to find "rarely executed paths," where dangerous bugs may lie. These locations would warrant further investigation and may contain valuable bugs that a dynamic analysis could not find.

The demand-driven data race detector described in Chapter 4 was able to accelerate the analysis of a number of programs. However, like other demand-driven tools, it cannot *guarantee* better performance. There have been works on sampling data race detection, so it would be interesting to quantify the potential accuracy gains under a particular overhead threshold offered by a demand-driven sampling detector. Would a demand-driven and sampling data race detector that is 10× more efficient find 10× as many races at the same runtime overhead?

Finally, the hardware/software interface and high-level design for an unlimited watchpoint system was described in Chapter 5, but the microarchitectural details were left ambiguous. The way this system would be integrated into an existing core would depend heavily on the microarchitecture of the core, and these details could significantly affect the watchpoint hardware design. For instance, power-constrained devices may only be able to fit a small number of ranges into the range cache, as the CAM-like range checks use quite a bit of dynamic power. Other architectures may find it difficult to save out a large amount of data as part of process state. This could significantly change the system's design, requiring new software algorithms (much like how the data race detection algorithm in Chapter 4 is different from that shown in Chapter 5 due to the hardware differences).

Of course, there are also many new uses for the watchpoint hardware waiting to be discovered.

6.3 Conclusion

This dissertation has presented a collection of hardware and software techniques to accelerate dynamic software analyses. By making these tools fast, it becomes possible to distribute them to end-users, who can find significantly more errors due to their vast numbers. Because so many users exist, it is possible to reduce the overhead of many tools by sampling, or only finding some small percentage of errors during each execution. With added hardware support, whether a sample cache, performance counters, or watchpoint hardware, these tests can become even more efficient.

Bibliography

[1] Alain Abran, James W. Moore, Pierre Bourque, Robert Dupuis, and Leonard L. Tripp, editors. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. IEEE, 2004.

[2] Ilan Adler and Sheldon M. Ross. The Coupon Subset Collection Problem. *Journal of Applied Probability*, 38(3):737–746, 2001.

[3] Anant Agarwal, Liewei Bao, Ian Bratt, John Brown, Bruce Edwards, Matt Mattina, Chyi-Chang Miao, Carl Ramey, Mark Rosenbluth, and David Wentzlaff. TILE-Gx ManyCore Processor: HW Acceleration Interfaces and Mechanisms. In *the HOT CHIPS Symposium on High Performance Chips*, 2011.

[4] Aleph One. Smashing the Stack for Fun and Profit. *Phrack*, 49:14, 1996.

[5] AMD Incorporated. *AMD Opteron 6000 Series Platform Quick Reference Guide*, 2010.

[6] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time Concurrent Collection on Stock Multiprocessors. In *the Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 1988.

[7] Andrew W. Appel and Kai Li. Virtual Memory Primitives for User Programs. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1991.

[8] Apple, Inc. Clang Static Analyzer. `http://clang-analyzer.llvm.org/`.

[9] Matthew Arnold and Barbara G. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *the Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 2001.

[10] Matthew Arnold, Martin Vechev, and Eran Yahav. QVM: An Efficient Runtime for Detecting Defects in Deployed Systems. In *the Proc. of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2008.

[11] Mona Attariyan and Jason Flinn. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *the Proc. of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[12] David F. Bacon and Seth Copen Goldstein. Hardware-Assisted Replay of Multiprocessor Programs. In *the ACM/ONR Workshop on Parallel & Distributed Debugging*, 1991.

[13] Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. A Theory of Data Race Detection. In *the Workshop on Parallel and Distributed Systems: Testing and Debugging*, 2006.

[14] Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. Unraveling Data Race Detection in the Intel Thread Checker. In *the Workshop on Software Tools for MultiCore Systems*, 2006.

[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *the Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[16] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *the Proc. of the International Symposium on Computer Architecture (ISCA)*, 2000.

[17] Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In *the Proc. of the International Symposium on Computer Architecture (ISCA)*, 2008.

[18] Leonard E. Baum and Patrick Billingsley. Asymptotic Dstributions for the Coupon Collector's Problem. *The Annals of Mathematical Statistics*, 36:1835–1839, 1965.

