UNIVERSITY OF CALIFORNIA, SAN DIEGO

Maintaining Safe Memory for Security, Debugging, and Multi-threading

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in
Computer Science (Computer Engineering)

by

Weihaw Chuang

Committee in charge:

Brad Calder, Chair
Jeanne Ferrante
Ranjit Jhala
Andrew Kahng
Chandra Krintz

2006

The dissertation of Weihaw Chuang is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

2006

For my parents, brother, and relatives who made this possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to acknowledge Prof. Brad Calder for his support as my advisor for these six years. I would also like extend my deep appreciation to my co-authors and acknowledge their hard work: Satish Narayanasamy, Dr. Osvaldo Colavin, Prof. Jeanne Ferrante, Prof. Ranjit Jhala, Dr. Gilles Pokam, Jack Sampson, Michael Van Biesbrouck, and Ganesh Venkatesh.

Section II.F contains materials to appear in "Bounds Checking with Taint-Based Analysis", in *2007 International Conference on High Performance Embedded Architectures and Compilers (HiPEAC 2007)*, W. Chuang, S. Narayanasamy, R. Jhala and B Calder. The dissertation author was the primary investigator and author of this paper.

Sections III.D, and III.E contain material to appear in "Unbounded Page-Based Transactional Memory", in *Proceedings of 12th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS XII)*, W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. Van Biesbrouck, G. Pokam, O. Colavin and B. Calder. The dissertation author was the primary investigator and author of this paper.

VITA

1995	Bachelors of Science, Massachusetts Institute of Technology
1995–2000	Intel Corporation
2000–2006	Research Assistant, University of California, San Diego
2006	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

W. Chuang, S. Narayanasamy, R. Jhala and B Calder, "Bounds Checking with Taint-Based Analysis", to appear in *2007 International Conference on High Performance Embedded Architectures and Compilers (HiPEAC 2007)*, Ghent, Belgium, January 2007.

W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M.V. Biesbrouck, G. Pokam, O. Colavin and B. Calder. "Unbounded Page-Based Transactional Memory", to appear in *Proceedings of 12th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS XII)*, October 2006.

W.K. Chen, S. Bhansali, T.M. Chilimbi, X.G. Gao and W. Chuang, "Profile-guided proactive garbage collection for locality optimization", In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI06)*, Ottawa, Ontario, Canada, June 2006.

W. Chuang, B. Calder, and J. Ferrante, "Phi Predication for Light Weight If-Conversion", In *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, March 2003.

W. Chuang and B. Calder, "Predicate Predicate Prediction for Efficient Out-of-order Execution", In *Proceedings of the 16th Annual ACM International Conference on Supercomputing (ICS03)*, June 2003.

L. Carter, W. Chuang and B. Calder, "An EPIC Processor with Pending Functional Units", In *High Performance Computing, 4th International Symposium (ISHPC 2002)*, May 2002.

ABSTRACT OF THE DISSERTATION

Maintaining Safe Memory for Security, Debugging, and Multi-threading

by

Weihaw Chuang

Doctor of Philosophy in
Computer Science (Computer Engineering)

University of California, San Diego, 2006

Brad Calder, Chair

As transistor budgets grow enabling chip multi-core processors, adding hardware support to ensure the correctness and security of programs will be just as important, for the average user, as adding another core. The goal of our research is to develop hardware support for software checks and for multi-threaded synchronization that protects memory from corruption. This is the source of a significant number of bugs and security issues. We want efficient low-overhead run-time performance of these checks and synchronization so that it can be left on all of the time, even in production code releases.

Bounds checking protects pointer and array accesses. It is the process of keeping track of the address boundaries for objects, and checking that the loads and stores do not stray outside bounds. It can serve two purposes: it can assist debugging by precisely capturing invalid memory accesses, and it can guarantee protection against buffer overflow attacks. Unfortunately, the high performance overhead of runtime bounds checking has prevented its inclusion in most released software. In this thesis, we analyze the sources of bounds checking overhead.

We then consider hardware/software enhancements: the effect merging the check into a single instruction, software optimizations based on potential for overflow to eliminate checks, and changes to meta-data layout to limit copying overhead.

Transactional Memories enable programmers to greatly simplify multi-threaded code by eliminating lock synchronization and its attending deadlock and livelock problems. Unlike locks they also enable speculative concurrent execution through the critical section. Specialized transactional memory can also aid concurrent programming models by providing determinism only when needed at run-time. A key limitation of past transactional memory proposals are that they have a finite memory capacity. Virtualized transactional memory (unbounded in space and time) are desirable, as they can increase the scope of transactions' use, and thereby further simplify a programmer's job. In this thesis, we introduce *Page-based Transactional Memory* to support unbounded transactions. We combine transaction bookkeeping with the virtual memory system to support fast transaction conflict detection, commit, abort, and to maintain transactions' speculative data.

I

Introduction

I.A The Importance of Software Bugs

Software safety is a paramount concern for developers and end users, because billions of dollars are lost, and lives are at risk when software fails. A NIST study estimated that $59.5 billion (2002) was lost due to software failure in the United States alone (0.6% Gross Domestic Product) [75] . One of the most famous case of software failure was caused by a data race condition in the Therac-25, a radiation medical device, resulting in five deaths [51]. Consequently a great deal of software development effort is spent avoiding software failure. Beizer [8] estimates that between 30% to 90% of software development cost is devoted to testing.

Broken applications do not fully describe the cost of these software failures. Malicious hackers exploit software bugs to steal personal, and financial information from computers, or to perform other nefarious activities. Computer viruses and Internet worms use software bugs to inject malicious code on a host computer to commit Internet crimes like launching denial-of-service attacks, installing back doors [56, 77] or keystroke loggers, or scanning for information like credit cards. Worms and viruses characteristically use that malicious code

to repropagate themselves, frequently overwhelming the ability of host and/or networks to handle normal workloads. Much like the organic viral epidemics, computer worms and viruses propagate at a rapid rate across the Internet reaching all corners of the globe. Just one of these worm attacks, "Code-Red" on 19 July 2001, infected 359,000 computers in fourteen hours that hosted the Microsoft Corporation "Internet Information Service" (IIS) server. The entire "Code-Red" epidemic has been estimated to cost the US $2.6 billion dollars [89]. Another IIS server worm called the "Witty Worm" in 2004 has the notoriety of being the first worm to intentionally delete data on the host machine [78]. Desktop machines are vulnerable too. A 2004 study found that an unpatched Windows XP SP1 connected to the Internet would be exploited by worms in less than four minutes [6].

Computer software has become more meaningful to the general public because they now touch everyday life by providing instant information from around the world and automating many common tasks like banking, maintaining contact lists and doing taxes. When computers fail, many more people are now affected. In the 1980's and 1990's, there was a large new demographic of non-technical computer users because of the reduction in cost of ownership of personal computers from transistor scaling which is commonly known as Moore's Law. This user base expects appliance like reliability from their computer, yet is unable or unwilling to manually patch their computer software when a bug is found [56], creating new demands on the computer vendors. Inter-networking of computers was also enabled by transistor scaling, allowing the rapid dissemination of knowledge, and unfortunately enabling the spread of worms and viruses as well. Microsoft faced both greater user expectations and new Internet based attacks, with the "Code-Red" worm exploits on their IIS server, and many other worm exploits on different software products. Bill Gates in a 2002 memo [26]

stated that Microsoft software needed to attain the reliability level of utility services such as water or electric, because their customers expected this reliability, especially because of earlier security breaches from worm attacks that shook their customers' trust. That memo launched Microsoft's Trustworthy Computing campaign, instituting more rigorous code development processes, automatic software updates, and other initiatives to root out exploits. This cost the company $100 million dollars for just 2002-2003 [55]. The Internet played a critical role with the "Code-Red" worm attack: Knowledge enabling the IIS exploit was posted on the World-Wide-Web in 18 June 2001 [56, 77], and shortly by 12 July the first worm appeared. Like all worms it infected its host, then used the Internet as the transmission medium to re-propagate itself to other machines.

I.B Sources of Software Failure

Identifying the sources of these software failures is important to us as we seek to prevent them. Sullivan and Chillarege [85, 13] analyzed the failures on a rigorously tested and deployed IBM mainframe MVS operating system. Sullivan and Chillarege classified the type of failure, its triggers, and the consequence to the customer. They found that memory overwriting bugs are more likely to cause a high priority bug report to be generated by a ratio of three-to-one than the general population of bug reports, even though memory corruption comprise just 15%-25% of that population. Memory corrupting bugs often allows the program to continue for some time (Byzantine failure) that potentially corrupts data and obscures the bug's identity, instead of stopping immediately at the point of failure. Another class of severe bugs overwrites the MVS recovery mechanism; it is discussed after the memory corruption bugs.

The causes of these memory corruption bugs are listed in Table I.1. Over half of the data-corrupting failures are directly due to memory mismanagement.

Table I.1: Causes of data-corrupting bug in MVS bug reports

Bug type	% of bugs
Buffer overflow	20%
Use of deallocated memory	19%
Use of corrupt pointers	13%
Unknown	13%
Type mismatch	12%
Synchronization	8%
Register Reused	7%
Uninitialized Ptr	5%
Undefined State	4%

Buffer-overflow accesses data outside its allocated region typically when an array index or pointers, exceeds the bounds. This either returns the incorrect value or corrupts nearby data. The second most prevalent bug is use of deallocated memory, which is also commonly known as the dangling pointers. In this case there exist two or more pointers with different uses of the same memory, caused by one pointer seeing the original use of the memory, and another pointer seeing memory that's been been freed once and been reallocated to it. The third bug occurs when a buffer overflow overwrites a pointer, that causes that pointer to reference the wrong location. The fifth set of bugs is due to memory corruption caused by multi-threaded synchronization problems such as race conditions. Here different ordering of thread execution causes different results in an unexpected way. Other types of concurrency problems can occur without having to corrupt memory as well.

Sullivan and Chillarege also classify the regular population of software failures that includes the memory corruption bugs described above, and all other bugs. The most severe non-memory corrupting errors prevent the MVS operating system from automatically recovering from failure, thus decreasing system availability. This strikes at the primary reason for having the MVS mainframe

Table I.2: Causes of severe non-memory corrupting failure that prevents automatic MVS reboot and system recovery.

Bug type	% of bugs
Deadlock	58%
Synchronization	22%
Undefined State	4%
Copying Overrun	4%
Data Error	3%
Unknown	3%

system marketed for high availability. Non-recoverable errors occur at a rate of 6.3% of the regular bug report population. Of these, the two largest causes are from deadlocks(58%) and synchronization (22%) errors, as given in Figure I.2, caused by multi-threaded concurrency bugs. Sullivan and Chillarege summarize that the three main causes of severe failure are memory corruption, concurrency and administrative errors.

The Sullivan and Chillarege's study succinctly provides the central motivation for this thesis. It covers two different areas of software bugs- memory corruption and concurrency bugs- that share the property that they have severe consequences and are difficult to isolate and repair in the field. However, there are techniques to avoid these bugs by systematically checking for failure conditions, though costly in terms of performance. Our approach is to provide hardware techniques that accelerate these checks to avoid the overhead.

I.C Silicon Fabrication Trends

In this section we look at silicon fabrication trends that enable special purpose hardware for software safety checks and error prevention. These trends also suggest future programming models will need to be multi-threaded cognizant,

as multiprocessors become the dominant means of improving performance.

Understanding the future trend of microprocessors comes in part from looking at the recent silicon semiconductor trends, and looking at the likely technology available to designers and manufacturers in that future time frame. Historic transistor scaling has caused the number of transistors per chip to increase at an exponential rate for the past three decades, doubling every 18 to 24 months, which famously is known as "Moore's law". Intel's microprocessors roadmap demonstrates this trend and is provided in Figure I.1. Gordon Moore, with visibility into the technology at Intel, has stated in 2005 that this trend will definitely continue for at least two more process generations but perhaps as long as twenty more years until silicon lithography reach limits when dimensions are the size of individual atoms [22]. While there are many barriers to further progress with transistor scaling, the historic trend is that the technologists have managed to find solutions, allowing Moore's Law to continue. This history is recorded in past International Technology Roadmap for Semiconductors (ITRS) [42] for future design and manufacturing direction. All previous ITRS "red bricks" where no technology is known to solve a particular problem, have been knocked off successfully[1].

While the number of transistors increases at a rapid rate and will do so into the future, translating that to performance has hit several obstacles. Microprocessor designers have historically found that performance could be derived from microarchitectural improvements and from frequency improvements. But taking Intel's roadmap as example, since 2004 this model has not kept us with the earlier 1.5 to 1.7X performance increase per processor generation. Microarchitectural improvements through instruction level parallel techniques has reached the

[1] Photolithographic technology limitations is perhaps the most famous of these "red-bricks". One example occurred at the 248nm wavelength of deep ultraviolet that the industry thought would limit feature sizes at around 0.25μ. Engineers succeeded in using optical correction techniques, and later introduced 193nm wavelength lithographic tools previously thought impractical, to overcome this limitation.

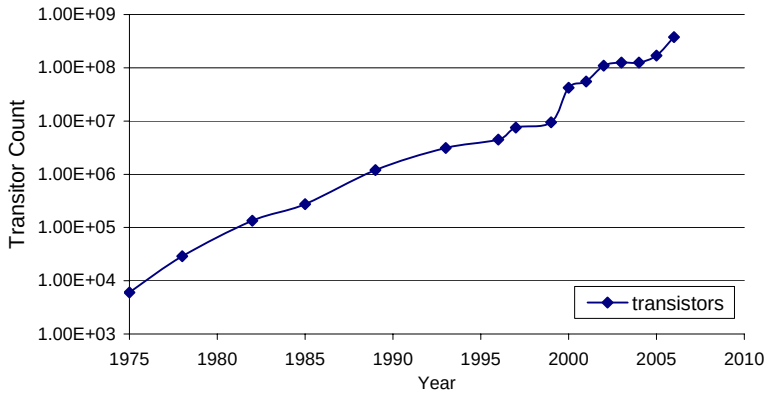

Figure I.1: Moore's Law: doubling of transistor per chip every 18-24 months on Intel Microprocessors

point of diminishing returns [64], due to scaling limitations of hardware structures. Increasing the instruction level parallelism enlarges the size of microprocessor structures increasing the power consumption and propagation delay of wires through them. For example register files, cache ports, and reorder buffers have a quadratic area growth rate of the number inputs with at best linear improvements in performance. There are other areas of microarchitectural improvements that scale better such as L3 caches that reduce overall memory latency, or vector processing units that increase parallelism without needing data bypassing. A second means to improving microarchitectural performance is to scale up the frequency of microprocessors. Historically Intel's microprocessor's clock rate approximately doubled every process generation, matching Moore's law (hence frequency used to be a corollary law). Of this 2X, half the increase comes from transistor scaling of 1.4X per generation, while the other half comes from reducing the logic depth per pipestage at about 1.4X [46]. However frequency speedup has encountered two barriers. First switching power overhead is proportional to frequency, and microprocessor design hit the "power wall" [67] in dramatic fashion in 2004, where an Intel microprocessor successor for the first time did not increase the

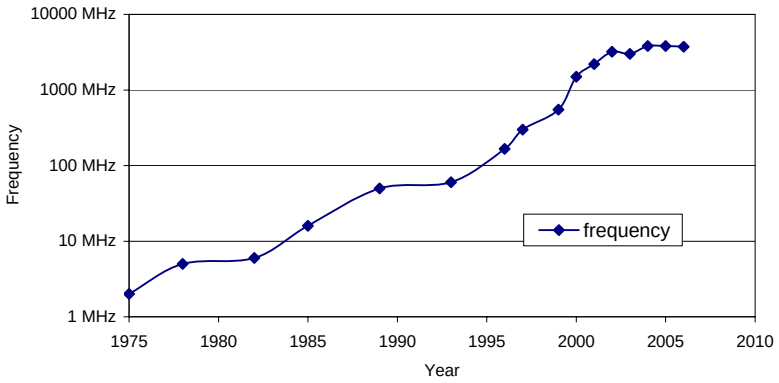

Figure I.2: Frequency doubled until 2003 on Intel Microprocessors

frequency over the earlier version. This is seen in Figure I.2. Since then Intel has not increased processor frequency beyond 3.8GHz though some smaller rate of increase is expected in the future. The second barrier is that there is diminishing returns as one partitions logic into more pipestages, due to clock jitter and sequential circuit overhead. Further frequency improvements will likely be limited to transistor scaling effects, leavened by power concerns.

One solution to performance scaling problems is to consider multicores on a single chip, as has been done by Intel and others. Figure I.1 graphs the transistor count of the largest Intel general purpose processor chips of a given year, and year 2006 already has a dual core chip. This Intel Smithfield chip has 376 million transistors using 65nm process node technology. Contrast that chip with its ancestor, the 1997 single core Intel Klamath chip with 7.5 million transistors at 0.35μm that already has all of the significant high-performance features like out-of-order execution, two-level branch predictor, multiple execution units, and multiple levels (two) of cache, of its larger relative. Intel expects to release sometime this year, the Montecito server chip with an even larger 1.76 billion transistors. One can see that current chip budgets already support having several high performance general-purpose cores on a single chip, even when using larger

caches. Alternatively one might consider specialized functions such as security and debug because there is such a abundance of transistors now.

Because fine grain parallelism finds diminishing returns while simultaneously there are many chip level procesor cores, using coarse-grain task or thread parallelism becomes preferable. The Chip Multiprocessors (CMP) [64, 46] are loosely coupled using cache coherence to communicate between the cores, where each runs separate threads. Combining the threads to solve a single task requires coordination between the threads that is error-prone as the Sullivan and Chillarege data demonstrates. Also such multi-threaded programs only provide useful speedup if the partitioned work can be done in parallel, that is work independent of one another. If data dependencies, that normally cause ordered execution, is only present part of the time, most parallelization techniques must be conservative and prevent potentially unsafe parallel partitioning. Our work will later demonstrate a solution to both problems.

Duplicating cores is not the only solution. These extra transistors enables specialization of functional units to improve performance in ways that general purpose processors are not able to. This was explored in the context of cryptographic Application-Specific-Processors by Wu et.al. [94] where certain bit manipulations were done much more efficiently in custom hardware, and several specialized processors were run in parallel, to assist a general purpose processor. Because of the overhead of software checks, we seek to create specialized functional units for software security later that reduce the overhead of software checks.

I.D Goals of Thesis

Software bugs are program behavior that are unexpected from what the software designer intended, which malicious hackers can often exploit on Internet

connected computers to remotely execute code or deny services. One common bug often exploited is called a buffer overflow. In this thesis we examine why such a simple bug is so frequently targeted to the detriment of so many systems. We then examine a simple but powerful software technique called bounds checking to defeat it. It comes with performance caveats that are addressed by this work.

Buffer overflow is a kind of memory bug caused by writing or reading beyond the bounds of allocated memory. Reading beyond the bounds may leak information. Writing past the bounds causes memory corruptions that often manifests itself non-deterministically because the bad data may not used hence observed by the program till long past the point of overwrite. Bounds checking forces non-deterministic data corruption to become deterministic, by immediately detecting if the buffer bounds have been exceeded, and before any memory is written. Dangling-pointers is another memory corruption bug, but less frequently seen. It occurs when memory has been freed, but is still used, leading to two different overlaid uses of that memory. When both write that memory, they will corrupt each others memory image. Protecting against both bugs is considered to be sufficient protection against the most important cases of memory corruptions by some [5].

We improve the performance of memory checks that prevent data corruption by software optimization and hardware acceleration:

- Identifying performance overheads of software memory checks through hardware performance counters and simulation. The causes of overhead are extra instruction execution, data-copying overhead, and branch mispredictions.
- Optimizing the organization of data association with the software checks. We found that associating the meta-data with the object rather than pointer reduces copying costs.
- Eliminate unnecessary checks outside the scope of code needing security

checks for dangerous data. We developed an interprocedural type-based analysis to discover code that touches safe and unsafe data.

- Provide a modified Instruction Set Architecture and hardware datapath to efficiently accelerate software checks. This eliminates unnecessary meta-data loads and reducing instruction fetch requirements.

A second trend we examine is multi-threaded programming for Chip Multi-Processors (CMP). Semiconductor scaling provides far in excess of transistors than needed for a single general purpose CPU core, consequently many CPU design companies will provide multiple cores on a chip. These multicores could be used to accelerate single program performance by splitting it apart into multiple threads, and executing the threads concurrently on the CMP cores. This multi-threaded program needs to share data across threads, requiring coordination of the data updates to maintain the original single threaded execution behavior. Lack of coordination amongst shared data typically results in non-deterministic execution called a race condition, and is often considered a software bug. We examine a technique called Transactional Memory (TM) that eliminates race conditions yet allows for parallelism.

We propose multi-threaded execution models for multi-core systems that avoid data-races and deadlocks by using Transactional-Memories. Memory maybe speculatively modified in parallel, but data conflicts detected between different threads force serial execution of the threads that avoids. One of the main limitations of prior techniques handling what happens when their speculative state no longer fits in cache due capacity constraints or when the thread must be context switched out.

- Propose a virtualized Hardware Transactional Memory model called Page-Transactional-Memory (PTM) that enables execution on general purpose computers, that is cached, multi-user systems. This backs the transactional

state in virtual memory using additional hardware assistance, allowing that state to overflow the cache, context-switched out or saved to disk. PTM features faster aborts and commits than all prior virtualized transaction work, by eliminating the need to copy data.

- Provide the first performance results for an important prior Transactional Memory technique called VTM

II

Software Check Optimization

II.A Motivation for Memory Checks

Buffer overflow attacks are still the most common internet exploit, and if current trends hold without intervention, it will remain so into the future. The common form of buffer overflow attack takes advantage of program bugs that allow writes of malicious data past the allocated memory boundary, such that the adversary can execute code of their choosing. Figure II.1 reports the number of CERT Security Alerts caused by different software bugs for 2004, 2005 and 2006. CERT Alerts are signifcant because they represent a security danger to the national computer infrastructure due to an exploit (e.g. worm, virus) or the potential for one. There are many categories: *Buffer-overflow* accounts for the majority at 55%, 50% and 73% for those respective years. The following two paragraphs describe the other bugs denoted by *italics*. Virtually all of these buffer overflow alert carry the warning that they allow remote code execution. Earlier CERT Security Advisory statistics before 2004 are found in [88], and have the same proportions of buffer overflow exploits to total, with variation year-to-year. Despite increased industry vigilance due to worm outbreaks in 2001, buffer overflow attacks still remain the most dangerous and common exploit.

Two other types of bugs found in the 2004-2006 CERT Alerts are important for this thesis; the rest we contrast to justify their exclusion. First *dangling-pointers* have three CERT Alerts reports in 2004. These are caused by bugs where memory is freed by one pointer, and still used by another pointer causing memory corruption. Second there are *data-race* alerts reported one each in 2005 and 2006, caused by unsynchronized memory update in multi-threaded execution. Both have the potential for memory corruption which allows execution of remote code.

The rest of the bugs are beyond the scope of this thesis for a number of provided reasons. The second largest category of CERT Alert exploits is *privilege* escalation (23.9%) caused by the improper use of commands, such as active X controls, or Internet URL exploit. In some cases this enables the remote execution of code, e.g. running a script by an external agent. Automated securing of these bugs is difficult because of the variety of controls to consider, and because of the likelihood of stopping a task that the programmer intended. One subcategory distinguished separately is privilege escalation to *leak* private information like passwords to the adversary. Format string exploit on buggy *sprintf* functions differs from privilege escalation in that it corrupts memory, to allow remote code execution. Fortunately sprintf is only one alert of sixty-eight total in 2004-2006; the lack of exploits in 2005 and 2006 may potentially be due to bug fixes. Denial-Of-Service was the third most common attack (7.0%), but arguably less severe than remote code execution. Memory *layout* errors cause memory corruption, that potentially may execute remote code. The two examples are caused by incorrect layout of arguments to RPC function calls, or reading of raw binary file data. These occur at such a low level of the operating system, it makes it difficult to include in the scope of this thesis's investigation.

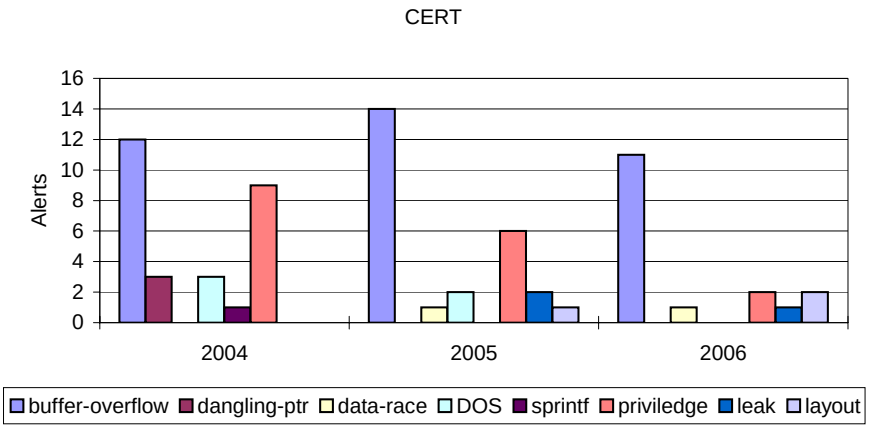

Figure II.1: Cert Alerts 2004-2006

II.A.1 Bounds Security Exploits

Buffer overflow may intentionally alter instruction control allowing the remote execution of code on a computer. We first describe the most basic form of buffer overflow on stack arrays called "Stack Smashing", as popularized by AlephOne [1]. Consider a fixed size buffer allocated on a stack. A buffer overflow can occur when the application copies external data into the buffer but does not check for the size of the input data, especially while copying strings (e.g. using strcpy). In the string case, the end of a string is determined by checking for the null character which maybe intentionally missing. An adversary can exploit buffer overflows by copying executable code or malicious data into the memory buffer. When writing to the stack array, the adversary can overwrite the return address with his own value targeting malicious code as illustrated in Figure II.2, and the control will jump to execute the malicious code upon return from the function. Stack smashing has been used in CodeRed, Nimda, and Slammer just to name a few Internet worms.

Buffer overflow remains by far the most prevalent and consequently dangerous of CERT Alert bug. One reason is that it maybe exploited in so many

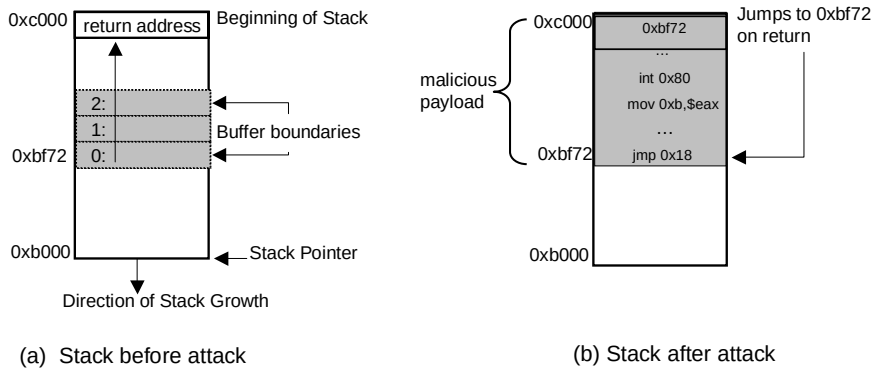

Figure II.2: Stack Smashing example

different ways. Once a defense technique is raised against an exploit, another variant appears circumventing the defense. A phalanx of defenses such as sentinels [20], reordering of stack variables [24], better code reviews, and varying the base of the stack, has blunted "stack smashing" exploits. However in the continuing arms-race this lead to heap exploits involving manipulating malloc/free data structures in Lea's Unix heap memory allocator [45] and similar data structures in Windows XP heap allocator [4]. As heap attacks do not touch the stack, stack protection schemes do nothing for this exploit. A heap attack manipulates free list pointers to allow the adversary to overwrite arbitrary memory locations, and has been used on Sudo, Secure Locate, Traceroute, and Netscape browser. This capability allows heap attacks to change function pointers, to point to malicious code, that will then get executed when the function pointer is invoked. Because heap exploits are flexible there are many variations. One example is the Global Offset Table (GOT) used for locating the run-time relocation data, including function pointers in dynamic link libraries. GOT is just an array of pointers, initialized by the dynamic linker. An adversary can corrupt a `libc` function pointer in GOT, then when the corresponding function is called, it can result in execution of malicious code written by the adversary. Another example are C++

virtual methods that are invoked through function pointers.

The above description of buffer overflow exploits might imply that they only perform malicious code execution, however data manipulation by buffer overflow is another dangerous capability. The adversary can modify the input strings to system calls such as "system" or "popen", by overflowing the buffer location adjacent to the input strings. These input strings determine the Unix command to be executed and hence the adversary gets to execute a command of his choice. Buffer-overflow attacks can also be used for denial-of-service by simply corrupting data in a program that causes the program to crash such as overwriting pointers with null values.

II.A.2 C and C++ Code Base

Much of the recent buffer overflow exploits occur in programs written in C and C++ due to those languages unsafe features. Though there exist safe languages protected by built-in run-time bounds checking and by full type-safety such as Java and C#, the C and C++ languages have a large and important installation base. For example, the Redhat 7.1 Linux distribution is composed of the following source languages: C is 71% and C++ is 15% [90]. Virtually all important operating system kernels such as Linux, BSD, other flavors of Unix, and Windows are written in C or C++, as are many potentially vulnerable device drivers. Even components of the Java and .Net virtual machine are written in unsafe languages (C and C++). C and C++ have the largest installed base of code as measured by open source projects. A snapshot of the *FreshMeat* Open Source database on 12 January 2005 in Table II.1 lists the prevalence of the top 5 languages having a total of 85.6% of projects. Combined C and C++ have 51.4% while Java has 18.1%.

Table II.1: Distribution of open source projects by languages

language	% of projects
C	34.3%
Java	18.1%
C++	17.1%
Perl	16.1%
Python	8.6%
Rest	14.4%
C,C++	51.4%

II.B Implementation of Bounds Check

Bounds checking is performed when a pointer is dereferenced to access an object or buffer to ensure that the access is to the intended object. Array access is treated similarly as it can be decomposed to pointer dereference and check on the effective address generated by the addition of the index and array base. Bounds check is done by keeping track of low and high bounds information of the objects and comparing them against the effective address used while dereferencing a pointer as seen in Figure II.3(a). It is called *two-branch* because an inlined implementation will have two branches assembly instructions. The base of the allocated memory region for the object defines the *low* bounds. The maximum extent of the region, that is the base plus size of the object, defines the *high* bounds. Bounds checking defeats buffer overflow attacks at the point it attempts to exceed the object's bounds, by raising an exception or jumping to a handler, thus preventing any out-of-bounds data from being read or written. For the code sequences used to do bounds checking shown in Figure II.3(a), we assume that *low*, and *high* meta-data are given by some mechanism discussed later in section II.E.

A second form of bounds check uses the low bound and size of the

object, that combines the low and high bound comparisons as illustrated by Figure II.3(b). For the high bounds check, the comparison is straightforward where the offset of the pointer from the low bounds is compared against the size of the object. Exceeding size results in raising error handling. For the low bound check, it uses wrap-around property of unsigned subtract on negative numbers (low out-of-bounds) to become a large positive number - typically larger than size. It is possible for an extremely distant pointer and large object that wrap around to appear in-bound, though this is exceedingly unlikely. We provide this more efficient *single-branch* bounds check for comparison. Figure II.3(b) assumes the availability of *low*, and *size* meta-data.

A third software bounds check is to call a dedicated bounds instruction that exists in the x86 instruction set. *bounds* instruction is functionally identical to case II.3(a) but clearly has fewer instruction. We compare different bounds checks implementations in greater detail in section II.D. The bound instruction has only two operands as shown in Figure II.3(c). The first input (`ptr`) specifies a general purpose register containing the effective address that needs to be bounds checked, and the second input (`b_ptr`) is a memory operand that references a memory location containing the low and high bounds. On execution, the bound instruction verifies if the first-operand's address is between the low and high bounds. If the bounds are exceeded, it issues a bounds-checking exception. The in-memory bounds information constrains the check code-generation because it is not possible to encode constant bounds in the compare instructions as with Figure II.3(a) and (b) sequences. The code generator uses meta-data given with the pointer and creates additional in-memory meta-data for arrays. The organization of this meta-data is discussed further in section II.E.

We consider the check code sequence to help us understand some of the performance overheads seen with bounds checking. Initially we are interested

```
start:
   if(ptr >= low) then low_ok
     trap
low_okay:
   if(ptr < high) then high_ok
     trap
high_okay:
```

(a) Two Branch

```
start:
   tmp=(unsigned)(ptr-low)
   if(tmp < size) then ok
     trap
ok:
```

(b) Single branch

```
start:
   bound ptr, b_ptr
```

(c) x86 bound

Figure II.3: Psuedo-Code of Bound Check and minimum number of assembly instructions. (a) Two Branch - 4 inst (b) Single Branch - 3 inst (c) Bounds - 1 inst

in how many instructions are present in a check as excess dynamic instructions reduce performance, and the number of branches due to the cost of branch mispredictions. Just for this approximate comparison we assume meta-data is directly loaded from memory allowing the CISC architecture to fold meta-data loads into another instruction, or converts them into a *move* instruction. Also a single *if* statement is converted to at least two instructions - *compare* and *conditional jump*, with potentially one or more *move* instruction(s) needed to marshal values to registers. Using these simple rules we find that the minimal number of executed x86 assembly instructions are four for two-branch (a), three for single-branch (b) and one for *bound* op (c) discussed next. As their descriptions imply, the branch counts are two (a), one (b), and zero (c).

II.B.1 Representing Bounds Information

A bounds checker needs bounds information to perform its verification. We will now describe how the bounds meta-data information is organized and recovered when a pointer is dereferenced.

We categorize the bounds checking techniques on how it manages the bounds information. There are three approaches to associating the bounds meta-

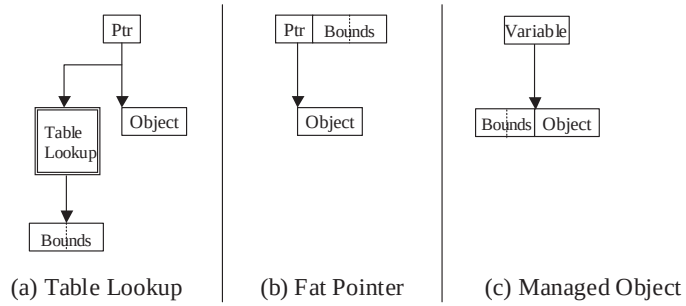

Figure II.4: Bounds Check Meta-Data representation

data to the object as shown in Figure II.4 - (a) meta-data table, (b) fat-pointer, and (c) adjacent to the object (referenced by pointer).

- Table Lookup

  The first approach maintains bounds meta-data separate from the pointer in a table. This makes it unnecessary to change the memory layout of the pointer, reducing the effort to port non-bounds checked applications. To lookup the bounds meta-data for a bounds verfication, it does a lookup on the bounds meta-data table with the pointer address as the index. The C language complicates table lookup by allowing interior pointers that change the addresses passed, though the same meta-data should be returned. Consequently the table is not organized as a fast hash lookup that requires exact match, but instead uses tree search with additional runtime cost that allow ranges of addresses to return the same meta-data. *Table lookup* is illustrated in Figure II.4(a).

  Jones and Kelly [44], and the derivative CRED by Ruwase and Lam [76] uses Table Lookup. CRED fixes the problem of out-of-bounds pointers found in the earlier Jones and Kelly. By definition out-of-bound pointers violate bounds and are forbidden by language specification, but in practice are often found in code and ignored so long as they are not referenced. Both table-

lookup implementations report much higher run-time overhead than fat-pointer based bounds checkers (10X or greater).

- Fat Pointer

  Fat pointer modify the representation of a pointer to include the low and high bounds [25, 60] meta-data. Figure II.4(b) shows that the bounds are stored with the pointer in memory adjacent to the pointer value. Because of the direct association, this *fat pointer* format can directly handle interior pointers without any special consideration, and because no table lookups are necessary, it is fast. As a consequence this thesis uses *direct-assocation*. Austin et.al. [5] proposed a variation of the fat pointer *directly assocating* the base and size representation to the pointer, to be used with single branch bounds check as described by Figure II.3(b).

- Managed Object

  Managed languages such as C# and Java name their objects without exposing the pointer to the programmer, making arbitrary pointers addressing impossible. Because the pointer always references the base of the object, bounds information can be fixed at a given distance to the object, that is known at compile time. Consequently lookup is fast. *Managed object* meta-data supports other features like managed memory e.g. garbage collection, dynamic typing, etc. This model is illustrated in Figure II.4(c).

For the rest of the thesis we consider only directly associated meta-data. This is due to the performance overhead of table lookup, and the frequency of interior pointers with C/C++ that prevent the use of managed objects.

II.C Software Memory Check Related Work

This section reports related work for software memory checks: We describe alternatives to fat-pointer bounds checking or other software checks to prevent memory corruption. We describe several software optimization techniques to minimize the number of checks. Because implementations exist for software checks, we can compare their run-time performance to our software optimization techniques to contrast the differences of these approaches. Third we report related hardware enhancements to reduce the performance overhead for memory checks.

II.C.1 Bounds Checking and Their Alternatives

Bounds Checking

As mentioned in subsection II.B there are two main styles for C/C++ bounds checking based on how they associate meta-data, which is through table lookup and directly associated meta-data in fat-pointers. Bounds checking techniques are represented by CRED [76] for table lookup, and McGary [25] for direct-association of the meta-data. CRED is an improvement of the earlier Jones and Kelly table lookup bounds checker [44]. CRED also reduces the numbers of bounds checks through an optimization limitting checks string object references. Both table lookup mechanisms have large run-time overheads. Using our set of benchmarks we measured an overhead of 1370% for base CRED (Jones and Kelly is similar). We then measured CRED, where their string optimization reduces the overhead to 120%. Recently CRED was improved by using BDD pointer analysis to filter out non-string types in their string-only optimization [7], reducing runtime overhead to around 100%. CRED was also used to tolerate bound check failures by safing the violation continuing, rather than raise an exception [74].

In this model, out-of-bound read returns back a nonsense value to prevent information leakage, while out-of-bound writes do not update memory to prevent memory corruption.

McGary [25] and Cyclone [43] perform bounds checking with fat-pointers meta-data. While McGary's inlines checks and bounds updates, all other bounds checkers we have examined invoke procedure calls to perform these tasks which explains in part McGary's much reduced overhead relative to them. Our measurements of McGary's reports an overhead of 72%, and in this thesis we present optimizations to reduce this overhead even further. Austin et al. [5] does bounds checking and dangling pointer checks using some additional meta-data state for a complete pointer verification.

Protecting Code Pointers

An alternative to bounds checking are pointer protection techniques represented by ProPolice [24] and PointGuard [19], that provide some of the lowest overhead of software techniques. We measured 1% performance overhead with ProPolice on our benchmarks, by using a "canary" - a small piece of data placed on the stack between buffers and the return address to check for buffer overflow. Upon exiting a function, the canary is checked, and if overwritten by some overflow attack, the thread is terminated and the incident is reported. ProPolice also provides variable reordering to further protect stack data. ProPolice is a refinement of the earlier StackGuard [20], and both protect stacked memory but not other memory such as heap. An improvement is PointGuard [19] that encrypts all pointers with an XOR function providing better coverage than ProPolice. In PointGuard, an overflow attack overwrites the encrypted pointers, which decrypt to most likely meaningless values. While much stronger, PointGuard does not prevent attacks that overwrite data not containing pointers, and PointGuard is

susceptible to information leakage during reads of out-of-bound data. PointGuard only protects pointers.

An advantage that bounds checking has is that it prevents the buffer from ever being overflowed, so the attack can be handled without crashing the program. ProPolice, StackGuard and PointGuard detect overflow only after it has been overwritten, which makes it hard to recover from an attack.

II.C.2 Compiler Optimization

There is a large body of work involving reducing redundant bounds checking in loops and acyclic code using compiler data-flow analysis. The earliest work was done by Markstein et al. [53] who proposed using a variation of Common-Subexpression-Elimination (CSE) for bounds on acyclic code, and Loop-Invariant Code Motion (LICM) on loops as two separate bounds redundancy elimination steps. Both optimizations match and eliminate redundant checks only if the exact same arguments are present as some earlier check. The approach in this thesis takes includes Markstein et al. CSE and LICM redundancy elimination. Subsequent papers improve upon the bounds redundancy by being sensitive to ranges of addresses instead of exactly the same bound expression to eliminate more redundant checks [49, 29, 11]. The idea here is to find an early bounds check that more tightly specify a bounds than a later check, thus making the later check redundant. The first of these performed this check for acyclic code [29]. Kolte and Wolfe [49] improved the check framework by using Partial-Redundancy-Elimination to generalize and combine acyclic and loop redundancy elimination with this range framework. Bodik, Gupta and Sarker [11] performed this elimination in the Java run-time, requiring an efficient graph representation of the ranges to reduce run-time optimization overhead.

Optimization is also possible using a type system framework to prove

the safety of memory references, and eliminate unnecessary checks on safe ones. CRED [76] globally applies bounds checking only to string types, observing that buffer overflow exploits occur on pointers to strings and string arrays. CCured [60, 35] uses type information to eliminate type-safe checks. For bounds checking they discover whether a pointer takes the result of some arithmetic operation that might cause the pointer to go out-of-bounds. CCured eliminates checks for completely safe pointers, while for indeterminate pointers use run-time checks, and provably unsafe pointers are flagged at compile time. Such strong properties has the disadvantage that CCured suffers from false errors due to un-dereferenced out-of-bounds pointers.

Austin et al. [5] implemented bounds checking and dangling pointer checks, claiming that using both checks provides complete pointer safety. This paper also proposes doing run-time optimization of redundant checks by marking a bit that indicates it is already checked, but is cleared if the arguments changes. Bounds checks use a single branch bounds check discussed in subsection II.B and the dangling pointer tag check described later. They use a data structure that is a mixture of direct-association for the bounds, and table based capability tags for the dangling pointers. Searching through the capability table and using directly associated meta-data for bounds checking is moderately expensive. Austin et al report an execution overhead in the range of 130% to 540%.

II.C.3 Comparison

Table II.2 summarizes a comparison of prior techniques to protect against buffer overflow attacks on the Spec 2000 integer programs we examined. The first column shows the slow down experienced from the technique versus no bounds checking, where numbers labeled $*$ are the average taken from the papers for comparable benchmarks, and the results for the rest of the techniques we measured

Table II.2: Comparison of prior techniques.

Technique	slow down	buffer overflow
Flow[84],Minos[21]	1-5%*	complete
Dynamic Taint[62]	5x*	complete
Prog Shepherd[48]	15%*	partial
ProPolice[24]	1%	stack frame
PointGuard[19]	10%*	pointers
CCured[60]	40%*	complete
CRED[76] (base)	1370%	complete
CRED[76] (string)	120%	strings
McGary[25]	72%	complete
bnd-array	40%	complete
all-interface-write	24%	interface

ourselves. The last column summarizes the type of protection provided against buffer overflow attacks: *Complete* specifies overflow protection for all memory refereces, while *partial* specifies limited protection since memory maybe corrupted. *Stack-frame*, *pointers*, *strings* protects specific types of data from corruption, but not other data. *Interface* protect data modified by the interface from overflowing buffers.