[19] Bert Beander. VAX DEBUG: An Interactive, Symbolic, Multilingual Debugger. In *the Proc. of the Software Engineering Symposium on High-level Debugging*, 1983.

[20] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *the Proc. of the USENIX Annual Technical Conference*, 2005.

[21] Tom Bergan, Joseph Devietti, Nicholas Hunt, and Luis Ceze. The Deterministic Execution Hammer: How Well Does it Actually Pound Nails? In *the Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, 2011.

[22] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe Multithreaded Programming for C/C++. In *the Proc. of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[23] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM*, 53(2):66–75, February 2010.

[24] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *the Proc. of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[25] Hans-J. Boehm. How to Miscompile Programs with "Benign" Data Races. In *the USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2011.

[26] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. PACER: Proportional detection of data races. In *the Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[27] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The World's Fastest Taint Tracker. In *the Proc. of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.

[28] Derek Bruening and Qin Zhao. Practical Memory Checking with Dr. Memory. In *the Proc. of the Symposium on Code Generation and Optimization (CGO)*, 2011.

[29] Michael L. Bushnell and Vishwani D. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer, 2000.

[30] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *the Proc. of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[31] Calin Cascaval, Colin Blundell, Maged Michael, Hardol W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *ACM Queue*, 6(5):46–58, 2008.

[32] Miguel Castro, Manuel Costa, and Jean-Philippe Martin. Better Bug Reporting With Better Privacy. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[33] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *the Proc. of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2008.

[34] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Rock: A High-Performance Sparc CMT Processor. *IEEE Micro*, 29(2):6–16, 2009.

[35] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun's ROCK Processor. In *the Proc. of the International Symposium on Computer Architecture (ISCA)*, 2009.

[36] Howard Chen, Wei-Chung Hsu, Jiwei Lu, Pen-Chung Yew, and Dong-Yuan Chen. Dynamic Trace Selection Using Performance Monitoring Hardware Sampling. In *the Proc. of the Symposium on Code Generation and Optimization (CGO)*, 2003.

[37] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In *the Proc. of the International Symposium on Computer Architecture (ISCA)*, 2008.

[38] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *the Proc. of the International Conference on Dependable Systems and Networks (DSN)*, 2005.

[39] Chris J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[40] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. Holmes: Effective Statistical Debugging via Efficient Path Profiling. In *the Proc. of the International Conference on Software Engineering (ICSE)*, 2009.

[41] R. Chinchani, A. Iyer, B. Jayaraman, and S. Upadhyaya. Insecure Programming: How Culpable is a Language's Syntax? In *the Workshop on Information Assurance*, 2003.

[42] Tzi-cker Chiueh. Fast Bounds Checking Using Debug Registers. In *the Proc. of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC)*, 2008.

[43] 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 *the Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[44] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *the Proc. of the USENIX Annual Technical Conference*, 2008.

[45] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *the Proc. of the USENIX Security Symposium*, 2004.

[46] Cliff Click. Azul's Experiences with Hardware Transactional Memory. In *the Bay Area Workshop on Transactional Memory*, 2009.

[47] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the Real World. In *the Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[48] Jonathan Corbet, Greg Kroah-Hartman, and Amanda McPherson. Linux Kernel Development: How Fast it is Going, Who is Doing it, What They are Doing, and Who is Sponsoring It. Technical report, The Linux Foundation, December 2010.

[49] The Mitre Corporation. Common Vulnerabilities and Exposures. `http://cve.mitre.org/`.

[50] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *the Proc. of the International Symposium on Microarchitecture (MICRO)*, 2004.

[51] James R. Dabrowski and Ethan V. Munson. Is 100 Milliseconds Too Fast? In *the Proc. of the Conference on Human Factors in Computing Systems (CHI)*, 2001.

[52] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Deconstructing Hardware Architectures for Security. In *the Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, 2006.

[53] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *the Proc. of the International Symposium on Computer Architecture (ISCA)*, 2007.

[54] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Tainting is Not Pointless. *ACM SIGOPS Operating Systems Review*, 44(2):88–92, 2010.

[55] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid Transactional Memory. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[56] Abraham de Moivre. De Mensura Sortis, seu, de Probabilitate Eventuum in Ludis a Casu Fortuito Pendentibus. *Philosophical Transactions of the Royal Society*, 27:213–264, 1711.