Several important properties of buffer overflow protection techniques are summarized in Table II.2. The run-time measurements are done on an Athlon processor. Bounds checking techniques have higher overhead (1370%) than non-bounds checked techniques using hardware assistance (measured 1%), but as noted earlier, the non-bounds check techniques have certain functional limitations such as being limited to stack or being imprecise. Full bounds checking using direct-associated meta-data (McGary 70%) is significantly lower overhead than table lookup (CRED 1370%), and in subsequent sections direct-associated meta-data will improve to 40%, and further 24% with optimization.

II.C.4 Hardware Proposals for Security

Recently several techniques have been proposed to dynamically track all data coming from any untrusted source, and if that data gets executed, then execution is marked as a potential attack [84, 21, 62]. Suh et al. [84] and Crandall and Chong [21] examined using hardware support to tag each memory word with a *Taint* bit indicating if the data put into memory was stored there from untrusted sources like the network. Then if any memory address is executed with this taint bit set a buffer overflow attack is flagged. Assuming this hardware support, they were able to provide this protection with only a few percent slowdown. More recently Newsome and Song [62] implemented this approach using a dynamic binary emulation to track data from external sources. They were able to do this without any hardware support, but with a slowdown of 5 times over native execution (ignoring the baseline overhead of dynamic emulation system used).

Program shepherding [48] is another approach that protects key control transfer points such as indirect jumps, and uses a binary optimization system to detect when control flow is being transfered to data regions. This is done efficiently by using a run-time binary rewriting tool that guard branches. It guards against these attacks, but does not prevent code-injection attacks like overwriting a function pointer in the global offset table.

These techniques do not protect against data overflow attacks, only code execution attacks. An adversary may still perform data attacks such as modifying function arguments or system calls. In comparison, the bounds checking optimizations we examine in this paper provides protection for both types of attacks, without requiring any additional hardware support, and keeping the overhead to 24% on average.

Tuck et al. [88] proposed hardware accelerated encrypting pointers to protect against security intrusions much like PointGuard [19]. In comparison, our

focus is on providing the precise reporting of check violation instead of obscuring pointer values for bounds and dangling pointer checking.

II.C.5 Hardware Support for Debugging

Recently there has been significant interest in providing hardware support to assist debugging. Zhou et al., proposed iWatcher [97] to monitor accesses to memory locations. The memory location that needs to be monitored and the monitoring function that needs to be executed when a monitored memory location ins accessed, are specified through a system call. A bit is associated with each word in the L1 and L2 caches, so that the hardware knows which locations need to be monitored (information will be lost when a block is evicted). A software table is used to map the addresses of monitored locations and the monitoring function corresponding to them. When there is an access to a monitored location, the software table is searched to get the monitoring function, which is then executed.

The goal of iWatcher is to provide efficient watch points to monitor memory locations for debugging. iWatcher can specify a range of memory to observe if memory is overwritten for buffer overflows. It does not provide a general framework for performing bounds checking for all pointers and array references, which requires storing and maintaining meta-data information for the checks that iWatcher does not do.

Witchel et al. proposed Mondrian Memory protection [92]. It protects memory across all the user and supervisor processes, at fine granularity down to a word using hardware support, mediated by the kernel. They demonstrate its applicability to fine grain protection for malloc headers surrounding allocated regions. Applying bounds checking to Mondrian Memory will have even finer granularity protection regions and many more of them than their malloc exam-

ple, incurring more communication overhead through the kernel to the protected hardware resources unlike application level only bounds checking.

In DISE [17], Corliss et al. proposed a programming interface to the dynamic instruction macro-expansion found in modern processors. A sequence of functions (essentially micro-ops) are associated with an instruction and are dynamically injected into the pipeline when that instruction is executed. They applied their technique for achieving memory fault isolation, which ensures memory access in a modules on the permitted data or code segments by doing micro-op address checks. In their follow up work, Corliss et al. [18] used the DISE mechanism to efficiently implement watchpoints that will be useful for implementing interactive debuggers. But DISE has not looked at providing complete bounds checking and dangling pointer checks, which require mechanisms to track the meta-data efficiently. Our research is complementary, since it would use support like DISE to perform the micro-op expansion of the meta-data checks.

The memory subsystem can be modified to protect memory regions for debugging. SafeMem [70] alters ECC to provide watched memory boundaries for detecting buffer overflows and to identify infrequently accessed objects for memory leak detection. They cite 0.7% to 29.4% overhead, which is comparable to our hardware performance overhead. While SafeMem cleverly makes use of existing hardware functionality, it cannot guard against all buffer overflows. Also, the overhead of the ECC boundaries are governed by DRAM word size. Current DRAM DIMM modules have 8 byte words, and for small objects sizes, SafeMem's overhead may be expensive.

AccMon [96] is based on the observation that a memory location is typically accessed by few instructions. Hence, they capture this invariant set of PCs accessing a given variable, and classifies any access by an outlier instruction not present in the invariant set as a potential cause of error. AccMon does not

implement comprehensive checks like bounds checking or the dangling pointer checks.

FDR [95] and BugNet [59] continuously collect a trace representing the recent execution of the program. When the program crashes, the collected trace is useful for debugging by deterministically replaying the last several millions of instructions executed before the crash. ReEnact [69] is another proposal that continuously records memory accesses in a shared memory system and when a data race is detected based on certain heuristics, the multi-threaded program is replayed again and again to characterize the data race.

Patil and Fischer [65] provided bounds and dangling pointers checks using a second "shadow" processor running on a separate co-processor to accelerate checking. The original program runs ahead while a sliced checker process follows the main thread, synchronizing at system calls with a combined run-time overhead of 10%. Their solution involves source to source translation to create a completely different shadow process which needs to be executed concurrently on a different co-processor. The two processes need to be kept in synchronization to ensure that they are executing along the same path in the program. When compared to this approach, ours is very lightweight and requires less hardware resources.

II.D Performance Analysis of Bounds Checking

Of the different bounds check sequences described in the previous section, the directly associated meta-data bounds check provides the best performance base. Our goal is to reduce directly-associated meta-data overhead further, to ease adoption. We consider improving the performance of bounds checking at the instruction level in this section, and at the data layout level in the following section II.E. Within this section, it is divided into a description of performance

analysis methodology using CPU performance counters, the code generation of the bounds checks sequences by the compiler, and the benchmarks used in this analysis. Additional information about how bounds for the bounds checks are generated in the compiler is available in the subsection II.D.4.

II.D.1 Performance Profiling

As there already exist bounds checking compiler tools, we want to perform hardware measurements to get an accurate and complete understanding of the overheads involved. To dig deeper, we desire microarchitectural data to identify the source of overhead when these instructions are executing. Fortunately performance analysis hardware is built into many CPU's, with the most common being counters that record microarchitectural events such as instruction count, cycle count, branch misprediction count, cache misses count, etc. By comparing the performance counters of the bounds checked binary against one without, we can see the performance change due to the bounds checking.

We use Mikael Petterson's Linux kernel patch [66] that instruments Linux to record these counters. Pettersson's tool provides application level access to these low level hardware performance counters and handles operating system details such as saving and restoring state during context switches. Over this we build an analysis application to access the counters we are interested in while it runs a target program.

For each result, we execute the program three times to factor out random system effects (e.g. disk IO) while executing on a real processor. Our primary results are based on the AMD Athlon 2400+ XP (K7), including all of our hardware performance counter numbers.

II.D.2 Bounds Tools

Our compiler is based on the Greg McGary's bounds checker [25], that patches the 2.96 GCC compiler, and generates bounds check using directly-associated meta-data (fat-pointer) and the branch sequence check in Figure II.3(a). We made several general modifications for bounds code generation and compatibility: First we extend bounds check code inlining to support single branch and x86 bounds, described in Figure II.3(b) and (c). Second we wrote bounds checking wrappers for many library functions. Third we modified the GCC value numbering optimization to recognize the bound instruction to eliminate redundant bounds corresponding in an acyclic region, and to perform simple loop hoisting of bounds checks. These optimizations were first done in Markstein et.al. [53]. Last, we modified the compiler to let us do inter-procedural analysis and optimization described later in Section II.F.

As our goal was to reduce the overhead of bounds checking while maintaining the security coverage that it provides, we needed to measure both performance and security. We used the SPEC 2000 Integer performance benchmarks running on AMD Athlon hardware using performance counters. We provide runtime results for all seven of the C SPEC 2000 Integer programs that compile and run correctly with our baseline McGary `gcc` compiler. The remaining four (`gcc`, `perl`, `gap` and `vortex`) failed to compile with the baseline McGary compiler we started with. All are compiled using GCC with the -O3 option.

We also verified the security of our bounds checking against several real program bugs from the AccMon benchmark suite in [96] and Wilander benchmark [91]. Accmon researchers found buffer overflow bugs in open source programs, occurring for example at the command line parsing or strcpy. In each case our compiler detected the buffer overflow as described in Table II.3. Wilander's benchmark tests for 18 different malicious code injection buffer overflow attacks.

Table II.3: Coverage for UIUC AccMon benchmarks

Benchmark	Bug
AccMon	
bc 1.06	string parsing overflow
gzip 1.2.4	strcpy overflow
man 1.5h1	cmd line parsing overflow
ncompress 4.2.4	cmd line strcpy overflow
polymorph 0.4.0	cmd line strcpy overflow

This includes varying stack and heap memory, attacking return address, or a data structure function pointer, etc. Our modified compiler passed all 18 tests.

II.D.3 Bounds Code-Generation for Directly-Asssociated Meta-Data

As x86 bound instruction promises to reduce dynamic instruction count and branch mispredictions, we modified the GCC code generation to emit this instruction as described earlier in section II.B. The bound instruction has two operands as shown in Figure II.3(c). The first input (`ptr`) is a register containing the effective address to be bounds checked, and the second input (`b_ptr`) is a memory operand referencing memory with the low and high bounds. We also modified the compiler to generate the single branch bounds check (see subsection II.B)

We would like to use the bound instruction for all bounds checks, but an issue arises when we try to do bounds checking for global and stack arrays, as they are referenced in C and C++ without pointers. To allow the `bound` instruction to be used in this case, we allocate memory adjacent to statically declared arrays to hold the meta-data bound information, which will be created and initialized when the arrays are created. Since the bound information for the global and local arrays are now located adjacent to the object data memory, the `bound` instruction can take in the corresponding memory location as its second

operand to do bounds checking for these arrays. This memory location is located at a fixed offset from the base of the array. Setting bounds meta-data in memory requires changes to the compiler data structure layout module as well as the code generator. Out of the techniques we examined, we found this method to provide the best performance for bounds checking array references, and we call this configuration *bnd-array*.

Also we want to understand to what extent one can reduce the bounds checking overhead, if one implements the `bound` instruction as efficiently as possible in a processor. To do this, we generate a binary like *bnd-array*, except we remove all the bound instructions, while keeping all the meta-data and instructions to maintaining meta-data. We call this configuration *bnd-ideal*. *bnd-ideal* will still have overhead from generating and maintaining the bounds information in memory, and from copying the fat-pointer caused pointer assignments or with pointers passed through function parameters.

II.D.4 Generating Bounds Information from Type Information and Code Generation

Bounds checker need bounds information to perform its verification. Bounds information of static objects is determined completely by the compiler, while bounds information of dynamic objects is determined partially at run-time if its statically sized or completely at run-time otherwise. Bounds information is generated using the size and memory location. Type information provides the size when the object is statically sized. The low bounds of any object is determined by its memory allocation location. Completely static objects such as global variables are placed in fixed memory and have low bounds set at compile-time. Statically sized objects in dynamic memory such as stack variables, have low bounds set at run-time when the address of the object base is determined. Dynamically

sized objects generate bounds differently, as type information is insufficient. C provides heap and stack dynamic memory allocation routines that take size as an input. These routines are modified to generate bounds from the supplied size, and from the low bounds obtained through the allocated object pointer. High bounds information is simply derived from the low bounds and size whenever both become known, by adding the size to the low bounds.

A bounds check obtains the bounds information through pointer or object meta-data or from constants generated at compile time. As noted earlier in subsection II.D.2, McGary's pointers are actually fat-pointers with the bounds meta-data directly associated with it. Fat-pointers are generated either from dynamic memory allocation routines or from a address operator (e.g. &). The address operator uses size information (from type) and low bounds information to construct the fat-pointer. Completely static objects have its bounds information placed in constants, as they lack fat-pointers. As we will see later in section II.E sometimes it makes sense to force these bounds to either pointer or object meta-data.

McGary's compiler (hence ours) generates bounds checks whenever C pointer dereference operators (e.g. *ptr) or array subscripting operator (e.g. base[index]) are seen in the program. Depending on the compiler, some of these checks maybe found redundant, and eliminated by optimization. Direct-Association of meta-data compilers converts pointers to fat-pointer representation. These too maybe optimized back to regular pointers along with any bounds check. Type-optimizations are discussed in section II.F.

II.D.5 Performance Analysis

We now examine the performance of the different implementations of bounds checking described in Section II.A. The two-branch bounds check (*GM*)

implementation is used by the baseline McGary compiler [25]. It has a compare-branch-compare-branch sequence. The single branch (*1BR*) uses one (unsigned) subtract-compare-branch-trap sequence [5]. We compare this to a single instruction bounds check by using the bound instruction with our bound array optimization for the *bnd-array* results. Reported numbers are run-time overheads in comparison to the baseline without any bounds checking, by normalizing the bounds results against the baseline. All three code combinations are shown in Figure II.3.

Figure II.5 shows the percent slowdown for running the compiled programs with the different bounds implementations on an AMD Athlon. The first observation of our comparison is that bounds checking using McGary's (GM) compiler results in overheads of 71% on average. The second observation is that using the single branch compare implementation *1BR* reduces the average performance overhead from 72% down to 48% on the Athlon. The third observation is that the x86 bound instruction overhead (*bnd-array*) provides the lowest bounds checking overhead when compared to these two techniques with an average slowdown of 40% on Athlon. Ideal bounds *bnd-ideal* has an average 28% slowdown, by eliminating all check overhead, but not meta-data copying overhead. This demonstrates headroom to eliminate check execution overhead as demonstrated by the 12% difference, and the copying overhead is even more significant with 28% overhead remaining.

The difference in overhead comes from several sources. The most important component overhead comes from dynamic instruction count as seen in Figure II.6. As expected there is significant 149% instruction overhead for *GM*, which is reduced to 121% for *1BR*, and roughly half of that 65% for *bnd-array*. Of this 65% as reported by *bnd-ideal*, 78% are support memory instructions not directly part of the check, and the remaining 22% are *bounds* instructions. We

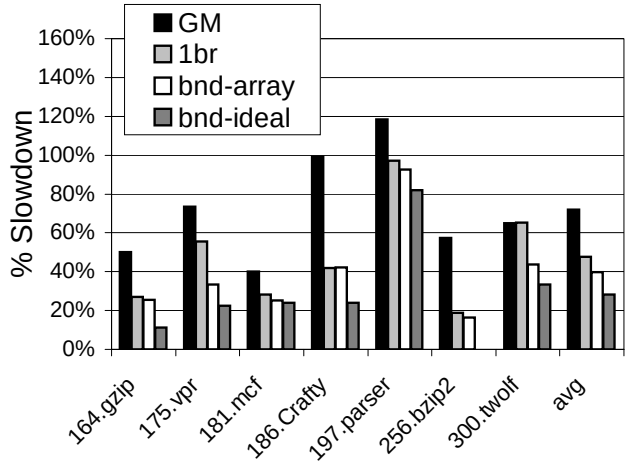

Figure II.5: Run-time overhead of bounds checking (AMD Athlon).

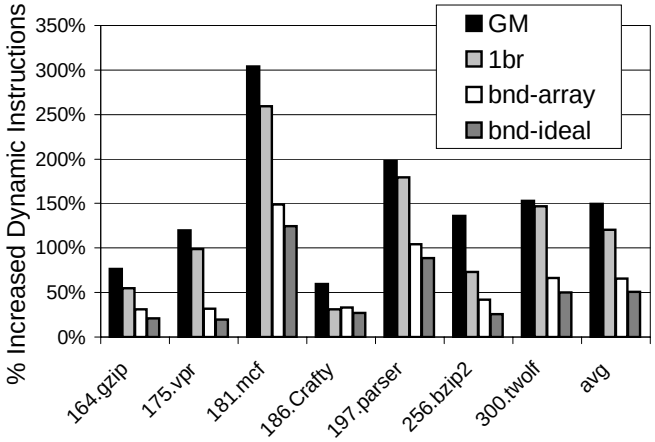

Figure II.6: Increased dynamic instructions due to bounds checking

also examined the branch misprediction and L1 data cache as provided in Figure II.7 and Figure II.8 respectively. The benefit of the bound instruction relative to the single or two branch bounds implementations can be partly attributed to the effect on branch misprediction rates on the Athlon. We can see in Figure II.7 that two-branch *GM* (256%) has much higher branch misprediction rate than *bnd-array* (11.5%) or the one branch version *1BR* (105%). The additional first level data cache miss effects due to additional meta-data and memory accesses is in Figure II.8, which is usually similar across different bounds checks. Array dominated programs fares equally for most programs (e.g. 300.twolf), but in one case is hurt by extra object meta data in *bnd-array* (e.g. 186.crafty). Pointer dominated programs like 197.parser that is sensitive to hit rate in L1 data-cache, suffered the most with additional meta-data induced cache capacity misses.

The *bnd-array* result demonstrates a significant improvement going from baseline 72% to 40%, by reducing dynamic instruction count and eliminating branch misprediction overheads. Looking over the performance data, we see that further improvements are contingent on either eliminating bounds checks or reducing copying overhead.

II.E Data Layout

In this section we focus on fat-pointer meta-data for C and C++ programs. We examine storing this meta-data either along with the pointer or with the object, having seen fat-pointers and manged object meta-data previously. Figure II.9(a) generalizes these two approaches as *Pointer Meta-Data (PMD) and* Object Meta-Data (OMD). For some checks, where to store the meta-data is an implementation option, whereas for other checks the information needs to be stored as either PMD or OMD. We use bounds checking and dangling pointer checks to demonstrate this.

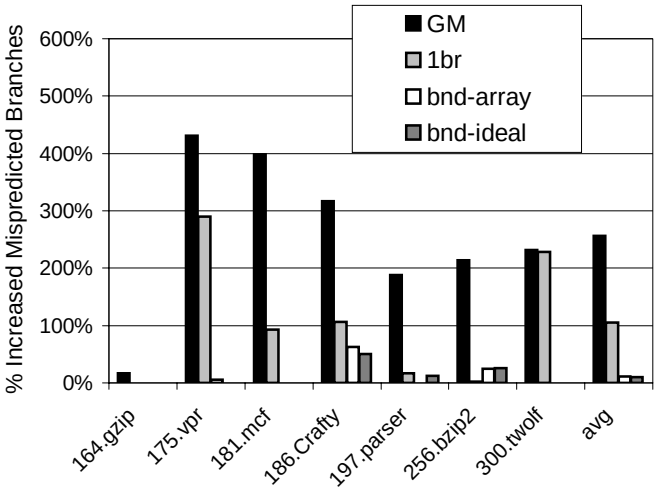

Figure II.7: Increased branch misprediction.

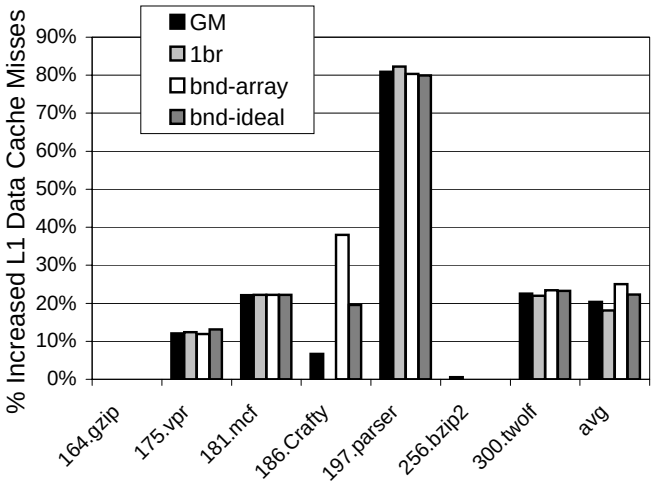

Figure II.8: Increased Level One Data cache misses

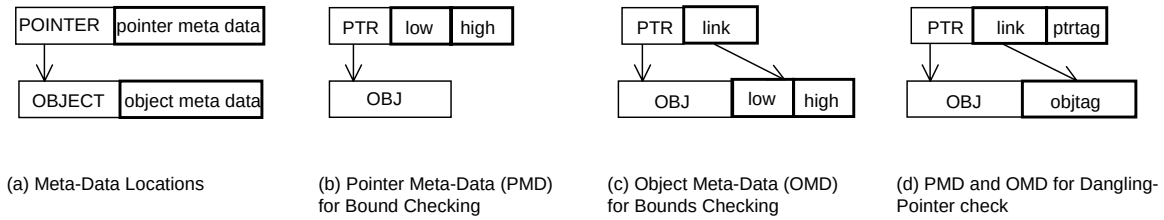

Figure II.9: Meta-data Representations. An arrow indicates a pointer to data associated with the object. Highlighted blocks are meta-data.

For bounds checking, the high and low bounds are typically stored adjacent to the pointer as PMD shown in Figure II.9(b) in the fat-pointer configuration. Alternatively, we propose that the meta-data for bounds checking could be stored with the referent object as OMD shown in Figure II.9(c). For this option, a *link* is stored adjacent to the pointer, which will provide the address to the location where the object meta-data is stored. The link handles interior and out-of-bounds pointer problem mentioned in subsection II.B.1. For dangling pointer checks, the meta-data is a pointer tag stored as PMD and an object tag stored as OMD. This is shown in Figure II.9(d). Just as with the OMD bounds checking, a link is required as part of the PMD to find the object tag stored as part of the object meta-data [5, 65].

Depending on where the meta-data is stored, as a PMD or OMD, the performance overhead will vary. This is because, the two representations will have different cache spatial locality. In this section we examine this trade-off, considering whether alternative meta-data storage is preferable to those currently proposed.

The differences between storing the bounds meta-data as OMD vs PMD are:

- Sharing of Meta-Data - Storing the meta-data with the object will allow the

meta-data to be shared across several pointers to the same object.

- Number of Pointers vs Number of Objects - Related to the above point is that some programs have many more pointers than objects. For example, programs like `mcf` and `parser` have N pointers for each object. For these programs, storing the bounds as PMD requires significantly more storage (and data cache usage) than storing them with the object. Storing meta-data with the object enables sharing them between pointers pointing to the same object.

- Reducing the PMD to 1 Word - Moving the meta-data to the object reduces the PMD from 2 words down to 1 word, and this is the link word to the object meta-data.

- Overhead of Extra Link Load - The OMD approach has the additional overhead of loading the link register. Note, that the link register overhead for the OMD case can actually be fairly small. This is because the link register can be hoisted to occur at the same time as the pointer load. If these both overlap, then the cost of the link load can be minimal.

II.E.1 Methodology

To better understand sources of delays in the processor pipeline, we modified SimpleScalar to classify every cycle in terms of generic delay sources. It also provides a larger data-cache configuration than current CPU's, suitable for modeling future designs. We used SimpleScalar 4.0 x86 Tool Set [12] for simulating our x86 binaries. The configuration is given in Table II.4 and based loosely on an AMD Athlon processor, as this represents a widely deployed modern desktop system, and a reasonable pipeline to emulate. If a delay prevents useful instruction execution for that cycle, then that cycle is categorized by that delay

type, otherwise that cycle is counted towards execution *ex*. A cycle is attributed to execution in this case, even if some other delay event is occurring, because the out-of-order pipeline is still doing useful work. Data-cache misses often stall data-dependent instructions, completely starving the pipeline, and are classified as *dc*. Because we want to know when data-cache misses occur, even though useful instructions are being executed, we classify cycles when this combination is occurring as *dc/ex*. Front-end pipeline starving events are caused by either branch misprediction *brm*, or by other front-end stalls such as instruction cache miss *fe*.

Table II.4: Simulation model based on the AMD Athlon.

Simulation Configuration	
Fetch Width	4 inst
Issue Width	4 inst
Func Units	4-ialu, 1-imult, 2-mem, 3fpalu, 1-fpmult
Reorder buf	RUU: 32, LSQ: 32
L1D	16KB, 2 way, 64B Block, 3 cycle latency
L1I	16KB, 2 way, 64B Block, 3 cycle latency
L2 Unified	2MB, 16 way, 64B Block, 20 cycle latency
DTLB	128 entry, 30 cycle miss penalty
ITLB	64, 30 cycle miss penalty
Memory	275 cycle latency
Branch Pred	16K meta chooser between gshare (8K entry) and bimodal table (8k entry); 16 Return Address Stack; 512 BTB; 10 cycle misprediction penalty

II.E.2 Meta-Data Overhead

To examine this trade-off, we ran experiments allocating different number of PMD and OMD words for all pointers and allocated objects in the Spec2000 benchmarks described in section II.D.2. At each pointer reference we make sure to

access the last meta-data word, but without performing any software check. For these results we broke the execution time into the percent of execution time (cycles) that was stalled due to microarchitectural effects described in the previous section. In Figure II.10, we compiled the programs so that there was 1 (1pmd), 2 (2pmd), 3 (3pmd) or 5 (5pmd) extra words associated with the pointer representing the effects of having PMD of that size. The additional overhead occurs from two sources with PMD. The first overhead comes from copying the meta-data. Every pointer assignment during execution has to also copy the pointer meta-data to the new pointer. The increase due to this can be seen in `twolf` as the number of execution cycles went up. The more dominant increase in overhead comes from the increase in data cache misses (dc) from the pointers with PMD. This effect is seen for the data cache sensitive benchmarks like (`mcf`, `parser` and `twolf`).

In Figure II.11, we experiment with varying OMD sizes. We store 2 (2omd), 3 (3omd), 6 (6omd), and 9 (9omd) extra words along with each allocated object. In addition, each pointer has 1 extra word, which provides the *link* from the pointer to the OMD as shown in Figure II.9(c). Irrespective of the size of OMD, the overhead has a fixed cost of copying just the link word on every pointer assignment as opposed to copying all the meta-data in the case of PMD. The size of the pointer is also a constant two words (one word for the pointer itself and another for the link). The graph shows a nearly flat trend even as larger object meta-data sizes are allocated.

II.E.3 Storing Meta-Data for Bounds and Dangling Pointer Checks

We now examine the overheads of implementing bounds checking and dangling pointer checks and show how these overheads differ based on the layout used for storing meta-data.

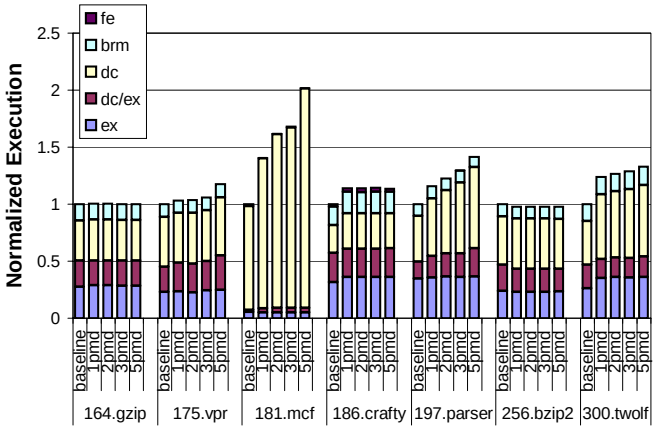

Figure II.10: Demonstrates the performance overhead for maintaining pointer meta-data and object meta-data of various sizes

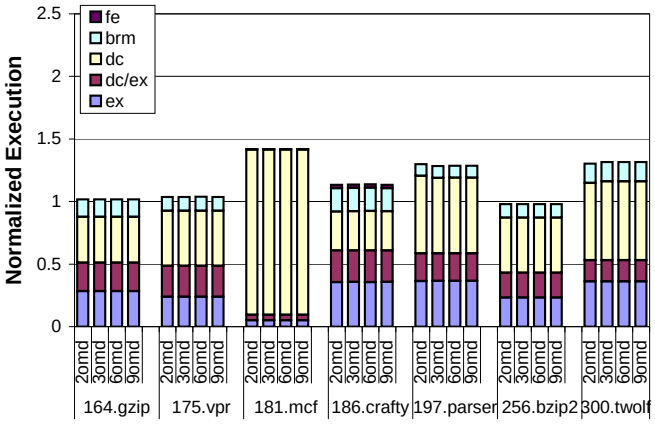

Figure II.11: Performance Overhead due to Object Meta-Data

```
bound ptr_reg, [base_reg+4]
```

(a) PMD x86 Bound Instruction

```
mov   [base_reg+4], link_reg
...
...
bound ptr_reg, [link_reg]
```

(b) OMD x86 Bound Instruction

```
mov [base_reg+4], link_reg
mov [link_reg], objtag_reg
mov [base_reg+8], ptrtag_reg
cmp objtag_reg, ptrtag_reg
jeq done
trap
```

(c) Dangling Pointer Check

Figure II.12: x86 implementation of the bound instruction storing the meta-data with the pointer (a), and storing the meta-data with the object (b). (c) shows the pseudo code for performing the dangling pointer check where the link register and pointer tag are stored as pointer meta-data and the object tag is stored as object meta-data.

Bounds Checking

Bounds checking uses the low and high boundary information associated with each memory object to determine if an out-of-bounds pointer reference has occurred. This is done for each source code pointer dereference or array reference, using the x86 `bound` instruction as shown in Figure II.12(a) and (b). The code example assumes that the pointer is stored in register `ptr_reg` and the base address for the two words stored the high and low bounds is the second parameter. Figure II.12(a) assumes the meta-data is stored as PMD as in Figure II.9(b). The other option would be to store the bounds as OMD as in Figure II.9(c), and Figure II.12(b) shows the code for this. In this case, the link pointer is loaded, and then passed to the `bound` instruction.

Dangling Pointer Checks

Dangling pointer check determines if a referenced object has been freed and potentially reallocated, but incorrectly accessed afterward with the old pointer.

```
--------------------------
bound ptr_reg, [base_reg+4]
--------------------------
load [base_reg+4], low_reg
cmplt_trap ptr_reg, low_reg
load [base_reg+8], high_reg
cmpgt_trap ptr_reg, high_reg
```

Figure II.13: The baseline micro-op expansion of the x86 Bound Instruction.

It does this by associating a tag with the pointer and a second tag with the object, with the property that they must match. At object creation, a unique tag id is assigned to both the pointer, and object tags. When the object is freed, the object tag field is cleared. A pointer dereference to the object performs a tag check. If they mismatch then the pointer must point to an object that's been either freed or reallocated. The x86 pseudo-code for implementing a dangling pointer check is shown in Figure II.12(c). The meta-data for the dangling pointer needs to be stored as in Figure II.9(d), where there is a link and pointer tag stored as pointer meta-data, and the object tag is stored as object meta-data.

II.E.4 Meta-Data Checking Overhead

When using bounds checking or dangling pointer checking, the checks occur at pointer dereferences, which can create large run-time overhead. Figure II.14 shows the overhead for using the bounds checking instruction in Figure II.12(a), where it is translated into the micro-op sequence in Figure II.13 when executed in the pipeline. The second bar in Figure II.14 shows the results for storing the bounds as PMD as in Figure II.12(a). The first bar shows the results for storing the bounds as OMD as in Figure II.12(b). The overhead of bounds checking is 81% on average when the bounds are stored in PMD but is 48.4% when the bounds are stored in OMD. The overhead comes from increased

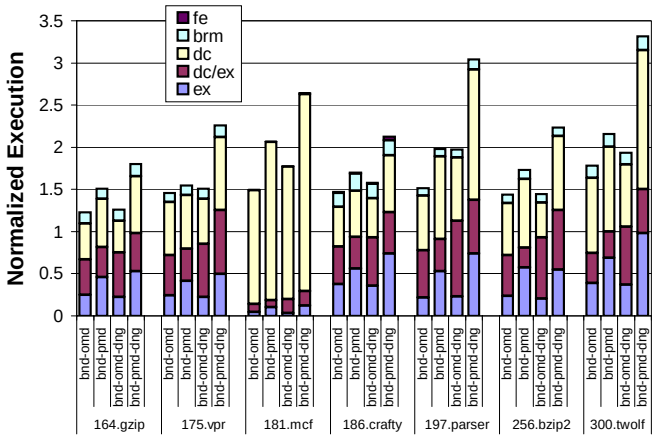

Figure II.14: Bounds and combined Dangling Pointer and Bounds check overhead

number of instructions from having to copy the pointer meta-data, the additional micro-ops to perform the check, and the increase number of cache misses.

As part of this study, we also want to examine the effect of performing multiple safety meta-checks on a pointer at the same time. In addition, looking farther into the future having multiple forms of meta-data stored with an object can potentially even aid hardware optimizations.

To examine the effect of performing multiple safety checks, we also provide results in Figure II.14 for performing both bounds checking *and* dangling pointer checks for pointers at the same time. This is equivalent to executing the code in Figure II.12(a) and (c) at the pointer dereference when the bounds are stored as PMD, or executing the code in Figure II.12(b) and (c) at the pointer dereference when the bounds are stored as OMD.

To perform the combined check for PMD, the pointer-meta data is now 4 words wide since it contains the high and low bounds, a link to the object meta-data, and the dangling pointer tag. Then the object meta-data contains just the dangling object tag. To perform the combined check for OMD, the pointer-meta data is only 2 words wide since it contains only a link to the object meta-data,

and the dangling pointer tag. Then the object meta-data contains 3 words, which includes the low and high bounds and the dangling object tag.

The fourth bar in Figure II.14 shows results for bounds plus dangling checks where the bounds information is associated with PMD. The third bar shows results for doing both the checks, but for these bounds information is associated with OMD. The performance overhead increases greatly due to the wider pointer-meta data as we saw in our earlier results in Figure II.10.

II.F Eliminating Unneeded Checks by Taint-Based Analysis

Bounds checking verifies all pointer and array accesses. While this provides complete protection against all kinds of buffer overflow exploits, it has significant performance overhead. In this section, we present a technique to limit the scope of bounds checking to only those objects that are vulnerable during a buffer overflow exploit. The goal of this optimization is to filter away those bounds checks that are not necessary to guarantee security against buffer overflow exploits. We focus on only bounds checking data that is passed to an application from the external interfaces.

II.F.1 Interface Analysis Example

Figure II.15 shows a contrived code example to illustrate what would be labeled as TAINTED when performing our analysis. In the example, the array `tainted_args` gets its value from the command line argument `argv` and the pointer `tainted_alias` gets its value as a result of pass by reference to the library call *gets*. Both `tainted_args` and `tainted_alias` are of type TAINTED and accesses to them will have to be bounds checked. The array `indirect` has an assignment from `tainted_alias` and hence accesses to `indirect` also need to be bounds

checked. In addition, `tainted_gets` will also be assigned to type TAINTED because it is an alias to the object referred by the pointer `tainted_alias` (line 5), and any references to `tainted_gets` later in the program would need to be bounds checked.

Note, even if all the accesses inside the library call `gets` are bounds checked, we still need to bounds check the arrays `tainted_gets` and `indirect` in the application code. This is because the data in the buffer populated by `gets` comes from the outside the program and could contain an exploit. That buffer can still be copied to other buffers inside the program, which might create a security problem if not bounds checked at the other buffer references. For example, the size of `indirect` array is less than the size of `tainted_gets`, which can cause a buffer overflow of `indirect` and expose a possible write-attack.

II.F.2 Interface Analysis Algorithm

Buffer overflow exploits are launched by an adversary by providing malicious input through the external interfaces to the application. External sources of malicious inputs to the program include, the return values from the library calls such as *gets*, and the command line argument *argv*. In order to protect against buffer overflow exploits, it should be sufficient to bounds check accesses to only those objects that get their values from the external input. We call such objects and their pointer aliases as TAINTED and all the other objects as SAFE. A TAINTED object can get assigned to external input either directly from the external interfaces or indirectly from another TAINTED object. Figure II.15 shows a simple code example to illustrate what would be labeled as TAINTED when performing our analysis, which we will go through in this section.

Limiting bounds checking to only TAINTED objects can decrease bounds checking overhead but at the same time provide a high level of security against

```
1. char* gets(char* s);
2. int main(int argc, char** argv)
3. {
4.   char tainted_args[128], tainted_gets[128],
          indirect[64]; // TAINTED and FAT
     // Pointer alias to object
5.   char *tainted_alias = tainted_gets;
     // source of malicious input: argv
6.   strcpy(tainted_args, argv[1]);
    // source of malicious input: return value of gets()
7.   gets(tainted_alias);
8.   for(i=0; (tainted_alias[i] != '\0') ; i++)
     {  // indirect is TAINTED because it 'uses'
        // tainted_alias
9       indirect[i] = tainted_alias[i];
     }
     // safe_array is SAFE and THIN
10.  char safe_array[128] = "I am not tainted";
     // foobar never passes safe_array to external
     // interfaces nor assigns it tainted data
11.  foobar(safe_array);
     ...
   }
```

Figure II.15: Accesses through the pointers and arrays named `tainted_args`, `tainted_gets`, `tainted_alias` and `indirect` need to be bounds checked. The array and pointer `tainted_args` and `tainted_alias` get their values directly from the external interfaces - `argv` and the library call `gets` respectively. Hence, they are of type TAINTED. The object `indirect`'s value is defined by a use of `tainted_alias` pointer and hence it is also of type TAINTED. All pointer aliases to TAINTED objects are fat pointers. Also, the fat pointer `tainted_alias` will propagate TAINTED to the array `tainted_gets` on line 5. Finally, `safe_array` is determined to be SAFE because it is passed to a function `foobar`, which does not pass `safe_array` to external interfaces and does not assign `safe_array` data from an external interface.

buffer overflow exploits. Reduction in the performance overhead can result from the following two factors. First, we can eliminate the bounds checks for accesses to SAFE objects. Hence, we will be able to reduce the number of instructions executed to perform bounds checking. Second, we do not have to maintain bounds information for all the pointers. This is because the pointers to the SAFE objects need to be only normal pointers instead of being fat pointers. We call the type of normal pointers as THIN and the type of fat pointers as FAT. Reducing the number of FAT pointers would lower the initialization overhead in creating them and copying overhead from passing those pointers as parameters to functions. More importantly, our optimization can reduce the memory footprint of the application and hence we can improve the cache performance.

Figure II.16 shows the steps used for the Interface Analysis algorithm. The goal of the Interface Analysis is to find all the objects and their aliases that should be classified as TAINTED in order to perform bounds checking on their dereferences. As described earlier, a TAINTED object is one that gets assigned with external data either directly from an external interface or from another TAINTED object.

Our algorithm processes the type properties using interprocedural data-flow, and points-to information: The data-flow graph models the assignments of scalar and pointer values. The points-to graph represents the relationship between the pointer and its referenced object. These graphs operate on arrays, scalars and pointers objects with other types reduced to these basic types. The TAINTED and SAFE properties apply to all objects, while FAT and THIN apply to only pointers.

The first step of the Interface Analysis is to construct the assignment data-flow graph, and to discover the initial points-to information from the pointer initialization. Address operators and dynamic memory allocator functions per-

Tainted Analysis

1. Construct the inter-procedural data-flow graph, and initial pointer aliasing information for points-to.
2. We forward propagate the points-to alias relationships [82, 3] though the data-flow pointer assignment network, generating additional aliases.
3. All objects (including pointers) are initialized to type SAFE. All pointers are also initialized to type THIN.
4. Apply the TAINTED type qualifier to the pointers and objects that are either (i) assigned to the return values of the external interface functions, (ii) are passed as reference to external interface functions, or (iii) get assigned to the command line parameter ARGV.
5. Using the data-flow graph propagate TAINTED forward along scalar dependencies and mark them as TAINTED.
6. Add bounds checking to all pointers and array dereferences that are marked as TAINTED.
7. All pointers that are bounds checked are assigned to be type FAT.
8. Backwards propagate FATNESS through the pointer assignment network.

Figure II.16: Algorithm for interface optimization

form this initialization, returning the reference of an object to a pointer. Next we propagate the pointer alias relationship, building up our points-to database. We describe properties of the points-to maintained at this step in the following section II.F.3. Third, we initialize all the pointer and object types to SAFE. The fourth step in our algorithm is to classify those pointers and objects that are assigned to the command line parameter `argv` and the return value of library calls as TAINTED. If this is a pointer, then the object referenced is TAINTED. Also, those objects whose value can be modified by library calls (pass by reference) are classified as TAINTED. In our example, in Figure II.15, the objects `tainted_args` and `tainted_gets` will be classified as TAINTED after this step. In step five, we propagate the TAINTED type information forwarded along the scalar data-flow graph dependencies, including values from array references. We assume that operations other than copy (e.g. arithmetic) will destroy the taintedness of the scalar assignment. In addition, we use the points-to analysis to mark any pointers that reference a TAINTED object as TAINTED. This step iterates until the TAINTED no longer propages. In doing this propagation, additional objects may be marked as TAINTED. After this propagation, the array `indirect` will get classified as TAINTED in our example code through forward propagation, and the array `tainted_gets` will be classified as TAINTED through points-to analysis. In step six, add bounds checks to all dereferences of pointers and arrays that are marked as TAINTED. In seven, all pointers that are bounds checked will be marked as FAT, and the rest will be marked as THIN. In step eight we backwards propagate FAT through the pointer assignment network to initialization, ensuring bounds information can reach the check.

II.F.3 Aliasing Properties

We use points-to analysis to determine which objects a pointer aliases [82, 3, 79] for two different applications. The first use is to allow pointers to determine if they reference a TAINTED object for which we use Andersen's analysis [3] to distinguish multiple aliased objects. From object aliasing we determine if the pointer references a TAINTED or SAFE object, consequently whether the pointer must be designated FAT or THIN. The second use fuses the multiple pointer-to-pointer aliases into a single class, as fusing simplifies how we use the alias information. This version of points-to helps us recognize nested pointers, and prevent conflicting pointer representations. Steensgaard [82] analysis does this fused points-to analysis for us. Consider as an example the type `char **`, which is a pointer to a char pointer `char *`. Variables that assign to or are assigned from the `char *` must have all of the same label, which is either THIN or FAT. TAINTED is similarly made consistent. Both points-to analysis use the pointer propagation data-flow to discover additional aliases.

Consider the following example of data-flow and alias analysis in Figure II.17 and II.18. We propagate TAINTED forward through scalars to correctly mark the array `x`, where getchar() is an external interface that may attempt to inject malicious code. Dataflow discovers pointer assignment `z=y;` meaning `z` shares `y` aliases, and points-to analysis would discover that pointers `y` and `z` aliases `x`. The pointer `y` and `z` become FAT, and remain FAT even if they reference a SAFE object.