[57] Hans de Vries. Chip-architect.com. `http://www.chip-architect.com/news/2007_02_19_Various_Images.html`, Feb. 2007.

[58] John Demme and Simha Sethumadhavan. Rapid Identification of Architectural Bottlenecks via Precise Event Counting. In *the Proc. of the International Symposium on Computer Architecture (ISCA)*, 2011.

[59] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. HardBound: Archiectural Support for Spatial Safety of the C Programming Language. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[60] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[61] David R. Ditzel. Architectural Support for Programming Languages in the X-Tree Processor. In *the Spring COMPCON*, 1980.

[62] Paul J. Drongowski. Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors. Technical report, AMD, Inc., 2007.

[63] Matthew B. Dwyer, John Hatcliff, Robby, Corina S. Păsăreanu, and William Visser. Formal Software Analysis: Emerging Trends in Software Model Checking. In *the Workshop on the Future of Software Engineering*, 2007.

[64] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Multithreaded Java Program Test Generation. *IBM Systems Journal*, 41:111–125, 2002.

[65] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective Data-Race Detection for the Kernel. In *the Proc. of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[66] Robert R. Fenichel and Jerome C. Yochelson. A LISP Garbage-Collector for Virtual-Memory Computer Systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[67] Cormac Flannagan. Hybrid Type Checking. In *the Proc. of the Symposium on Principles of Programming Languages (POPL)*, 2006.

[68] Cormac Flannagan and Stephen N Freund. Atomizer: a Dynamic Atomicity Checker for Multithreaded Programs. In *the Proc. of the Symposium on Principles of Programming Languages (POPL)*, 2004.

[69] Cormac Flannagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *the Proc. of the Conference on Programming Language Design and implementation (PLDI)*, 2009.

[70] Forrester Consulting. eCommerce Web Site Performance Today: An Updated Look at Consumer Reaction to a Poor Online Shopping Experience. Technical report, Forrester Research, Inc., 2009.

[71] Marc Fossi, Dean Turner, Eric Johnson, Trevor Mack, Téo Adams, Joseph Blackbird, Stephen Entwisle, Brent Graveland, David McKinney, Joanne Mulcahy, and Candid Wueest. Symantec Global Internet Security Threat Report. Technical Report Volume XV, Symantec Corporation, April 2010.

[72] Aurélien Francillon and Claude Castelluccia. Code Injection Attacks on Harvard-Architecture Devices. In *the Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2008.

[73] Jr. Frederick P. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975.

[74] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *the Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[75] Robert Golla. Niagara2: A Highly Threaded Server-on-a-Chip. In *the Fall Microprocessor Forum*, 2006.

[76] Google, Inc. Breakpad. `http://code.google.com/p/google-breakpad/`.

[77] Google, Inc. Socorro. `http://code.google.com/p/socorro/`.

[78] Joseph L. Greathouse, Chelsea LeBlanc, Todd Austin, and Valeria Bertacco. Highly Scalable Distributed Dataflow Analysis. In *the Proc. of the Symposium on Code Generation and Optimization (CGO)*, 2011.

[79] Joseph L. Greathouse, Zhiqiang Ma, Matthew I. Frank, Ramesh Peri, and Todd Austin. Demand-Driven Software Race Detection using Hardware Performance Counters. In *the Proc. of the International Symposium on Computer Architecture (ISCA)*, 2011.

[80] Joseph L. Greathouse, Ilya Wagner, David A. Ramos, Gautam Bhatnagar, Todd Austin, Valeria Bertacco, and Seth Pettie. Testudo: Heavyweight Security Analysis via Statistical Sampling. In *the Proc. of the International Symposium on Microarchitecture (MICRO)*, 2008.

[81] Joseph L. Greathouse, Hongyi Xin, Yixin Luo, and Todd Austin. A Case for Unlimited Watchpoints. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[82] Andy Greenberg. Meet The Hackers Who Sell Spies The Tools To Crack Your PC (And Get Paid Six-Figure Fees). *Forbes Magazine*, 189(6):1, April 9, 2012.

[83] Ashish Gupta, Bin Lin, and Peter A. Dinda. Measuring and Understanding User Comfort With Resource Borrowing. In *the Proc. of the International Symposium on High-Performance Distributed Computing (HPDC)*, 2004.

[84] Shantanu Gupta, Florin Sultan, Srihari Cadambi, Franjo Ivančić, and Martin Roetteler. Using Hardware Transactional Memory for Data Race Detection. In *the Proc. of the International Parallel & Distributed Processing Symposium (IPDPS)*, 2009.

[85] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, 2000.

[86] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A Single-Chip Multiprocessor. *IEEE Computer*, 30(9):79–85, 1997.

[87] Ruud A. Haring, Martin Ohmacht, Thomas W. Fox, Michael K Gschwind, David L. Satterfield, Krishnan Sugavanam, Paul W. Coteus, Philip Heidelberger, Matthias A. Blumrich, Robert W. Wisniewski, Alan Gara, George Liang-Tai Chiu, Peter A. Boyle, Normal H. Chist, and Changhoan Kim. The IBM Glue Gene/Q Compute Chip. *IEEE Micro*, 32(2):48–60, 2012.

[88] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition (Synthesis Lectures on Computer Architecture)*. Synthesis Lectures on Computer Architecture. Morgan and Claypool, 2010.

[89] Matthias Hauswirth and Trishul M. Chilimbi. Low-Overhead Memory Leak Detection Using Adaptive Statistical Profiling. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2005.

[90] Hexus.net Editors. Intel Pentium D Dual-Core Desktop CPU. `http://www.theregister.co.uk/2005/05/26/review_pentium_d/`, May 2005.

[91] Mark D. Hill, Derek Hower, Kevin E. Moore, Michael M. Swift, Haris Volos, and David A. Wood. A Case for Deconstructing Hardware Transactional Memory Systems. Technical report, University of Wisconsin-Madison, 2007.

[92] Martin Hirzel and Trishul Chilimbi. Bursty Tracing: A Framework for Low-Overhead Temporal Profiling. In *the Workshop on Feedback-Directed and Dynamic Optimization (FDDO)*, 2001.

[93] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical Taint-Based Protection using Demand Emulation. In *the Proc. of the European Conference on Computer Systems (EuroSys)*, 2006.

[94] Charles Antony Richard Hoare. Programming: Sorcery or Science? *IEEE Software*, 1(2):5–16, 1984.

[95] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. Informing Loads: Enabling Software to Observe and React to Memory Behavior. Technical Report CSL-TR-95-673, Stanford University, 1995.

[96] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. Informing Memory Operations: Memory Performance Feedback Mechanisms and Their Applications. *ACM Transactions on Computer Systems (TOCS)*, 16:170–205, 1998.

[97] Derek R. Hower and Mark D. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *the Proc. of the International Symposium on Computer Architecture (ISCA)*, 2008.

[98] IBM Corporation. *IBM PowerPC 970MP RISC Microprocessor User's Manual*, 2.3 edition.

[99] Carnegie Mellon Software Engineering Institute. CERT Statistics. `http://www.kb.cert.org/vuls/`.

[100] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manuals*.

[101] Intel Corporation. *Intel Architecture Instruction Set Extensions Programming Reference*, February 2012.

[102] John F. Jacobs. *The SAGE Air Defense System: A Personal History*. MITE Corporation, 1986.

[103] Nicholas Jalbert, Cristiano Pereira, Gilles Pokam, and Koushik Sen. RADBench: A Concurrency Bug Benchmark Suite. In *the USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2011.

[104] Ali Jannesari, Kaibin Bao, Victor Pankratius, and Walter F. Tichy. Helgrind+: An Efficient Dynamic Race Detector. In *the Proc. of the International Parallel & Distributed Processing Symposium (IPDPS)*, 2009.

[105] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. Instrumentation and Sampling Strategies for Cooperative Concurrency Bug Isolation. In *the Proc. of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2010.

[106] Mark Scott Johnson. Some Requirements for Architectural Support of Software Debugging. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1982.

[107] Stephen C. Johnson. Lint, a C Program Checker. Technical report, Bell Laboratories, 1978.

[108] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic Taint Analysis with Targetted Control-Flow Propagation. In *the Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2011.

[109] Kirk Kelsey, Tongxin Bai, Chen Ding, and Chengliang Zhang. Fast Track: A Software System for Speculative Program Optimization. In *the Proc. of the Symposium on Code Generation and Optimization (CGO)*, 2009.