II.F.4 Memory Writer Algorithm

In addition to the above *interface* optimization, we also perform another optimization which we call the *memory-writer* optimization. Buffer overflow exploits for remote execution of malicious code are launched by writing beyond the

```
char x[100]; int c, i=0;
char *y=x,*z;
while ((c = getchar()) != EOF) {
  x[i++]=c;
}
z=y;
```

Figure II.17: TAINTED flow code for Figure II.18

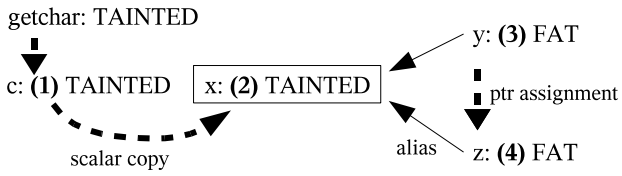

Figure II.18: TAINTED flow via scalar assignment and aliasing

boundaries of a buffer, which implies that we have to do bounds checking only for memory accesses that are writes in order to guard against memory corrupting exploits. Write attacks are the most common form of buffer overflow exploit, and probably the most serious due to the possibility of its use in injecting malicious code. We now summarize the memory writer algorithm when used by itself, and then when used with the interface optimization.

Mem-Write Only - The first step is to mark all writes to array and pointer dereferences as being bounds checked. These are left hand side of assignments through array and pointer dereferences. All of these pointers are marked as FAT. The next step is to find any pointer that will directly or indirectly (through data-flow) define these writes, so that they will also be marked as FAT. They need to be FAT, since they need to also maintain bounds meta-data information for the bounds check. For this we start from the FAT writes and propagate the type FAT backwards, along the edges of the data-flow graph through pointer assignments, to find all pointers that *define* the value of any FAT pointer.

Mem-Write with Interface Optimization - For this approach, we first per-

form the interface algorithm in Figure II.16. The only modification is step 6, where we only add bounds checking for arrays or pointers to buffers that are written as described above and marked as TAINTED.

II.F.5 Implementation Details

To build our optimizations, we need an implementation that can appropriately assign type qualifiers to all variables. The very first problem in this analysis is completely identifying all variables that we want to optimize, and to provide *a name to each pointer level* of the variable. By this, we mean that a variable can have multiple nesting of pointer types, and each level of nesting needs to be assigned the same type of FAT or THIN pointer. For example `char **` has two levels of references. To create a name for each of these variable nesting levels for our analysis, we start with the name of the variable. We then distinguish each pointer nesting level by pre-pending a label indicating the reference depth to the base name. A two level reference would have "p_p_", "p_", and "" (empty string) pre-pended for each level. Each of these levels can then be assigned a type of FAT or THIN. For this analysis, C structures are flattened such that each individual field is treated as a distinct variable, and the fields name are concatenated with the structure name making it unique. Overall, this naming scheme is similar to the one used by Austin et al. [5].

We build a graph representing assignments of data-flow information, derived from program assignment statements, address operators, function parameters and return values. After building our data-flow graph on a per function level, we merge the graphs to create a global inter-procedural graph. Our type-based data-flow analysis is path-insensitive; a qualifier that is computed for a variable holds throughout the entire program. Assignments that cause a pointer to become FAT affect the representation seen by the entire program, not just the

path it was assigned along. Similarly, type information passed through procedure calls to other functions must be consistent across all the call sites as we permit only one representation (no specialization) of that parameter inside of the function. For example, if a pointer parameter becomes FAT at one call site, then that same parameter will be labeled as FAT for all other call sites to that function. In other words, our inter-procedural analysis is context insensitive. Both path and context insensitivity greatly simplify the analysis. In addition, indirect function calls are also treated conservatively, where all possible function definitions that might match an indirect call site will have their parameters assigned with the same FAT or THIN label.

II.F.6 Network External Interface Results

Our analysis and the pruning of bounds checking can be applied to all external interfaces to an application, which would include disk, keyboard, mouse, cross-process communication, network traffic, etc. Or it could be applied to just a subset of these interfaces.

The current systems based upon dynamic taint analysis only focus on tainting network traffic [84, 21, 62], since this is how a worm attack occurs. In doing this, our approach will only need to bounds check buffers that are passed to the network system calls, and any of the data in the program that is tainted by it with the above analysis.

To analyze this effect, we performed our bounds checking analysis on the benchmark `ATPhttpd-0.4b`, which is a small web server, with a buffer overflow vulnerability [62]. In applying our external interface only guarding against write attacks as described above, we achieve a 6% slowdown over no bounds checking. We also verified that the vulnerability was caught using our taint-based bounds checking approach.

II.F.7 All External Interface Results

Since the SPEC integer benchmark suite does not have any network traffic, the amount of bounds checking is zero, which is not that interesting of a result to analyze. Therefore, we also examined applying our interface optimization for all system call interfaces to the program. For the SPEC benchmarks, we are bounds checking data from the operating system interface, command line ARGV parameter and anything tainted by it as described with the above analysis.

We will analyze the advantages of our two optimizations, *interface* and *memory-writer*, in this subsection. The binaries that we use for this analysis are generated using the x86 `bound` check instruction, with the code generation features given with *bnd-array* binaries as described in subsection II.D.3. We conducted this experiment on our AMD Athlon processor using the methodogy from subsection II.D.2.

Figure II.19 shows the performance advantage of our optimizations. The result labeled as *bnd-interface-only* corresponds to implementing only the interface optimization, the result corresponding to the label *bnd-mem-write* refers to our memory-writer only optimization, and *bnd-interface+mem-write* stands for the implementation where we applied both of the optimizations.

When each of the two optimizations are applied individually, the interface optimization reduces the overhead to 29%, whereas the memory-writer optimization reduces the overhead down to 28%. When both of the optimizations are applied together we find that the average overhead is reduced to 24%, which is a significant reduction when compared to our previous best result of 40% that we achieved using the *bnd_array* implementation. The *bnd-interface-only* represents performing bounds checks and maintaining fat pointers for all tainted data coming from external interfaces. This provides protection against both write and read buffer overflow exploits. Since write buffer overflow exploits

are the most harmful, *bnd-interface+mem-write* provides protection for all writes that write data from external interfaces.

Both memory-writer and interface contributes some of portion of the overhead reduction with an overhead of 29% (memory-writer) and 28% (interface). 181.mcf benefits significantly from interface-only as its pointer intensive data-structure was filled with internally generated data. These data-structures do not need to be bounds checked, nor turned into fat-pointers. Interface optimization converts these pointers back to regular pointers in the mcf data-structure as seen in 181.mcf in Figure II.21 where the heap overhead reduction from the baseline is dramatic. Contrast this with the smaller gains writer-only has on 256.bzip2 for example, which fills its compression matching array from external data, hence does not benefit from interface, and spends most of its time reading the array. Consequently, the execution time of 256.bzip2 benefits from bounds check elimination as seen in Figure II.20. All other benchmarks have a similar bounds check reduction from writer-only. From the figures we can see that both interface and writer optimizations can be combined, obtaining an overhead of 24%.

To analyze the main source of reduction in the performance overhead, in Figure II.20 we show the number of *static* bounds check instructions that remain in the binary after applying our optimizations. We can see that the *bnd-array* implementation, where we bounds check all of the memory accesses through pointers, contains 2203 x86 bound instructions on average. Our interface optimization which eliminates the bounds checks to the SAFE objects is able to remove 660 bounds checks from the binary to 1573 on average. The memory-writer optimization eliminates the bounds check to all the load memory operations. Hence, it significantly reduces the number of checks to 581 on average. When both the optimizations are combined together there are about 495 bounds checks left in

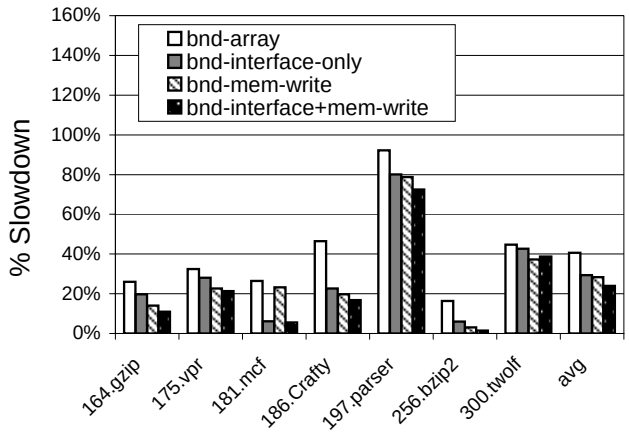

Figure II.19: Performance advantage of interface and memory-writer optimizations.

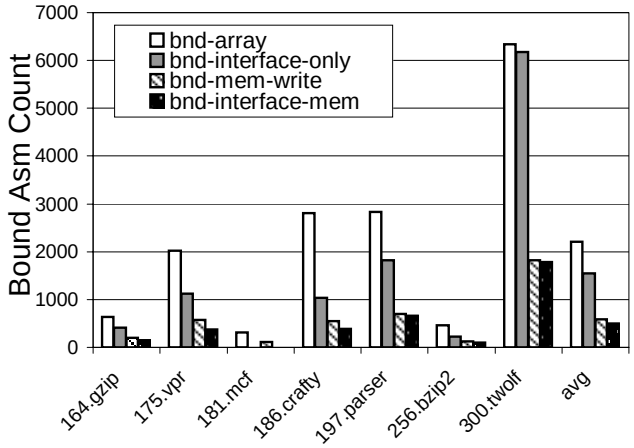

Figure II.20: Reduction in the number of static bound instruction in the binary.

the binary on average.

The performance savings shown in Figure II.19 are proportional to the number of bounds checks that we managed to eliminate. We would like to highlight the result for our pointer intensive application `mcf`. For `mcf`, we see significant performance reduction for interface only optimization. The reason for this is that the there was a decent size reduction in the heap memory used as a result of our interface optimization as it managed to classify 50% of the pointers as THIN pointers.

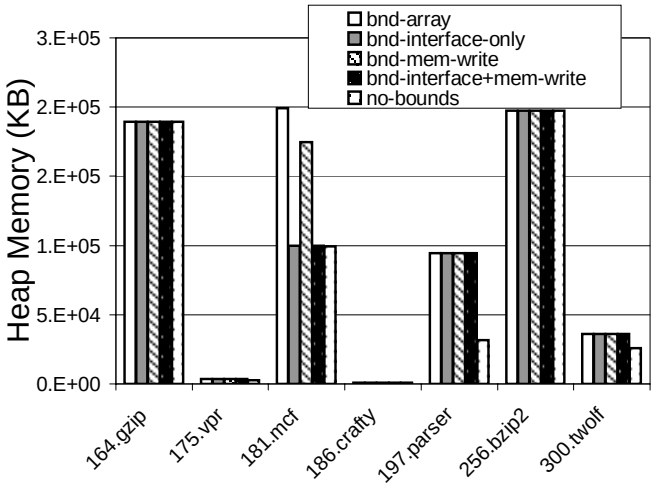

Figure II.21: Comparison of heap size overhead in KB

II.G Hardware Acceleration

In this section we examine the *Meta-Data Checking* (MDC) architecture hardware extensions to reduce the overhead of meta-data checks. The architecture extensions include, extending the x86 ISA with a new instruction, called the *meta-check* instruction and the necessary hardware support to implement and use it.

II.G.1 Motivation for Meta Checker Instruction

We make the following observations that identify the sources of overheads for doing meta-data checks for our two example applications - bounds checks and dangling pointer checks. The special meta-check instruction that will be explained later is designed to target these observations.

- Extra Instructions in Binary to Perform Check - As shown in Figure II.12 the dangling pointer check executes about five x86 instructions for each check (around 7 micro-ops expanding out the address generation). This can add pressure to the execution window and fetch bandwidth, which can adversely

affect the performance. A generic meta-data instruction could be used to concisely represent this check.

- Flexible Meta Data Representation and Efficient Cache Usage - As we noted in the prior section, how the meta-data is laid out and how it is associated with PMD or OMD can affect the data cache miss rate. So having the flexibility to associate the meta-data as either PMD or OMD (wherever appropriate) would be important for adding customized instructions for performing safety checks.

II.G.2 Overview of Meta Data Check Architecture Extensions

We propose extending the ISA with a special instruction called the *meta-check* instruction to perform the memory safety verification. A sequence of meta-check instructions perform the bounds and/or dangling pointer check functionality. One can view each meta-check as an assumption or rule that a pointer value must obey. Each meta-check instruction specifies the sources for meta-data, which can be either from PMD or OMD meta-data or a pointer register. It also specifies the check to be performed on that meta-data (e.g.: less than comparison, greater than comparison, equal to comparison, etc.). The meta-check instruction is associated with a specific virtual register, and whenever that register is used during execution, the meta-check will be performed.

The meta-check instructions program a *Meta-Data Check Table*(MDCT), which is a finite sized buffer to hold all of the information needed to perform the meta-data checks associated with a given architecture register. A meta-check instruction is assigned an entry in the MDCT. Once updated with a sequence of meta-data checks, the MDCT determines during renaming for all register uses if there is a corresponding check for that register. If so, micro-op instructions are inserted to automatically load the meta-data and to perform the check.

The meta-checks are bound to a triggering register, and are decoded when a memory operation uses it. This register is any of the general purpose registers, and contains a pointer value dereferenced by a load or store memory operation. Then when triggered, the MDCT sequence is read out, the corresponding micro-ops for the checks are inserted before the triggering memory operation. The micro-op expansion for the meta-check instruction could be provided from mechanisms such as DISE [17]. By generating the micro-ops to perform the safety check when the pointer register is used, we avoid the need to explicitly insert those checks in the binary. The format and the implementation of meta-check instructions are flexible enough to support different types of meta-data layouts, as well as potentially different uses.

II.G.3 Meta-Check Instruction

Meta-check instructions specify an instruction template filled in when the check sequence is bound to a pointer target register. The following is the format of the meta-check instruction we modeled:

```
meta-check  ptr_reg, slot, offset(ptr_base),  meta-operand-1, meta-operand-2, cond
```

The definition of the fields for the instruction are itemized below:

- `ptr_reg` - is the virtual register that contains the pointer value that the compiler wants to monitor. The pointer would have been loaded before executing the meta-check instruction.
- `slot` - To bound the size of the MDCT, we must limit each virtual register to at most N meta-check instructions. For this study, we use a limit of 4. Therefore, the slot bits represent which of the N meta-check instructions is being defined for the specified ptr_reg.

- `offset(ptr_base)` - ptr_base is the register used to load the virtual ptr_reg. An offset from this ptr_base address, is where we find the PMD, and from the PMD we can get access to the OMD as described in the previous section II.E. This ptr_base plus offset is saved to `MD_base`.
- `meta-operand-1 and operand-2` - Source operands of the of check operation, are summarized as the following four options:

  ```
  O(OMD_Mask)|P(PMD_Mask)|ptr|const
  ```

  These formats are either:

  - `N-bit PMD_Mask` - this indicates which pointer meta-data word(s) should be used in this meta-data check. This could be implemented as an offset instead of a Mask.
  - `N-bit OMD_Mask` - this indicates which object meta-data word(s) should be used in this meta-data check. This could be implemented as an offset instead of a Mask.
  - ptr - Pointer register that triggered the check (same reg as ptr_reg).
  - const - A small N-bit constant. Unspecified more significant bits are treated as zeros.

- `cond` - this determines the type of check to perform using the meta-data. The supported traditional types of checks could be: EQ, NEQ, GT, GTE, LT, and LTE.

The meta-check instruction allows the comparison of two items using the condition specified. The two items could be both meta-data, or the comparison could be between one meta-data and the value in `ptr_reg`. They also could be both from the PMD or both from the OMD. The order in which the expression

is evaluated is from left to right in terms of the `ptr_reg` and meta-data words specified in the PMD and OMD being compared.

When there is the load (definition) of the pointer register `ptr_reg`, this load indicates to the hardware to (a) save the meta-data base value `MD_base`, and (b) associate with the `ptr_reg` register with the specified check operation. (a) means we allocate a physical register to save the meta-data `MD_base` pointer computed from the address containing pointer plus a fixed one pointer-word offset- (`offset(ptr_base)`). This `MD_base` pointer contains the address of the 1st word of the PMD meta-data and if OMD is present that address is instead the link pointer to the OMD. From base pointer we obtain all PMD addresses by adding the PMD_mask offset, and from link pointer we obtain all OMD by adding OMD_mask offset. The masks mark use of a single word within the vector of PMD or OMD meta-data corresponding to a single set bit within the mask. By mask offset, we mean the address offset to access that word. Also at this load, we bind the register containing the pointer to the checks. Thus, whenever `ptr_reg` register is used by a load or store memory operation, the corresponding checks will be inserted into the pipeline. The hardware support to enable this is described in more detail below.

The reason for going with the above fairly generic meta-check instruction description is to not make an assumption about where data is located in the PMD and OMD for the type of checks that might want to be performed. The only assumption is that when there is a link, the first word of the PMD is the link to the OMD.

II.G.4 Using the Meta-Check Instruction

To better understand the meta-check instruction, lets look at using it to perform bounds checking and dangling pointer checks. In the example in

```
-------------------------------------------------------------------------------------
(1) meta-check  ptr_reg, 00, off(ptr_base),  P(0100), O(1000), NEQ  // Dangling Pointer Check
-------------------------------------------------------------------------------------
(2) meta-check  ptr_reg, 00, off(ptr_base),  P(1000), ptr, GT   // PMD Bounds Check Lower Bounds
    meta-check  ptr_reg, 01, off(ptr_base),  P(0100), ptr, LT   // PMD Bounds Check Upper Bounds
-------------------------------------------------------------------------------------
(3) meta-check  ptr_reg, 00, off(ptr_base),  O(1000), ptr, GT   // OMD Bounds Check Lower Bounds
    meta-check  ptr_reg, 01, off(ptr_base),  O(0100), ptr, LT   // OMD Bounds Check Upper Bounds
-------------------------------------------------------------------------------------
(4) meta-check  ptr_reg, 00, off(ptr_base),  O(1000), ptr, GT   // OMD Bounds Check Lower Bounds
    meta-check  ptr_reg, 01, off(ptr_base),  O(0100), ptr, LT   // OMD Bounds Check Upper Bounds
    meta-check  ptr_reg, 10, off(ptr_base),  O(0010), P(0100), NEQ  // Dangling Pointer Check
-------------------------------------------------------------------------------------
```

Figure II.22: Example meta-check instructions for dangling pointer and bounds checking. `P` stands for meta-check data being from the PMD, and `O` stands for it being from the OMD.

Figure II.22, (1) corresponds to the dangling pointer check in Figure II.9(d), (2) corresponds to the PMD bounds checking in Figure II.9(b), (3) corresponds to the OMD layout of bounds checking in Figure II.9(c), and (4) corresponds to performing both OMD bounds checking and the dangling pointer check on the same pointer. In this last case, the object tag is the third word of the object meta-data.

In Figure II.22(1), the first meta-check instruction is for specifying a dangling pointer check. As explained in Section II.E.3, to perform a dangling pointer check, the tag stored in the pointer PMD is compared against the tag stored in the OMD. Because it uses OMD, the first word after the pointer (in the PMD) is the pointer to the OMD. The second word in the PMD is the pointer tag and the first word in OMD is the object tag. These are specified by the bit masks `PMD_mask` and `OMD_mask`. These source operands for the NEQ check operation will cause a trap if they are not equal. Note, the example shows just 4-bits for the masks, but the masks can be longer based on how many bits are available in the instruction encoding. In addition, to allow access to larger meta-data structures, an offset into the meta-data could be used instead of a mask.

Figure II.22(2) shows using the meta-check instructions for bounds checking using the layout where the bounds information is stored in the PMD, as shown in the Figure II.9(b). One check instruction is for comparing the `ptr_reg` address, when it is used in a later instruction, with the lower bound stored in the first word of PMD and the other one compares the `ptr_reg` address with the higher bound stored in the second word of PMD. As bounds check uses multiple meta-check instructions, we enumerate and order the different instructions by the slot number.

After the meta-check is registered for a given `ptr_reg`, any instruction that uses that register (before it is redefined) for an address calculation has the corresponding checks inserted into the instruction stream. The architecture to support this is described later. The checks are inserted directly after the address generation and before any remaining operations for that instruction.

To give an example of how the checks are inserted automatically into the instruction stream, assume we insert the bounds checks in Figure II.22 (2) into the binary after a load of a pointer to virtual register `r1`. Then before `r1` is redefined, we see a use of it in the instruction `sub offset(r1), immediate`. Below is the micro-op sequence generated for the x86 subtract instruction along with the two meta-checks for bounds checking and their meta-data access loads that are inserted right after the address generation.

```
// Original x86 instruction
sub offset(r1), immediate

// micro-op expended of subtract
1.  agen  tmpAddrReg = r1 + offset    // address generation
2.  agen  lowaddr  = P(1000)+tmpLink  // meta-check uOp insertion - generate low bound address
3.  load  low  = M[lowaddr]           // meta-check uOp insertion - low bound loaded from PMD base
4.  cmp_gt_trap low, tmpAddrReg       // meta-check uOp insertion - compare the low bound
5.  agen  highaddr = P(0100)+tmpLink  // meta-check uOp insertion - generate high bound address
6.  load  high = M[highaddr]          // meta-check uOp insertion - high bound loaded from PMD base
7.  cmp_lt_trap high, tmpAddrReg      // meta-check uOp insertion - compare the high bound
8.  load  tmpReg = M[tmpAddrReg]      // load the current value
9.  sub   tmpReg = temReg - immediate // perform the subtract
10. store M[tmpAddrReg] = tmpReg      // store the result back in the address
```

One advantage of doing the above, is that if a trap occurs, it will be

caught before the store commits and the PC that will be set to *sub* instruction to re-execute it if necessary. This allows an exception handler or debugger to know exactly the instruction that violated the safety check. In comparison, when a `bound` instruction is used, the PC of the bound instruction would be marked as having the exception.