[110] Ritchie S. King. The Top 10 Programming Languages. *IEEE Spectrum*, 48(10):84, October 2011.

[111] Donald E. Knuth and Luis Tabb Pardo. The Early Development of Programming Languages. In Nicholas Metropolis, Jack Howlett, and Gian-Carlo Rota, editors, *A History of Computing in the Twentieth Century*. Academic Press, 1980.

[112] Akinori Komatsubara. Psychological Upper and Lower Limits of System Response Time and User's Preference on Skill Level. In *the Proc. of the International Conference on Human-Computer Interaction*, 1997.

[113] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-Way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, March-April 2005.

[114] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous Chip Multiprocessors. *IEEE Computer*, 38(11):32–38, November 2005.

[115] Lap-chung Lam and Tzi-cker Chiueh. Checking Array Bound Violation Using Segmentation Hardware. In *the Proc. of the International Conference on Dependable Systems and Networks (DSN)*, 2005.

[116] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[117] Pierre-Simon Laplace. Mémoire sur les suites récurro-récurrentes et sur leurs usages dans la théorie des hasards. *Mémoires de l'Académie Royale des Sciences de Paris (Savants étrangers)*, 6:353–371, 1774.

[118] Eric Larson and Todd Austin. High Coverage Detection of Input-Related Security Faults. In *the Proc. of the USENIX Security Symposium*, 2003.

[119] Kevin P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal*, 29:1, 1996.

[120] Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[121] David Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 Processors. Technical report, Intel, 2009.

[122] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[123] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug Isolation via Remote Program Sampling. In *the Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[124] Benjamin Robert Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, 2004.

[125] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[126] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[127] 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 *the Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[128] Janghoon Lyu, Youngjin Kim, Yongsub Kim, and Inhwan Lee. A Procedure-Based Dynamic Software Update. In *the Proc. of the International Conference on Dependable Systems and Networks (DSN)*, 2001.

[129] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, 2002.

[130] Vincent Maraia. *The Build Master*. Addison-Wesley, 2005.

[131] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *the Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[132] Gale L. Martin and Kenneth G. Corl. System Response Time Effects on User Productivity. *Behaviour & Information Technology*, 5(1):3–13, 1986.

[133] Stephen McCamant and Michael D. Ernst. A Simulation-based Proof Technique for Dynamic Information Flow. In *the Workshop on Programming Languages and Analysis for Security (PLAS)*, 2007.

[134] Bruce McClintock. On the Measurement of Chance, or, on the Probability of Events in Games Depending Upon Fortuitous Chance (Translated from French). *International Statistical Review*, 52:237–262, 1984.

[135] Paul E. McKenney, editor. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, 2011.

[136] R. E. McLear, D. M. Scheibelhut, and E. Tammaru. Guidelines for Creating a Debuggable Processor. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1982.

[137] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *the Proc. of the USENIX Annual Technical Conference*, 1996.

[138] Microsoft Corp. Microsoft 2002 Annual Report and Form 10-K. `http://www.microsoft.com/msft/ar02/`, 2002.

[139] Barton P. Miller and Jong-Deok Choi. A Mechanism for Efficient Debugging of Parallel Programs. In *the Workshop on Parallel and Distributed Debugging (PADD)*, 1988.

[140] Charlie Miller. The Legitimate Vulnerability Market. In *the Workshop on Economics of Information Security (WEIS)*, 2007.

[141] Robert B. Miller. Response Time in Man-Computer Conversational Transactions. In *the AFIPS Fall Joint Computer Conference*, 1968.

[142] Sang Lyul Min and Jong-Deok Choi. An Efficient Cache-based Access Anomaly Detection Scheme. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1991.

[143] Nate Mook. AMD Delivers Dual-Core Athlon 64 X2. `http://www.betanews.com/article/AMD-Delivers-DualCore-Athlon-64-X2/1117550622`, May 2005.

[144] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based Transactional Memory. In *the Proc. of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2006.

[145] Richard John Moore and Suparna Bhattacharya. Computer System with Watchpoint Support, May 2006.

[146] Abdullah Muzahid, Norimasa Otsuki, and Josep Torrellas. AtomTracker: A Comprehensive Approach to Atomic Region Inference and Violation Detection. In *the Proc. of the International Symposium on Microarchitecture (MICRO)*, 2010.

[147] Abdullah Muzahid, Dario Suárez, Joseph Torrellas, and Shanxiang Qi. SigRace: Signature-Based Data Race Detection. In *the Proc. of the International Symposium on Computer Architecture (ISCA)*, 2009.

[148] Shashidhar Mysore, Bita Mazloom, Banit Agrawal, and Timothy Sherwood. Understanding and Visualizing Full Systems with Data Flow Tomography. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[149] Vijay Nagarajan and Rajiv Gupta. ECMon: Exposing Cache Events for Monitoring. In *the Proc. of the International Symposium on Computer Architecture (ISCA)*, 2009.

[150] Wolfgang E. Nagel. Using Multiple CPUs for Problem Solving: Experiences in Multitasking on The CRAY X-MP/48. In *the Proc. of the International Conference on Vector and Parallel Processors in Computational Science*, 1988.

[151] Satish Narayanasamy, Gilles Pokam, and Brad Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *the Proc. of the International Symposium on Computer Architecture (ISCA)*, 2005.

[152] National Institute of Standards and Technology. National Vulnerability Database. `http://nvd.nist.gov/`.

[153] National Vulnerability Database. Vulnerability Summary for CVE-2002-0068: Squid 2.4 STABLE3 and earlier. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2002-0068, 2002.

[154] National Vulnerability Database. Vulnerability Summary for CVE-2004-0803: Libtiff 3.6.1. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2004-0803, 2004.

[155] National Vulnerability Database. Vulnerability Summary for CVE-2005-3120: Lynx 2.8.6. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2005-3120, 2005.

[156] National Vulnerability Database. Vulnerability Summary for CVE-2006-6170: ProFTPD 1.3.0a and earlier. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2006-6170, 2006.

[157] National Vulnerability Database. Vulnerability Summary for CVE-2007-0104: xpdf 3.0.1. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-0104, January 2007.

[158] National Vulnerability Database. Vulnerability Summary for CVE-2007-0774: Apache Tomcat JK Web Server Connector 1.2.19/1.2.20. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-0774, 2007.

[159] National Vulnerability Database. Vulnerability Summary for CVE-2007-2807: Eggdrop 1.6.18. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-2807, 2007.

[160] National Vulnerability Database. Vulnerability Summary for CVE-2010-3864: OpenSSL 1.0.0. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-3864, 2010.

[161] Umesh Gajanan Nawathe, Mahmudul Hassan, Lynn Warriner, King Yen, Bharat Upputuri, David Greenhill, Ashok Kumar, and Heechoul Park. An 8-core, 64-thread, 64-bit, power efficient SPARC SoC: (Niagara 2). In *the Proc. of the International Solid State Circuits Conference (ISSCC)*, 2007.

[162] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, 2005.

[163] Naveen Neelakantam and Craig Zilles. UFO: A General-Purpose User-Mode Memory Protection Technique for Application Use. Technical report, University of Illinois at Urbana-Champaign, 2007.

[164] Netcraft Ltd. April 2008 Web Server Survey. http://news.netcraft.com/archives/2008/04/14/april_2008_web_server_survey.html, April 2008.

[165] Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-Checking Entire Programs without Recompiling. In *the Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE)*, 2004.

[166] Nicholas Nethercote and Julian Seward. How to Shadow Every Byte of Memory Used by a Program. In *the Proc. of the International Conference on Virtual Execution Environments (VEE)*, 2007.

[167] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *the Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[168] Robert H. B. Netzer and Barton P. Miller. What Are Race Conditions? Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[169] Robert H.B. Netzer and Barton P. Miller. On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions. In *the Proc. of the International Conference on Parallel Processing*, 1990.

[170] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *the Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2005.

[171] Next Generation Magazine Editors. Sega Saturn. *Next Generation Magazine*, 1(2):43, February 1995.

[172] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing Security Checks on Commodity Hardware. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[173] Robert O'Callahan and Jong-Deok Choi. Hybrid Dynamic Data Race Detection. In *the Proc. of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[174] Marek Olszewski, Qin Zhao, David Koh, Jason ansel, and Saman Amarasinghe. Aikido: Accelerating Shared Data Dynamic Analyses. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[175] Kunle Olukotun, Lance Hammond, and James Laudon. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. Morgan and Claypool, 2007.