II.G.5 Hardware support for Meta-Check Instruction

Meta-check instructions are buffered in the Meta-Data Check Table as noted above. Currently we specify its capacity to be four entries for each register capable of referencing a pointer, and for x86 this is eight registers times four for thirty-two entries. When a meta-check instruction is decoded, it is placed into the MDCT table and it is assigns a physical register to hold the base pointer to the PMD meta-data. The base pointer is stored in a second table- the Meta-Data Register Map table. This has an entry for each general purpose register, meaning eight map entries. Both the MDCT and MDRM can be directly written and read from to enable context switching.

On executing a memory operation the MDCT table is consulted to determine if a check sequence is bound to its pointer-register. If so, check instructions corresponding to that pointer-register are micro-op expanded and issued forth to register renaming from decode. The MDCT table, shown in the Table II.5, contains the following fields. The first field holds the virtual register that will hold the pointer we want to check, the second is the slot identifier, the next two fields hold the first and the second meta-check operand bits, and the last field holds the condition to evaluate the check expression. The table is direct-mapped indexed first by the virtual register and then by the slot number. Similar to the register rename map, the MDCT keeps track of only the most recent definition for each virtual register, and must be restored on a branch misprediction.

Table II.5: Meta-Data Check Table (MDCT)

Pointer Virtual reg	Slot	1st Operand reg	2nd Operand reg	Operation
r1	0	O(1000)	ptr	GT
r1	1	O(0100)	ptr	LT
r1	2	O(0010)	P(0100)	NEQ
r1	3			

We also maintain a mapping between each possible general purpose register containing pointer to be used in a meta-data check and a physical register containing a pointer to the meta-data, as described in Table II.6. If this field is empty then no meta-check has been assigned.

Table II.6: Meta-Data Register Map (MDRM)

Pointer Virtual reg	Meta_data physical MD_base
r1	p20
r2	p2
r3	-

We now describe what happens when a meta-check is fetched, and when the pointer register we are watching is used for an address generation.

Expanding a Meta-Check Sequence- Take for example the three meta-check instructions in Figure II.22(4). After executing those three meta-check instructions, the state of the MDCT table will be as shown in Table II.5 and MDRM table in Table II.6. The virtual register $r1$ is the pointer register to be checked (the register into which the pointer would have been loaded). On executing the first meta-check instruction, a physical register $p20$ is allocated to

address of the start of the PMD. This is the `MD_base`, and it is shared among meta-checks for the same virtual register definition.

Then the micro-code engine automatically inserts into the instruction stream instructions to perform the check comparisons in the table. First we need to obtain the meta-data. We can derive from `MD_base` the link pointer, which for a given sequence, is generated only once at the beginning of the sequence. The micro-op expansion accomplishes this by checking a sequence of meta-check instructions for OMD operands, allocating a physical register to hold the link value, and issuing out the load. Subsequent instructions may use the link register. In this example it allocates $p8$ as the link register. Next expansion generates the load operations for the meta-data, using as the base register: for PMD meta-data the base is the `MD_base` register $p20$, and for OMD meta-data the base is the link register $p8$. An `agen` address generation instruction then sums up the pointer and the mask offsets into a temporary register. The temporary is consumed by the load to obtain the value of the meta-data, which are then used by check comparison instructions.

For the above example, it would insert the following four address-generation and four loads, three compare and trap instructions that would perform bounds and dangling pointer check.

```
load  p8  =    [p20]
agen  p25 =    O(1010) + p8
load  p2  =    [p25]
cmp_gt_trap    p2,  p10
agen  p26 =    O(0100) + p8
load  p4  =    [p26]
cmp_lt_trap    p4,  p10
agen  p27 =    O(0010) + p8
load  p8  =    [p27]
agen  p28 =    P(0100) + p20
load  p5  =    [p28]
```

```
cmp_neq_trap   p16, p5
```

As described earlier for the store example, these checks will be inserted in the instruction stream between the address generation and the rest of the instruction's execution that is being checked.

Freeing MDCT and MDRM Table Entries and their Physical Registers - When a virtual register is redefined by an instruction, the MDCT and MDRM entries corresponding to that register are removed, since the virtual register has been redefined. However, the physical base register allocated to those entries is not freed until the instruction that is redefining the virtual register commits. When a new register definition occurs, if there are hits in the MDCT, we (1) remove the entries from the MDCT, and (2) remove the base pointer register mapping from MDRM. When this new instruction commits, we know that we can then free the base pointer physical register. This is similar to the conventional algorithm used to manage freeing physical registers in current architectures.

Even though multiple definitions of a virtual register can be alive at a time, the MDCT and MDRM table needs to only hold the check instructions and base meta-data mapping corresponding to the latest definitions of the virtual registers. This is because decoding and renaming are done in-order, and the tables are used to just generate the micro operations in-order during the decode stage.

Branch Mispredictions, Context Switches and Exceptions - Branch mispeculations are handled in modern architectures by checkpointing the register rename table. To support our extensions, the physical register mapping of the MDCT is checkpointed as with any other renamed register set. Upon recovering from a misprediction, the check-point map is restored.

Context imposes additional burdens, as the MDCT and MDRM state must be saved to software memory (kernel stack). We want to narrowly expose the MDCT architecture to enable efficient saving and restoring of MDCT state but no more. MDCT saved state are the original meta-check opcode encoding. When we store an entry from the MDCT to memory, it recovers the original meta-check representation, which is stored in the MDCT. We also save and restore the value of the base pointer register from the MDRM, as each general purpose register maps to a meta-checked pointer physical-register that containing the address of the base of the PMD meta-data. Upon restoring the meta-check instruction along with the base value, we re-execute the meta-check to regenerate the MDCT state. As there are up to 32 meta-check instruction entries and eight base pointer entries, the context switcher checks if the register is used for meta-check instructions, spilling them only if necessary. We keep track of the use status in a bit vector indexed by virtual register number. Upon restore we walk through the bit vector, and reload the corresponding previously used MDCT table entries and the base register.

Because meta-check instructions only raise exceptions as a side effect, they maybe re-executed without harm. This simplifies exception handling as temporary state generated during micro-code expansion does not need be saved. Upon exception caused by the checks or by some external event, the PC is placed on the the triggering instruction allowing control flow to returns back after handling exception. The ability to re-execute is useful if we want to tolerate check violation by repairing incorrect pointer values as proposed by Rinard et al. [74]. A violation raises an exception before any state is modified, intercepted by a handler, which then nullifies or safes the load so it does not fail a second time, then re-executes.

Impact on Physical Registers - Supporting the meta-check instructions has an impact on the required number of physical registers for the machine. Each MDCT entry, in the worst case needs a physical register. We assumed for our implementation that the number of physical registers is 64.

II.G.6 Performance Result

In this section we will discuss the benefit of Meta-Data Checking (MDC) architecture. First we will discuss the results for doing just the bounds checking and then discuss results for doing both bounds and dangling pointer checks.

Performance of Bounds Checking In the subsection II.E.4 we discussed the overheads of bounds checking implementations. There we did not assume any architectural support but instead implemented bounds checking using existing x86 assembly instructions. Those results are shown again in the Figure II.23. The result labeled as *bnd-pmd* shows the overhead of bounds checking using PMD layout (shown in the Figure II.9(b)) and the one labeled as *bnd-omd* shows the bounds checking overhead when we use OMD layout (shown in the Figure II.9(c)). In addition, Figure II.23 shows the overhead of bounds checking when we implement it using the meta-check instruction described earlier. This result is labeled as *bnd-omd-MDC*. For all the results we again break the execution time between fetch stall (fe), branch misprediction (brm), data cache misses (dc), overlapped data cache miss with execution (dc/ex), and execution (ex) where there were no stalls as described in subsection II.E.1.

We see that the average overhead is 81% when the bounds are stored with the PMD but we incur only 48% overhead when the bounds are stored with the OMD. This improvement can be attributed to the improvement in cache miss rates as we now share the bounds information for an object across the all the pointers to that object.

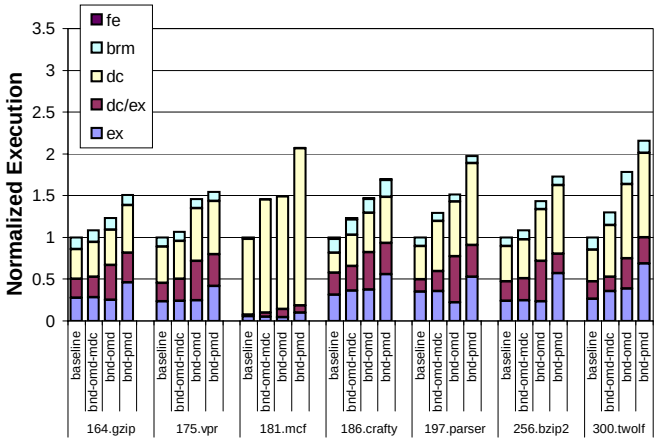

Figure II.23: Performance Overhead of Bounds Checking using meta-check instruction and MDC hardware

Using the MDC architecture the average overhead of bounds checking is reduced significantly to 21%. These savings can be attributed to the reduction in time spent in execution (represented by ex and dc/ex in the Figure). Time spent due to `ex` and `dc/ex` is consistently reduced across all the benchmarks. Especially for programs like `bzip`, the performance improvement is significantly reduced from 43.7% to 8.3%.

For programs like `mcf`, we do not see appreciable gains. The reason is that `mcf` is memory bounded and a greater proportion of the execution time is spent servicing cache misses. The MDC architecture, though it optimizes the number of instructions fetched and executed, the overhead due to increased memory footprint to store the meta-data information still remains. But, note that the stalls due to data cache misses is significantly reduced in OMD layout (*bnd-omd*) as compared to PMD layout (*bnd-pmd*).

To summarize, our meta-data layout coupled with meta-check instruction reduce the average overhead of bounds checking to 21% slowdown which is a significant reduction when compared to 81% incurred by current software implementations when providing complete bounds checking.

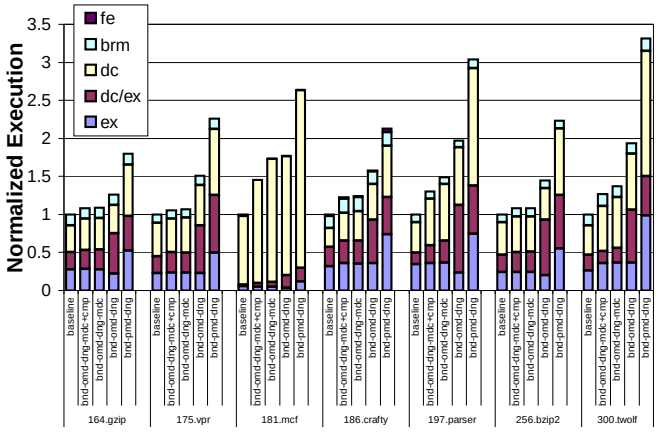

Figure II.24: Performance Overhead of Dangling-Pointer Checking using meta-check instruction and MDC hardware

Performance of Dangling Pointer Check Figure II.24 shows the overhead for performing dangling pointer checks on top of bounds checks. The two results from Figure II.11, *bnd-pmd-dng* and *bnd-omd-dng* are reproduced here for comparison. The bounds check and the dangling pointer check are implemented for these two results using only x86 instructions.

Before discussing the results, here is a quick summary on how the meta-data is laid out. For *bnd-pmd-dng*, the bounds information is associated with PMD. There will be four PMD words: two for bounds, one for link address and another for pointer tag needed for the dangling check. In addition, there will be one OMD word to hold the object tag needed for dangling pointer check. For the *bnd-omd-dng* results, bounds information is associated with OMD, which means there will be just two PMD words (one for link address and another for pointer tag) but three OMD words (high, low bounds and one more word for object tag).

The overhead of these implementations are pretty steep. The overhead for *bnd-pmd-dng* configuration is 148%, which is what we expected as it uses four PMD words. Since the dangling pointer check needs a link address it is definitely better to store bounds in OMD. In doing this, we see a significant reduction in

the average overhead to 63.9% (corresponding to *bnd-omd-dng*).

The *bnd-omd-dng-MDC* result in the Figure II.10(b) corresponds to the implementation that assumes the MDC architecture. The average overhead reduces to 29.8% from 63.9% when we apply MDC architecture optimizations. We achieve this reduction in performance overhead by reducing the number of instructions inserted into the binary to perform the check. This can be noted by comparing the reduction in `ex` and `dc/ex` components.

Finally, we can compress the link address and the pointer tag into one PMD word. The result corresponding to this optimization is labeled as *bnd-omd-dng-MDC+Comp*. This compression reduces the increase in memory footprint and as a result yields better cache performance. On average, the overhead reduces to 21.2%, which is only a slight increase in overhead for adding dangling pointer checks on top of bounds checking. This shows that our approach scales well and that as long as we can avoid increasing the PMD size we can keep the performance degradation within tolerable limits.

II.H Summary

Automatic run-time pointer checking can detect memory bugs, provide security, and help software developers isolate and find memory bugs efficiently. As programs get ever larger, and the cost of bugs in dollars and security adversaries becomes painfully expensive, these techniques become increasingly important.

Computer architecture needs to play a role in lowering the overhead of these software checks. Bounds checks protect against over half of the CERT exploits, providing reasonable coverage with reasonable run-time cost. We first considered optimizating the code sequence of software bounds checks to use the x86 bounds, having realized that dynamic instruction count and branch misprediction accounts for much of the execution overhead. This lowers overhead on

two-branch bounds check from 70% to 40% using *bnd-array*, when measured on an AMD Athlon processor. Next we propose a check redundancy elimination technique to eliminate unneeded software checks when protecting against security exploits. By focussing checks only on memory buffers that write external data, we can further reduce this run-time overhead to 24%.

Next, we provided a detailed analysis of the trade-offs for where to store the meta-data, with the pointer or with the object. The results show that storing the meta-data with the object instead of the pointer provides better results, especially for programs like `mcf` and `parser` where there are many more pointers stored in memory than objects (each object has several pointers). In addition, as many more different checks are done on a pointer, storing the required meta-data with the object scales better in terms of performance. To obtain further coverage, the meta-data checks we examine in this analysis are dangling pointer and bounds software checks.

Incorporating both bounds and dangling pointer checks using OMD data layout approach results in an average simulated slowdown of 63.9%. This slowdown is still too large for the checks to be used in released software. We therefore propose an ISA and CPU extension using the meta-check instruction. The meta-check loads the bounds and stores them into physical registers, and associates with a pointer register a set of micro-ops to be inserted to perform the dynamic check whenever that register is used to generate an address. This resulted in an average slowdown of 21.2%.

II.I Acknowledgement

Section II.F contains materials to appear in "Bounds Checking with Taint-Based Analysis", in *2007 International Conference on High Performance Embedded Architectures and Compilers (HiPEAC 2007)*, W. Chuang, S. Narayanasamy,

R. Jhala and B Calder. The dissertation author was the primary investigator and author of this paper.

III

Transactional Memory

III.A Introduction to Transactional Memory

Effective utilization of multi-core processors is stymied by the difficulty of programming multi-threaded programs. Failure to obey data dependencies between threads results in race conditions where variables are modified in patterns not possible if the program were executed by a single thread, and often modified nondeterministically. This results in bugs that are extremely hard to detect, understand and replicate. Many serious bugs are attributed to race conditions. The 14 August 2003 blackout of large parts of North America was caused in part by a race condition in widely-deployed electrical monitoring software [68]. After the blackout, even though the engineers knew which software module failed, it still took eight weeks of searching through millions lines of code to find the root cause of the race condition. The Therac-25 radiation machine mentioned earlier was another example of a race condition in its control software, that killed at least five patients [51].

Race-conditions causes unexpected results from concurrently executing threads due to different interleavings or rates of execution of instructions. Different rates of execution are caused by non-determinism inherent in any general

purpose system like disk access, network traffic, etc. Shared variables communicated between the threads are often timing dependent, and will result in different values forwarded hence different non-repeatable results generated. Programmers often find these unexpected results to be wrong.

Most parallel programs use locks and other synchronization primitives to impose an order between threads that use shared memory to communicate. Poor lock usage results in incorrect code and performance penalties [73, 40]. Too coarse-grain a lock results in threads inefficiently waiting when they could otherwise be executed in parallel. Too fine a granularity adds programming complexity and increases the likelihood of deadlock or other incorrect behavior. Of course this assumes the programmer correctly knows where to place the synchronization primitives.

Thread priorities introduced for scheduling real-time tasks, compounds problems with locking. A low priority thread may acquire and lock a resource. If a high priority thread requests the resource, it has to wait on the low priority thread to complete ownership, thus essentially inverting the priority of the threads. In the worst case the high priority thread might pre-empted the high priority thread, never allowing it to release ownership of the resource, thus causing lack of forward progress and deadlock.

We want to consider a multi-threaded programming model that allows the program to avoid data-races called Transactional-Memories, by providing the appearance to any external viewer of memory, atomic execution of code regions. Threads concurrently execute Transactional-Memory (TM) regions speculatively and any memory access that aliases with another thread's TM memory access is detected, undone by speculation recovery, and the threads' execution serialized to prevent data-races. If no conflict is detected, then the concurrent threads execute freely. This conflict-free execution provides atomicity guarantee for the transac-

tion's execution. A further specification of Transactional-Memory called Ordered TM can provide completely deterministic control flow. Determinism is desirable for debugging, or for applications that need a specific execution sequence.

Multi-threaded programming models using speculative memory, must perform two tasks- detect conflicts and recover from these conflicts. Doing them efficiently under all conceivable operating conditions is a significant challenge. We focus on Hardware Transactional Memories (HTM) [40], which provide these features while maintaining good performance. HTM general characteristics are that it facilitates conflict detection and recovery through the cache hardware commonly found in processors, using cache coherence to detect conflicts and cache block versioning to maintain speculative and non-speculative state. If a conflict is detected from coherence, the hardware can quickly abort, discarding the speculative memory modifications, and recovering to the non-speculative version held in cache. Details of the HTM baseline will be discussed later in section III.C on related work.

One critical issue with HTM is what to do when a program's transactional working set exceeds the cache capacity, or when the cache contents must be evicted due to a context switch. We say a Transactional Memory is *unbounded* when its capacity is not limited by processor buffering capacity e.g. cache or write buffer. We also describe a Transactional Memory as *virtual* when it stores transactional state in virtual memory allowing context switching. The implies that an unlimited number of TM threads can exist and that these threads can share managed resources like the CPU or IO. With both capabilities, unbounded and virtual transactional threads are treated like any other thread but additional safety guarantees. Several recently proposals have been made (LogTM, TCC, VTM, LTM, UTM, Hy-TM), but each are limited either (or combined) by having bounded capacity, being unable to context-switch, or having poor per-

formance. We propose a *Paged Transactional Memory* (PTM) approach that at once resolves the generality and performance issues troubling HTM by using paging features already present in modern processors. Essentially we use a second shadow page plus the original home page to store the non-speculative and speculative state. We also use conflict detection hardware organized by blocks in pages that map neatly into bits in a bit-vectors. There is one set per page per transaction, simplifying organization. Because the technique does not copy data during transaction commits and aborts, it is faster than prior techniques. We contrast PTM to a prior technique that supports unbounded and virtual Transactional Memory called VTM.

In this chapter we will discuss how our PTM version of Hardware Transactional-Memory can simplify multi-threaded programming by eliminating data-races. First, in this section III.A, we define the race-condition problem, and the solution to guard critical sections by locks and transactions. In section III.B, we introduce and contrast the programming models for TM and TLS. Next we ground the work in this chapter by understanding earlier publications on Hardware-Transactional-Memory (HTM), Software-Transactional-Memory (STM), Thread-Level-Speculation (TLS), and databases, in section III.C. Most prior HTM, and TLS depend on the cache to hold speculative memory updates. We introduce the Page-Transactional-Memory (PTM) technique transactional data in virtual main memory, allowing virtualization of transactional state and unbounded state size. This is discussed in section III.D. We compare PTM's performance to VTM and Lock synchronization. There are limitations common to all cache based HTM including our Page-Transactional-Memory, due to cache line false conflicts that cause extra aborts hence reduce performance. We introduce a technique to eliminate false-conflicts. PTM performance is discussed in section III.E.

III.A.1 Race-Conditions

Concurrent execution often allows interleaving of reads and updates to shared variables that causing different results due to different ordering of the reads and updates. The different orderings are in turn caused by external events such as paging or context switches. Whether or not this non-determinism in outcomes is a bug depends largely on whether the application can tolerate non-determinism. One example is a credit-card account that accepts atomic deposits and purchases in parallel. If the credit-card account can tolerate negative and positive balances, then reordering a sequence of deposits and purchases is associative hence results in the same final balance. This property is useful if we want to do parallel updates concurrently, where different executions of threads will result in different sequences of updates, yet compute the same final result. If the credit-card account must always remain above a certain negative limit to prevent *overlimit*, then order does matter, as a purchase before a deposit may create a negative balance that is not allowed. In this situation the non-deterministic execution is not desirable because it can result in a negative overlimit balance in some sequences of execution. Non-determinism caused by concurrent execution is called a *race-condition* and many parallel programming styles attempt to avoid it. If non-determinism results in a bug, the random outcomes generated by race-conditions often makes the bug hard to replicate, isolate and remove.