[176] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The Case for a Single-Chip Multiprocessor. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[177] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[178] Andrea Pellegrini, Joseph L. Greathouse, and Valeria Bertacco. Viper: Virtual Pipelines for Enhanced Reliability. In *The Proc. of the International Symposium on Computer Architecture (ISCA)*, 2012.

[179] Brian Randell Peter Naur, editor. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee*. NATO, 1969.

[180] Jonathan Pincus and Brandon Baker. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. *IEEE Security and Privacy*, 2(4):20–27, 2004.

[181] Milos Prvulovic. CORD: Cost-effective (and nearly overhead-free) Order-Recording and Data race detection. In *the Proc. of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2006.

[182] Yao Qi, Raja Das, Zhi Da Luo, and Martin Trotter. MulticoreSDK: A Practical and Efficient Data Race Detector for Real-World Applications. In *the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 2009.

[183] Feng Qin, Shan Lu, and Yuanyuan Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Produciton Runs. In *the Proc. of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[184] Feng Qin, Cheng Wang, Zhenmin Li, H.-S. Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *the Proc. of the International Symposium on Microarchitecture (MICRO)*, 2006.

[185] Shlomo Raikin, Shay Gueron, and Gad Sheaffer. Protecting Private Data from Cache Attacks, June 2008.

[186] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *the Proc. of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2007.

[187] Jeffrey Rott. Intel Advanced Encryption Standard Instructions (AES-NI). Technical report, Intel, 2010.

[188] Avi Rushinek and Sara F. Rushinek. What Makes Users Happy? *Communications of the ACM*, 29(7):594–598, 1986.

[189] Olatunji Ruwase, Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Decoupled Lifeguards: Enabling Path Optimizations for Dynamic Correctness Checking Tools. In *the Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[190] Paul Sack, Brian E. Bliss, Zhiqiang Ma, Paul Petersen, and Josep Torrellas. Accurate and Efficient Filtering for the Intel Thread Checker Race Detector. In *the Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, 2006.

[191] 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.

[192] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. Programming the Cell Processor. *Dr. Dobb's Journal*, 32(4):26–31, 2007.

[193] Bruce Schneier. Stuxnet. `http://www.schneier.com/blog/archives/2010/10/stuxnet.html`, October 2010.

[194] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994.

[195] Edith Schonberg. On-The-Fly Detection of Access Anomalies. In *the Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 1989.

[196] Eric Schrock. Watchpoints 101. `http://blogs.oracle.com/eschrock/entry/watchpoints_101`, July 2004.

[197] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *the Proc. of the IEEE Symposium on Security and Privacy*, 2010.

[198] Robert C. Seacord. *The CERT C Secure Coding Standard*. Addison-Wesley Professional, 2009.

[199] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer – Data Race Detection in Practice. In *the Workshop on Binary Instrumentation and Applications (WBIA)*, 2009.

[200] Julian Seward and Nicholas Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-precision. In *the Proc. of the USENIX Annual Technical Conference*, 2005.

[201] Jinuk Luke Shin, Kenway Tam, Dawei Huang, Bruce Petrick, Ha Pham, Changku Hwang, Hongping Li, Alan Smith, Timothy Johnson, Francis Schumacher, David Greenhill, Ana Sonia Leon, and Allan Strong. A 40nm 16-Core 128-Thread CMT SPARC SoC Processor. In *International Solid State Circuits Conference (ISSCC)*, 2010.

[202] Ben Shneiderman and Catherine Plaisant. *Designing the User Interface*. Addison-Wesley, 2010.

[203] Arrvindh Shriraman and Sandhya Dwarkadas. Sentry: Light-Weight Auxiliary Memory Access Control. In *the Proc. of the International Symposium on Computer Architecture (ISCA)*, 2010.

[204] Alex Shye, Yan Pan, Ben Scholbrock, J. Scott Miller, Gokhan Memik, Peter A. Dinda, and Robert P. Dick. Power to the People: Leveraging Human Physiological Traits to Control Microprocessor Frequency. In *the Proc. of the International Symposium on Microarchitecture (MICRO)*, 2008.

[205] Karan Singh, Major Bhadauria, and Sally A. McKee. Real Time Power Estimation and Thread Scheduling via Performance Counters. In *the Workshop on Design, Architecture, and Simulation of Chip Multi-Processors*, 2008.