There are two types of race-conditions called *general-race* and *data-race*, both resulting in non-deterministic execution of concurrent threads that may not be intended by the programmer. Race-conditions as defined by Netzer and Miller [61] are caused by concurrent data-access to a shared memory without synchronization. *General-races* in concurrent execution occur in programs meant to have deterministic execution but do not. What is meant by deterministic execution is that the threads update memory in some order. The previous

non-negative balance credit-card account is one example of a general-race caused by non-associative operations. Another example is maintaining proper ordering of data-dependencies. *Data-races* are different in that the programs are considered non-deterministic but lack atomic memory update by concurrently executing threads. *Atomic* execution of a section of code means that the outcome of the section depends only upon the initial state and the operations of that thread, and not upon any other thread. A *critical-section* is one that depends on atomic execution to behave correctly. From the credit-card account example even when we allow negative balances, if the the deposit code was not atomic, we could find sequences of execution that would compute the incorrect final results. We observe that deposit and purchases typically occur in two memory accesses on modern processors- a read followed by a write. If two threads are trying to make a deposit for each thread, one possible interleaving might read the same credit-card account at the same time, and then add the deposit, and then write the shared variable one after another, causing one of the deposits to be lost. This is obviously different than if the deposits execute serially one after another. We refer the reader to Netzer and Miller [61] for a formal definition of these properties.

III.A.2 Lock Synchronization and Transaction

We can guarantee atomic execution of critical sections through two different techniques. The first depends on *synchronization* primitives like locks to prevent concurrent access of the critical section. The second uses speculative execution to *transactionally* execute the critical section, allowing concurrent execution of critical section if there are no conflicts, but preventing concurrent access to shared hence conflicting variables.

Locks Synchronization

Synchronization primitives such as locks and barriers modify control flow to regulate concurrency. It can prevent concurrent execution in a critical section as in the case of locks, or to enforce a schedule of execution as in barriers. Locks enforce atomic execution of a critical section by allowing only one thread to acquire the lock, leaving other threads attempting to obtain the lock to block on the lock until the acquiring thread releases the lock value by writing a value denoting its availability. Subsequently the remaining threads compete for the lock, where one thread acquires the lock by successfully writing the lock variable, and the process repeats. When a thread blocks, the thread spin-waits or sleeps at the lock until the lock is released, and effectively blocks execution at the lock. Because blocking creates the possibility of deadlock, or issues with priority inversion which is a problem in real-time systems, there many proposals to avoid locks.

Barriers synchronize the execution of multiple threads such that threads block at the barrier until a given number of threads arrive, and then all are released. Barriers block using the same methods as locks. There are other types of synchronization primitives such as semaphores, and conditional variables not further discussed as they are not needed in later discussion.

One problem with lock based synchronization is *deadlock* that prevents forward progress when the thread is unable to obtain the lock. This will block the thread by spin-waiting forever or switching out the thread but never returning. Deadlock occurs when there is circular request dependency between the owner of a lock and another requester. This might occur due to lock variable management, where taking care of the status of multiple, concurrent variables indirectly representing control flow may be confusing to the programmer. More precisely deadlock occurs under four "Coffman conditions" [16]: (1) mutual exclusion- a

resource is either given to one thread or not at all, (2) hold and wait condition- a thread may request new resources when it already has another resource, (3) no preemption condition- only that thread may release the resources it owns, (4) circular wait condition- two or more threads form a circular chain of request, where each thread waits on another thread for a resource. One technique to break deadlock is by preemption and rollback, that is stopping one of the threads, forcing it to give up its lock, and restarting it after the other threads complete their critical sections. However if occurs too many times to the losing thread, preemption may result in livelock.

Livelock also results in a lack of progress, however threads in livelock do not have a circular request dependency. Livelock threads can even complete instructions, albeit so slowly that its often not perceived. It is caused by frequent preemption, or restart on the thread such that the thread is "starved" of forward progress.

Figure III.1 illustrates an example of lock synchronization primitives for the credit-card account example.

We guard the critical sections of deposit and purchase with lock synchronization as we intend concurrent execution. If locks were missing we can see that there could be interleaving of the read and write operations that would not be replicated by a serial execution equivalent. This leaves the *account_balance* inconsistent. We have taken into consideration what happens when there is a credit overlimit condition in the account when there is insufficient credit that as a result makes a call to *overlimit*. Lets re-examine the example with a definition of *overlimit* in Figure III.2.

We can see that depending on the global flag, *global_debug* that it may execute a signal or may redeposit back the money withdrawn. If we deposit, while within a purchase, then the lock *account_lock* is unavailable, causing deadlock.

```
lock_t acount_lock;
int    acount_balance;
int    account_overlimit;

void deposit(int amount)
{
    int sum;
    lock(account_lock);
    sum  = acount_balance + amount;
    acount_balance = sum;
    unlock(account_lock);
}


int purchase(int amount)
{
    int diff;
    int given;
    lock(account_lock);
    diff  = acount_balance - amount;
    acount_balance = diff;
    given = amount;
    if(acount_balance < account_overlimit) {
        overlimit(amount);     // maybe raise exception?
        given = 0;
    }
    unlock(account_lock);
    return given;
}
```

Figure III.1: Pseudo-code illustrating Lock Synchronization

```
void overlimit(int amount)
{
   if(global_debug) {
       raise(SIGUSR1);       // signal SIGUSR1 interrupt
   }
   else {
       deposit(amount);
   }
}
```

Figure III.2: Pseudo-code for overlimit that illustrates deadlock

One reason why such cases maybe difficult to find is that say the program was debugged using the "raise" system call to handle overlimit error. For production they used deposit thinking that silent behavior was better, masking the deadlock until then.

Transactions

Transactional Memories solves the deadlocks problem seen with lock based synchronization by using speculative execution and rollback if something violates the transactional properties derived from databases. Database *transactions* maintain *ACID* properties- atomicity, consistency, isolation and durability [28]. These properties allow a database to run reliably even in the presence of concurrency and unreliability of components. We describe the ACID properties next:

- **Atomicity** guarantees that the updates will either be applied completely or not applied at all.
- **Consistency** refers to the properties applying at all transactional sections and throughout executing the transaction. If any atomicity violation occurs then the transaction must back out the action to maintain consistency.
- **Isolation** refers to whether activity in the transaction is visible to an external observer.
- **Durability** species that state is persistent even after soft failure e.g. power outage.

Transactions [40, 41] also have a *linearizability* property that states transaction appear to execute serially. Steps of one transactions never interleave with another transaction, as they execute atomically. This property also

implies that there exists an ordering of transactions executed on a single thread that will have the same result as the concurrent execution.

Transactional Memory only takes a subset of this- atomicity, consistency and isolation. Transactional memories do not need state durability as this is unnecessary for general computation that enables non-blocking, concurrent execution. Transactional memories maintain ACI property by using transactions or speculative execution that can be rolled back upon conflict. Transactional memories will be discussed further in the next section.

III.B Transaction Software Model

Parallel programming models help the programmer express how to partition the program to minimize the cost of concurrent execution. Hardware support influences these models, and in this section we will examine some of the properties tied to hardware that distinguish parallel execution models. Principally we are interested in the programming models' ability to eliminate race-conditions, and to use transactional threads like any other thread (generality) except with additional safety guarantees.

Transactional Memory (TM) programming model has several advantages over prior concurrent models: (1) Transactional Memory's provides atomic execution, that protects critical sections of code against data-races, without deadlock because transactional memories provide *non-blocking* execution. This guarantee is particularly attractive when protecting fine-grain or nested lock regions that are prone to deadlocks. We believe that such regions will become more prevalent as programmers try to eke out greater parallel performance. (2) Another advantage is the simplicity of transactional memories programming where it just needs simple begin and end instructions markers. (3) Transactional memories enable greater performance over locked based execution, by allowing parallel

execution of critical sections so long as there are no run-time data conflicts, that otherwise would have to execute serially. There are additional variations of transactional memories ranging up to Thread-Level-Speculation that enables more powerful programming models that ease multi-threaded programming, and provide full virtualization and unbounded memory support. In this section, first we describe how to program transactional memories and how the execution model works, and second we contrast variations on the basic transactional memory model.

III.B.1 Transactional Memories Programming Model

Programming with transactional memories is easy. The programmer places at the entry(s) of the atomic region a *TM_start* instruction, and at the exit(s) of the region a *TM_end*, enclosing the region. Executing transactional regions is similarly easy to reason about as we summarize next. When *TM_start* executes, the system stores a checkpoint at that instruction, then continues execution. If transactional code has no data conflicts, they can execute in parallel. If there is a data conflict that could cause a data-race due to accessed shared variables between two or more threads, all but one thread stops execution by either stalling until the conflict is resolved, or performing *abort* that restores the checkpoint and re-executes the region. This allows the winning thread to complete the region, while preventing inconsistencies in memory. When *TM_end* executes, speculation stops and the transaction commits. *Commit* releases the checkpoint, and makes the speculative state into non-speculative that is now exposed to other threads. We now examine the policy level features of transactional execution in detail. Mechanisms for transactional memory are discussed in section III.D in the context of our proposed PTM transactional memory system.

Recovery (Abort/Stall)

In order to recover from atomicity violation, we need a safe state given by a checkpoint and recovery policy to restore that checkpoint. Conceptually checkpointing stores the entire state at the *TM_start* instruction, whose state is non-speculative. Registers are typically fully copied and stored this way. However checkpointing the entire memory state including state unmodified by the transaction is expensive, so we only checkpoint memory updates and maintain the last update to a given location. Execution afterward, within the transaction, is speculative. Updates within the transaction modify speculative state but not the checkpointed state, hence doing abort restores the checkpointed state as found at the *TM_start*. *Abort* is assumed to be handled atomically by some mechanism to be discussed later. Speculative state will co-exist with the checkpointed state at a given location by versioning memory into two "copies"- one speculative and one non-speculative although not necessarily at the same level in the memory hierarchy. To do a complete and general checkpoint, all speculative state throughout the memory hierarchy- at register, cache, main physical memory, and virtual paged-out memory is monitored and saved as necessary. Similarly recovery must function at all levels of the memory hierarchy. In fact general recovery has been an area of recent interest including that of this thesis because of the complexity and difficulty; earlier transactional memory implementations limit the recovery scope to registers and cache.

An alternative approach for recovery is stalling the thread [57], instead of aborting and re-executing. The system arbitrates as before. At the point of conflict detection, the system reverses the effect of the losing conflicting instruction and stalls that thread, until the winning thread completes the transaction. While functionally this has the same result as doing the abort, it eliminates the re-execution overhead from the losing threads, potentially increasing performance.

Also stalling can be used for recovery on losing non-transactional threads, which otherwise cannot abort. Unfortunately stalling does not work for all cases. If a transaction updated (wrote) a location previously, but loses arbitration then stalling does not help recover the state before the write. In this case abort is necessary.

Arbitration

Once a conflict is detected, the transactional memory system must figure out which threads need to recover, and which single thread may complete execution of the transaction. The system uses a process called *arbitration* [72] to decide the "winner" and "losers", by considering properties that guarantees forward progress and possibly other programmer specified requirements. One general strategy is to use an aging property that selects the oldest running transaction to be the winner, and allows losers to re-execute until eventually they become the oldest transaction. One variation assumes a global clock that provides a time-stamp at the initial transaction starts, which is retained even after restart from abort [2]. Maintaining a global clock accurately across a distributed system may pose difficulties as it it may not scale efficiently. Another variation uses a distributed clock based on communicating a local clock through conflict messages to loosely synchronize. After a successful commit, it increments the local clock, and takes the maximum of the sent clocks and local clock to discover the new clock [72]. Programmer specified properties are discussed latter in subsection III.B.2, in the context of commit ordering. Arbitration can be done centrally or locally depending on the trade-offs of simplicity versus distributed-system scaling.

Without arbitration providing the non-blocking execution, conflicts can cause transactions to deadlock when transactions have cross dependencies that

causes mutual restarts. Long running transactions may also be livelocked as they eventually get preempted by shorter running but conflicting transaction. Unfortunately early Transactional Memory proposals did not include arbitration so lack guarantee of forward progress. Current proposals including our own provide arbitration and a stronger guarantees of at least obstruction free execution. Depending on conditions of the code to be transactified, this may result in wait-free or lock-free code.

Commit

Once execution reaches the end of the transaction, it finishes with a commit. There maybe an arbitration check to order the *commit* before ending, which is discussed later. If the check is successful or not necessary, the commit discards the checkpointed state, and makes the speculative state non-transactional. If not it aborts. Commits are atomic operations.

How abort and commit actually manage copying state depends on upon two properties [57]. First one must select where the recovery copying cost must occur. In creating the speculative and non-speculative memory versions, typically one is copied from the other and leaving the original in place. If one selects speculative data to be the in place location, then abort must copy non-speculative checkpoint data, and commit leaves memory in place making commit fast. Alternatively if one selects non-speculative data to be in place, then commit must copy and abort keeps data in place. This makes abort fast. Second any data copying during commit or abort may be *eager* or *lazy*. *Eager* commit or abort completes any copying before letting anything else execute. *Lazy* will allow the thread to execute before all the copying is done, to reduce the performance impact, while remaining copying continues in the background until completed. This appears atomic as long as any memory access does not observe the inconsistent state, so

loads and stores attempting to access uncopied state must stall.

Conflict Detection

Transactional atomicity maybe violated anytime an external memory access causes the thread to calculate a different result than if it was not executing concurrently, or anytime a transactional write to shared memory is observed by a read before the completion of the transaction. The essential process of violation detection then becomes detecting if there is a shared memory between two or more threads, one of which is a transaction, upon which a write occurs in one thread, and a read or write occurs in another thread. Thus violation detection keeps track of transactional reads and writes for a given thread, and observes subsequent external threads' reads and writes to see if there is an aliasing memory access. We can describe these aliases in terms of dependencies by a pair of threads as Read-After-Write (RAW), Write-After-Write (WAW), Write-After-Read (WAR), and Read-After-Read (RAR). The temporal ordering on current memory operations is precise because the memory system will serialize simultaneous multiple memory operations to same location. The system then filters out dependencies that would not cause atomicity or isolation violation, meaning all RAR. This leaves RAW, WAW, and WAR dependencies to trigger abort and recovery. Isolation maybe weakened such that they only apply to other transactional threads (weak atomicity) [10] as opposed to all threads including non-transactional (strong atomicity) [14]. Other models such as Thread-Level-Speculation allow forwarding of speculative data, meaning that RAW dependencies can be resolved without abort and recovery.

Conflict detection may occur at the point of the memory access where it is termed *eager* [57], or it maybe delayed after the memory access where it is termed *lazy*. Eager conflict detection reduces the amount of wasted work

before calling recovery, however lazy conflict detection when done just prior to ending ordered transactions filters away false dependencies (WAW and WAR) from causing recovery. While most transactional memories are eager including ours, TCC [73] makes use of lazy conflict detection to recover only on true RAW dependencies violations as discussed later.

III.B.2 Details of Transactional Memory Programming Model

This subsection describes additional features of programming with transactional memories in regards to safe programming. These issues are forward progress guarantees, composition due to nesting, system calls, and commit ordering.

Non-blocking Execution

Transactional synchronization is notable for providing some guarantee of forward progress even when another thread is in conflict. Any programs that guarantees completing some operation under conflict and does not block (wait) is considered having *non-blocking* execution, a broad term for forward progress. Because it does not allow blocking, these programs do not have locks. Unfortunately being free of locks does not eliminate deadlock because it is still possible to create two block-free transactional threads in a mutual memory-dependency cycle with control-flow that enters spin-waits. These threads wait for each other to complete its transaction, but never does so [10]. Livelock may still afflict non-blocking programs. To provide more guarantees, there are three more restrictive definitions of forward progress- obstruction-free, lock-free, and wait-free- going from least to most [37, 72, 38]. All of these definitions assume the threads do not crash.

- *Obstruction-free* execution guarantees that some operation will complete when the thread runs in isolation meaning by itself, without any conflict from any other thread. Livelock and deadlock may still occur on programs that are obstruction-free. All obstruction-free programs are also non-blocking programs.

- *Lock-free* execution guarantees that some useful operation completes after a finite amount of time on some thread. Under this guarantee in the worst case, only one thread needs to make forward progress in the system yet be considered lock-free. This does protect lock-free programs from deadlock, but not necessarily livelock. In the forward progress hierarchy, all lock-free programs are also obstruction-free.

- *Wait-free* execution guarantees that all threads will make forward progress executing useful instructions after a finite amount of time. By being starvation-free it is free from both deadlock and livelock. In the forward progress hierarchy, all wait-free programs are also lock-free.

Programs with transactional memory are generally obstruction-free. However with the use of arbitration and restrictions on control-flow in the critical sections, one can create lock-free or wait-free programs.

Nesting

Placing additional start and ends within a transactional region will *nest* the transaction, meaning a transaction scope can be enclose another. Nesting transactions creates scopes out of matching start and end transaction instructions. Nesting transactions composes into another transaction, meaning that the atomicity guarantee applied to the components also applies to the composed whole. Function calls inside transactions whose encapsulation hides control flow,

increases the likelihood of nesting even if undesired, hence all proposed transactional systems provide some support for nesting. Checkpointing capabilities governs the recovery of nested transactional memory regions as they either recover to the outermost scope or the nearest enclosing scope. Doing this at the outermost enclosing region only needs to save one checkpoint, but will cause recovery to restart from a larger distance than from the nearest enclosing start [34], thereby *flattening* the nest. This is the most basic form of nesting support allowed but still provides correct recovery execution, and has the least state and implementation cost. A more efficient recovery implementation will recover from the innermost enclosing nest [14, 58], but has greater state storage requirements as a checkpoint is made at each enclosing transaction start, and greater implementation costs as abort must pick the right checkpoint to recover from. The implementation must also recover an abort or perform a commit to speculative state, when processing a nested region, which was not previously required. For the work in this thesis we consider only flattening nested transactions as it is orthogonal to nesting, but note this is an area of intense research.

System Effects

Handling system effects such as virtualization is an area of great interest recently to researchers due to the many difficulties it poses. System calls, interrupts and exception handling potentially swap out the current thread's execution state and installs another through context switching. If this occurs either the transactional memory must be virtual, or the system must assume context switching does not happen [40]. Virtual Transactional Memory means all transactional state can be saved and later restored, thus allowing multiple transactional contexts to co-exist [73]. Our work uses the virtual TM policy, as the latter for many systems is an unrealistic assumption. Virtualization introduces another

problem: Memory updates to a context-switched out thread requires the system to also perform conflict detection on all threads in addition to those running, and perform recovery if necessary when that thread is restored. A different problem is system calls within transactions may have side-effects that cannot get undone by abort, such as IO to network or file-system access. The common solution [30] is to serialize execution of system calls by pre-validating one thread that will definitely commit, while all other threads abort or stall. We assume this approach subsequently. Not very much research progress has occurred in this area, largely we feel because of the great difficulties, however its importance should spark interest [99].

Mixing Transactional Code with Non-Transactional Code

For many programs transactions are placed around only the critical sections of code leaving the rest non-transactional. In this programming style there will be a mix of transactional and non-transactional code that interacts, with the complication that arbitration may not know the age of the non-transactional code and that non-transactional code cannot abort if recovery is necessary. One solution is to treat the non-transactional current instruction as a transaction and aborting that instruction is effectively stalling as described before. This also provides an aging mechanism thus solving both problems.

Hardware vs. Software Transactional Memories

The underlying infrastructure for transactional memory maybe hardware or software, or some hybrid in between. *Hardware Transactional Memory* (HTM) was first proposed by Herlihy and Moss [40], and uses hardware structures such as the cache to provide fast checkpointing and conflict detection but limits the HTM memory capacity. Shavit and Touitou shortly thereafter proposed *Soft-*

ware Transactional Memory (STM) [80] that performs checkpointing and conflict detection through software. Because the transactions are layered over the native virtual memory system, this makes STM transactions portable, unbounded in capacity, and can have multiple contexts. However STM is slower than hardware since every speculatively updated memory must be tracked and checked for conflicts by using some software verification method. Kumar et al. [50] measured STM to be 2-7X slower than HTM. Commit and recovery is similarly slowed by software execution. STM also requires explicit programmer-written function or language level support to perform the conflict detection and any recovery, whereas HTM offers a simpler programming model described earlier. Efforts by several groups [2, 73, 57] have focused on making HTM virtualizable and unbounded by storing contexts to virtual memory, though at some runtime penalty. Our work in this thesis continues the effort by further minimizing the virtualization and unboundedness overhead.

Ordering Constraints

Execution ordering affects the correctness, the starvation-free execution, and the performance of the transactional memory. Ordering refers to scheduling imposed on transactional threads' execution at arbitration when a conflict is detected as described earlier or when the transaction ends. Adding arbitration at transaction end allows the system to schedule commits, enforcing data-dependency between the threads because that is when data becomes exposed to other threads. Threads win and lose commit ordering, just like conflict arbitration, where there is one winner allowed to execute and the remaining losers perform recovery. For example, using the credit card example to show the safety derived from ordering, we might specify an order on all deposits to execute before purchases. This prevents a temporary negative balance that was previously

disallowed. As noted earlier, the forward progress guarantee is necessary because unconstrained abort and recovery may pathologically prevent some threads from ever moving past that transaction, causing starvation. A model that still lacks any other ordering other than forward-progress, is called *unordered TM*. *Ordered TM* at commit use some other age metric usually specified by the programmer to verify that no "older" transaction is executing, performing recovery until all older transactions commit. Ordering commits eliminate non-determinism in the execution order as repeated execution of the program provides the same commit order, eliminating general-races, yet ordered TM still allows concurrent execution of non-conflicting transactions. It simplifies the debugability of the parallel code, because commit order is now deterministic. The disadvantages of Ordered TM are that the constraints reduces opportunities for parallel execution, and it requires all concurrent execution be transactional.

Ordering age information maybe passed to the transactional system by using phase-numbers [34], integer numbers whose values imply an ordering. Typically smaller value implies "younger" when compared against several phase numbers. When the phase numbers are equal, then the no ordering is implied and conflict resolution is only dependent on forward progress requirements. This enables *phase-numbers TM* to implement both ordered and unordered conflict resolution.