[206] Asia Slowinska and Herbert Bos. Pointless Tainting? Evaluating the Practicality of Pointer Tainting. In *the Proc. of the European Conference on Computer Systems (EuroSys)*, 2009.

[207] Asia Slowinska and Herbert Bos. Pointer Tainting Still Pointless (but we all see the point of tainting). *ACM SIGOPS Operating Systems Review*, 44(3):88–92, 2010.

[208] Brinkley Sprunt. The Basics of Performance-Monitoring Hardware. *IEEE Micro*, 22(4):64–71, 2002.

[209] Wolfgang Stadje. The Collector's Problem with Group Drawings. *Advances in Applied Probability*, 22(4):866–882, 1990.

[210] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[211] Zeev Suraski. Re: [PHP-DEV] Run-time taint support proposal. PHP-DEV Mailing List, December 19, 2006.

[212] Gregory Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing. Planning Report 02-3. Technical report, National Institute of Standards and Technologies, November 2002.

[213] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–35, 2002.

[214] Christian Terboven. Comparing Intel Thread Checker and Sun Thread Analyzer. *Parallel Computing: Architectures, Algorithms and Applications*, 38:669–676, 2007.

[215] Shyamkumar Thoziyoor, Naveen Muralimanohar, and Norman Jouppi. CACTI 5.0. Technical report, Hewlett-Packard Laboratories, October 2007.

[216] Mohit Tiwari, Banit Agrawal, Shashidhar Mysore, Jonathan Valamehr, and Timothy Sherwood. A Small Cache of Large Ranges: Hardware Methods for Efficiently Searching, Storing, and Updating Big Dataflow Tags. In *the Proc. of the International Symposium on Microarchitecture (MICRO)*, 2008.

[217] Niraj Tolia, David G. Andersen, and M. Satyanarayanan. Quantifying Interactive User Experience on Thin Clients. *IEEE Computer*, 39(3):46–52, 2006.

[218] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *the Proc. of the International Symposium on Microarchitecture (MICRO)*, 2004.

[219] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. FlexiTaint: A Programmable Accelerator for Dynamic Taint Propagation. In *the Proc. of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2008.

[220] Guru Venkataramani, Ionnis Doudalis, Yan Solihin, and Milos Prvulovic. MemTracker: An Accelerator for Memory Debugging and Monitoring. *ACM Transactions on Architecture and Code Optimization*, 6(2):1–33, 2009.

[221] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In *the Proc. of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2007.

[222] Bruce Verduyn. 2005 FBI Computer Crime Survey. Technical report, Federal Bureau of Investigation, 2005.

[223] Robert Wahbe. Efficient Data Breakpoints. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1992.

[224] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, 1997.

[225] Brian Waldecker. AMD Quad Core Processor Overview. `http://www.nccs.gov/wp-content/training/scaling_workshop_pdfs/AMD_ORNL_073007.pdf`.

[226] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering Kernel Rootkits with Lightweight Hook Protection. In *the Proc. of the Conference on Computer and Communications Security (CCS)*, 2009.

[227] Emmett Witchel. *Mondriaan Memory Protection*. PhD thesis, Massachusetts Institute of Technology, January 2004.

[228] Emmett Witchel. Considerations for Mondriaan-like Systems. In *the Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, 2009.

[229] Emmett Witchel, Josh Cates, and Krste Asanovic. Mondrian Memory Protection. In *the Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[230] Min Xu, Rastislav Bodik, and Mark D. Hill. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *the Proc. of the International Symposium on Computer Architecture (ISCA)*, 2003.

[231] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *the Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2007.

[232] Matt T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *the Proc. of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2007.

[233] Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *the Proc. of the ACM/IEEE Conference on Supercomputing*, 1996.

[234] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Umbra: Efficient and Scalable Memory Shadowing. In *the Proc. of the Symposium on Code Generation and Optimization (CGO)*, 2010.

[235] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *the Proc. of the International Symposium on Computer Architecture (ISCA)*, 2004.

[236] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *the Proc. of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2007.

[237] Craig Zilles and Gurindar Sohi. Master/Slave Speculative Parallelization. In *the Proc. of the International Symposium on Microarchitecture (MICRO)*, 2002.