We might order commits according to the execution order found in a single-thread execution giving us the *Sequential TM*. Sequential execution order for conflict resolution has the advantages and disadvantages of Ordered TM. It further simplifies debugging because the programmer is able to follow the single thread execution order more easily when looking at the source code. Further any partitioning error that causes a data conflict hence recovery, simply falls back to serial order. However Sequential TM has the disadvantage that it requires

a constrained parallel partitioning and ordering that will serialize to a feasible control flow in the original sequential program.

Providing a determined order of execution over the entire program execution requires that all concurrent execution be transactional. This allows commit arbitration to totally determine ordering over concurrent execution thus enabling the total ordering. Mixing non-transactional and transactional code weakens the ordering, as transactions executing with non-transactional code allows the transaction to complete out-of-order with respect to previous and subsequent transactions on the non-transactional thread. Ordered TM and Sequential TM requires total ordering to maintain its semantics hence must be completely transactional during parallel execution. Current ordered TM proposals require executing transaction all the time [34], while unordered TM may mix transaction and non-transactional code [40]. However we conjecture that it maybe possible to integrate the ordered and unordered TM code by separating them with a barrier, though a thorough exploration has yet to be done.

III.B.3 Related Model: Thread-Level-Speculation (TLS)

Thread-Level-Speculation whose modern form was first proposed by Sohi, Breach and Vijaykumar [81] takes a sequential (non-concurrent) program, and speculatively creates and executes concurrent threads. It maintains a non-speculative primary thread that the speculative thread can fall back upon due to data misspeculation, or from any other recovery. The speculative threads attempt to perform useful work which the non-speculative thread can then avoid, thereby increasing the run-time performance. Like TM it may nest regions, but unlike TM each nest represents a distinct forked thread, that when joined will commit that region. Also like transactional memories TLS speculates memories by using memory versioning, but unlike TM, it may have multiple speculative versions

corresponding to the nested speculative threads. TLS also speculatively forwards results from older threads to the younger. This causes TLS to look for memory conflicts like Transactional Memories but differs in that it is looking for data mis-speculation that occurs when data has been forwarded but is redefined in the older thread. Other data-conflicts such as WAW, and WAR are hidden by memory versioning. TLS has several advantages that are offset by its steeper implementation cost. TLS is easier to program and debug because it has the appearance of a single-threaded program. TLS also performs automatic parallel partitions of the program through hardware or software without the intervention of the programmer. We provide TLS and the previously described Lock-synchronized parallel programming models as a counterpoint to Transactional Memories.

III.B.4 Transactified Examples

With transactional memories defined, we return to the potentially deadlocked credit card account example, to see how transactional memories can solve that problem in figure III.3. When written as transactional memory, the path through the code that goes through purchase, overlimit, and deposit no longer deadlocks. It does create a nesting of transactions though that composes properly and does not deadlock.

Transactional memories can also protect against other types of memory bugs. Take for example the loop in Figure III.4 partitioned for concurrent execution. As it has loop carried dependence between `a[i]` and `a[i-1]`, clearly the order of the loop iterations matter, and a naive parallel execution will often result in incorrect results due the race-condition. However such errors are not apparent to automatic parallel partitioning such as OpenMP compilers, that will generate code regardless [54]. Using transactions all the time in such cases will

```
int acount_balance; int account_overlimit;

void deposit(int amount)
{
    int sum;
    tm_start;                  // transaction start
    sum  = acount_balance + amount;
    acount_balance = sum;
    tm_end;                    // transaction end
}


int purchase(int amount)
{
    int diff;
    int given;
    tm_start;                  // transaction start
    diff  = acount_balance - amount;
    acount_balance = diff;
    given = amount;
    if(acount_balance < account_overlimit) {
        overlimit(amount);     // maybe raise exception?
        given = 0;
    }
    tm_end;                    // transaction end
    return given;
}


void overlimit(int amount)
{
   if(global_debug) {
       raise(SIGUSR1);       // signal SIGUSR1 interrupt
   }
   else {
       deposit(amount);       // previously could cause deadlock
   }
}
```

Figure III.3: Pseudo-code of credit card example using transactions synchronization

```
int loop(int slice, int width, int order)
{
    int i;
    // each slice maybe executed in parallel
    tm_start(order);
    for(i=slice;i<(slice+width);i++) {
        a[i]=0.5*a[i]+0.5*a[i-1];
    }
    tm_end(order);                          // if conflict, commit in order
}
```

Figure III.4: Transactified loop example- Ordered transactions protect against loop carried dependency

detect the dependency and sequential ordered will correctly serialize execution of thread that will obey the loop-carried-dependency.

III.B.5 Programming Models Comparison

We categorize the properties of the different hardware transactional memory programming models in terms of their ability to eliminate race conditions, and their different hardware requirements. We distinguish several Transactional Memory models based on ordering constraints- unordered, ordered, sequential ordered, and phase-number ordered TM. Though our work does not mandate ordering, ordering complements PTM by providing additional safety guarantees. On the other side of the coin our PTM work enables ordered TM models as they can execute on unbounded and virtualized TM hardware.

Later in some comparisons we also lump together Transactional Memory except TCC TM [30] as "TM", as the various proposals share the described characteristics. We similarly lump together the many published Thread-Level-Speculation as "TLS". Later in related work III.C, we distinguish several transactional memory models based on its virtualizability and unboundedness- HTM, STM, and VTM that enables general transactional memories.

Table III.1: Comparing Safety and Convenience of Transactional Memory Programming Models

	introduces deadlock	ordering	total TM
Lock	yes	no*	-
Unordered TM [40]	no	no	no
Ordered TM	no	yes	yes
Sequential TM	no	sequential	yes
Phase-Number TM [34]	no	either	yes
TLS [81]	no	sequential	yes

Comparing Ordering

We summarize the ordering capabilities of the different parallel programming models, comparing their ability to eliminate race-conditions in Table III.1. *Introduces deadlock* indicates whether that programming model's construct to maintain atomicity in a critical section via lock or transactions will induce deadlock. As discussed earlier locks will deadlock with other locks trivially. Our abstract transaction models will not introduce deadlocks as they are all defined to provide forward-progress guarantees. However other control flow such as spin-waits already present in critical section may cause the transaction to block, so we do not provide a systemic guarantee about forward progress beyond describing these TM's as non-blocking and obstruction-free. *Ordering* provide deterministic execution eliminating general-races. For lock-based synchronization, the asterisk (*) indicates that additional "conditional-variable" synchronization construct can provides ordering though lock synchronization by themselves cannot. Phase-Number TM allows the ordering to be specified by the programmer who may pick Ordered, Sequential or Unordered TM. Sequential TM and TLS allow sequential ordering that has the same commit order and effect as if there was a single thread of execution, a stronger requirement than ordering. *Total TM* indicates whether

the entire concurrent part of the program must be guarded by either TM transactions or TLS speculation. This is an indicator of programmer effort to port a program using that model as total ordering is much more effort than guarding only critical sections.

Conflict Detection Rules Comparison

Conflict detection is used to eliminate data-races as described earlier in section III.A, and is an important differentiator between transactional memory models, and TLS as seen in Table III.2. Most Transactional Memory (TM) models – Ordered, Unorderd, Sequential, HTM and STM – have the same memory conflict detection between threads, and will abort or stall on RAW, WAW, and WAR. TCC [34] is a variation of Transactional Memories that uses Phase-Numbering but eliminates false dependencies (WAR, WAR). Because they perform verification only at transaction end in preparation for the commit, TCC is certain that no other will commit between the verification and commit. This fixes the order of thread commits, so it only needs to consider true dependencies. TLS differs from TM by avoiding true dependency violations by using speculative data forwarding [81]. Lock synchronization does not perform conflict detection.

Debugging Comparison

We have already described properties that segregate the transactional memories, so now we compare properties that further distinguish TLS and Lock synchronization from transactional memory in Table III.3. The start and end of the speculative execution differ between TM and TLS, where transactions starts at TM_start instruction and commits at the TM_end, while TLS speculation starts at the fork and commits at the join. This causes differences in the control-flow. TM fork and join are separate from the transactional start and end,

Table III.2: Comparing Conflict Detection Actions of Parallel Programming Models Without and With Ordering- Read/Write conflicts with ordering are initially detected in some execution order. Ordering may change the dependency to the order found in the committed sequence.

	detected dependency		
	RAW	WAR	WAW
conflict	true	false	false
TM [40]	recovery	recovery	recovery
Unordered TCC TM [34]	recovery	no	no

	detected dependency/ordered dependency				
	RAW/RAW	RAW/WAR	WAR/WAR	WAR/RAW	WAW
conflict	true	false	false	true	false
Ordered TCC TM [34]	recovery	no	no	recovery	no
TLS [81]	forwarding	no	no	recovery	no

hence immediately after TM recovery we will find concurrent threads. TLS, with its speculative fork, kills off threads for recovery. If it finds multiple conflicts or requires serial execution, it can recover to a single thread. This single thread vs. multiple thread of execution (in recovered state) potentially simplifies debugability of TLS vs. TM. It also implies that fork/join performance is much more important for TLS than for TM.

As noted earlier TLS does data-forwarding between speculative threads compared to the isolation of transactional regions. Transactional memories isolate updates in their regions until commit, which simplifies reasoning of data-dependencies in those programs. Similarly Thread-Level-Speculation updates occur sequentially at commit which is also very easy to reason.

The number of memory copies varies among techniques. Speculative execution introduces two copies- speculative and non-speculative potentially per thread. TM has only two copies for the entire system, while TCC and TLS

Table III.3: TM vs TLS and Locks- Differences in concurrency and isolation affect how the debugger sees the program.

	speculation		recovery-	isolation	copies
	start	end	concurrency		
Lock	-	-	-	no	1
TM [40]	TM_start	TM_end	concurrent	yes	2
TCC TM [34]	TM_start	TM_end	concurrent	yes	N
TLS [81]	fork	join	single thread	forwarding	N

maintain as many speculative versions in their caches as there are threads to provide for the memory renaming capabilities (described as "N" in the table). Providing unbounded storage for baseline TM with two copies is difficult as we will show, however we suspect that providing more storage in TCC and TLS will prove to be a proportionately greater challenge. This later point is perhaps the most significant reason for choosing TM over TCC or TLS.

III.C Transactional Memory Related Work

In this section we discuss related work to our Page Transactional Memories.

We build a family tree of Transactional Memories research dividing them by major features. We start by discussing the ancestors to Transactional memories the early database systems, that perhaps unsurprisingly share many characteristics with our PTM technique. Next we peruse the design space for Transactional Memories that can be subdivided by implementation. This is bounded by Hardware TM, and Software TM, and is spanned in between by hybrid hardware/software TM. The latter Hybrid TM include Virtual TM systems that provides multiple contexts and unbounded state capacity through virtual memory.

III.C.1 Early Database Systems

Early Relational Databases Management Systems (RDMS) established many of the precedents of transactional systems. We describe two with relevance to PTM: the first System R [87] proposed the concept of versioning memory called "shadow paging", and the second CPR/801 [15] proposed fine grain conflict detection.

System R

System R [87, 9] was an early IBM research database system with dedicated software and hardware, which is considered to be the progenitor of modern RDMS. It introduced SQL, and provided high performance for that time with full ACID transactional property. System R used virtual memory to map pages of memory to disk to provide durability in case of soft system failure e.g. power outage. It used a two step process to log and checkpoint transactional state to disk to maintain the durability property. First System R logged committed state to disk, occurring after the transactions executed in memory. Second to prevent the log from growing too large, it periodically checkpointed the entire memory image to disk. As that disk based memory image was also used for swap, it versioned the disk swap space using "shadow paging" where there were two copies: a "new" page containing updates and the "old" page containing the checkpointed copy. This protected against inconsistency caused by soft failure after a checkpoint. Later we will see that our PTM technique uses a technique similar to "shadow paging" hence the usage of that name, but differs in that PTM does not focus on database persistence. Another difference is that System R used locking, and performed rollback only if a deadlock cycle is detected. This prevents opportunistic concurrent execution when there are no data conflicts.

CPR / 801

A successor RDMS to System R was the CPR Operating System running on the IBM 801 RISC processor [15]. Unlike System R that focused on SQL, CPR/801 was targeted towards memory mapped IO and transaction research. Chang et al. observed software transactional support would provide insufficient performance and more complex programming model, and used hardware assists to improve this. They introduced support for efficient fine-grain locks in hardware. Each 128 byte block of memory was organized as 16 blocks per page, and had a bit in a lock bit-vector that is associated with the page via the page table and TLB. When a transaction read or wrote the block and depending on the semantics of the transaction, it sets the lock bit. Memory access with other threads' locked shared memory caused conflicts on a block, thus raising a lock fault interrupt. This then in turn called a handler to undo the transaction and restarted it. The primary similarity between PTM and the IBM CPR/801 system is that PTM also associates a bit-vector used for conflict detection with each transactionally touched page. However, the PTM extensions are used for supporting unbounded Transactional Memories, as opposed to persistent database transactions. PTM also uses fine grained memory versioning recover the checkpoint, instead of the logging.

III.C.2 Hardware Transactional Memory (HTM)

The projects listed under this section use the cache and additional hardware structure as their state buffering mechanism, thus have finite storage capacity. Some like SLE and TCC have a fall back mechanism that uses serial execution to safely execute transactions without buffering support. However this eliminates parallel execution performance gains.

Herlihy and Moss

Herlihy and Moss [39, 40] proposed using hardware support for transactional memory as an alternative to lock-based concurrency control. In their paper all accesses to shared memory regions use special load and store instructions. Shared data accessed by transactions are kept in a fully-associative transaction cache and excluded from the regular cache. This cache allows a transaction to utilize the results of its previous stores without transactions on other processors observing the stores. Conflicts between the transactional cache and updates to memory are detected by observing writes to memory made by other processors using minor modifications to normal memory consistency protocols (either bus- or directory-based [39]).

Recovery with Herlihy and Moss Transactional Memory's is less convenient than all other described schemes. Conflict detection via cache coherence maintains a consistent view of transactional memory allowing a speculate update if no conflict exist otherwise aborting. However it does not automatically restart the transaction, as that policy is left to the programmer to encode. Also interrupts (including timer interrupts) will cause transactions to abort. This constraint and the limited size of the transactional cache mandate short transactions with small working sets. Nonetheless, Herlihy and Moss effectively generalized hardware-supported atomic memory operations to a number of words limited only by the size of the transactional cache.

SLE and TLR

Rajwar and Goodman considers speculatively converting lock guarded critical sections into transactions. The first proposal is Speculative Lock Elision [71] (SLE) that introduces the idea of dynamically detecting lock protected regions surrounding critical sections, removing the lock, and speculatively exe-

cuting the region using the same hardware found in out-of-order machines for branch prediction. It detects the locks by scanning for instruction sequences that are likely to be locks with high but not complete accuracy. Data conflicts between regions are found through cache coherence when it snoops the speculative write-buffer. If the capacity of the write buffer is exceeded or if there is a conflict, SLE re-executes the region with explicit locking. Speculative Lock Elision increases performance by allowing greater concurrency than lock approaches, yet is completely safe because missed regions still will execute with lock synchronization, and false positive regions execute transactionally.

The second proposal refines SLE in an approach called Transactional Lock Removal [72] (TLR) that provides both greater concurrency and wait-free execution. SLE may deadlock during lock based recovery, or livelock as cross dependency may cause repeated recovery. TLR introduces arbitration to prevent live-lock by ordering commits to provide forward progress. It also introduces conflict recovery through stalling and deferring the request by using cache coherence NACK's (negative acknowledgments). The deferrals can create dependencies cycles that induce deadlock, so TLR introduces an instruction that break those cycles by restarting younger nodes. The authors state that the limited storage capacity must be managed or it will prevent full guarantee of forward progress. Similarly unless critical section are explicitly marked so that TLR can turn the locks into transactions, it cannot guarantee blocking-free (deadlock free) and wait-free execution.

TCC

Transactional Coherence and Consistency (*TCC*) [31, 30, 34, 54] provides a novel hybrid of HTM and TLS techniques. TCC does not use existing memory coherence protocols to detect transactional conflicts. Instead, it defines

a new coherence protocol for CMP's based on transactions. TCC requires that all instructions are executed within transactions to ensure that all memory accesses will be coherent. Their implementation assumes that speculative updates are kept in an L1 cache with speculative state marked for every word. Unlike the previous proposals, the cache is not fully associative; to reduce associativity conflicts they add a victim cache. It also stores updates to a write-buffer to gather them for coherence message. If capacity is exceeded or system calls or IO encountered, TCC executes the threads serially. At commit time, a processor arbitrates for access to the bus and communicates from the write-buffer its memory updates in one packet. At this point conflict detection detects only true RAW memory violation. Other processors snoop the addresses and values, updating their caches and aborting transactions as necessary. Another unique features of TCC is flexible support for ordering commit operations through "Phase-Numbers". These can specify sequential order similar behavior to Thread-Level Speculation, and arbitrary ordering including unordered. A certain ordering of phase numbers can also be used as a barrier. By mandating transactions all the time, and ordering, TCC parallel programs guarantee commit forward progress, and need not worry about incorrect partitioning of the program that might cause a data-race. They determine that false sharing is a problem with TCC (and true for all TM and TLS), and experiment with different storage granularity.

As noted earlier, for a given memory location TCC supports as many speculatively updated copies of a given memory location as there are threads, thus making WAW conflicts unnecessary. This feature turns out to hinder virtualization and unboundedness by potentially requiring overflow buffering for that many speculative threads. For large numbers of threads this may pose a scaling problem.

LTM

LTM [2] appeared in the same paper as UTM and focused on what the authors felt was a feasible implementation. It supports reasonably large transactions with the memory footprint size comparable to that of physical memory, and uses the memory coherence protocol to detect conflicts. It uses an overflow bit in caches to let the coherence protocol know if there is potentially a conflicting overflowed transactional block. LTM stores all the overflowed speculative values in a memory-based hashed data structure until the transaction commits. This approach results in an efficient abort operation, but the commit operation can incur high overhead as the new values need to be copied from the backup structures to their corresponding memory locations. LTM can avoid conflict-detection overhead for non-overflowed blocks using its overflow bits, but it must do multiple memory lookups to resolve conflicts for the overflowed blocks. LTM cannot support transactions longer than a time slice or with footprints larger than physical memory.

LogTM

LogTM [57, 58], like LTM [2], supports reasonably large transactions that fit in the physical memory. LogTM uses a directory-based coherence protocol for conflict detection, but requires that transactional state never paged out because it maintains transactional state in the directory. It makes in-place memory updates for overflowed speculative values and stores non-speculative state in logs stored in virtual memory. Hence abort can potentially be a high overhead as the checkpoint is restored from its log. Also, aborts are handled in software with LogTM, which makes them costly. To ameliorate the abort cost, LogTM stalls the transaction whenever possible instead of aborting it. The LogTM approach does not handle thread migration, context switches and paging. LogTM

can also support nested transactions [58], to improve performance and better support system calls within a transaction.

III.C.3 Software Transactional Memory (STM)

In this section we describe Transactional Memory techniques that do not require support from hardware. Consequently they are portable and use virtual memory that can be swapped out and have a capacity unlimited by hardware (the exception being Shavit and Touitou). But STM must do additional work to support conflict detection and memory versioning. The most significant limitation for all STM is run-time performance. Unfortunately the only known direct comparison between HTM and STM is from Kumar et al. [50](Hy-TM) where they found STM performs worse than HTM by 2-7 X. STM proposals have thus far been theoretical (Shavit and Toitou) or target Java (Harris and Fraser, Atomos). It would be interesting to see if other languages such as C, which is important to the Open Source community, can be targeted.

Shavit and Touitou

Soon after Herlihy and Moss published (Hardware) Transactional Memories, Shavit and Touitou [80] proposed implementing TM in software as the TM was not available. There are limitations. The authors limited the scope and capacity of the transactional operation to make it provably correct. They also introduced versioning state directly in memory. When a transaction starts it must provide a set of data that requires ownership, limiting that data to what is known at compile time. Also context-switches forced their atomic swap primitives (Load-Linked/Store-Conditional) to fail ownership. Its unlikely but possible that repeated swapping could cause their transaction to block.

Harris and Fraser

Harris and Fraser [36] generalize the STM concept. First they observe that Conditional Critical Regions (CCR) make a convenient program language structure to express a transactional region. Second they implement this CCR's in Java, an example of a modern language, and consider language issues such as composition like nesting and interaction with other Java constructs like lock based synchronized regions. They eliminate the static data limitation found in Shavit and Touitou. Memory versioning happens through heap structures that provides a level indirection through meta-data. This extra layer provides notions of transactional ownership for conflict detection, and speculative/non-speculative versioning.

Atomos

Atomos is a STM proposed by Carlstrom et al. [14]. Like the earlier Harris and Fraser work, this paper provides CCR constructions in the Java language. Atomos then goes onto to add missing functional and performance language features found in other transactional contexts. It adds strong atomicity, missing in earlier STM work, to prevent isolation problems with transactional/non-transactional code. Atomos provides open and closed nested transactional support found in RDMS's. Committing closed nests do not expose the transactional state, whereas open nests does. Nesting in general reduces abort recovery cost while open nesting allows data to be available sooner to other threads. Atomos adds phase number ordering to their STM, inspired by their TCC project. Also Atomos provide handlers for commits and aborts inspired by RMDS's that execute following that operation.

III.C.4 Hybrid Hardware/Software Transactional Memory

These techniques use Hardware Transactional Memories for the expected common case, but fall back on software Transactional Memory (Hy-TM) or Operating System (Virtual TM) due to capacity or other consideration. VTM should belong in this section, but has been split from this section due to its importance in our later comparison.

UTM

UTM [2] is the first approaches to completely support unbounded transactions. UTM uses its XState data structure to log all transaction-related information. Each memory block has a "log pointer" associated to the list of transactions that accessed it. All writes done inside a transaction modify the memory in place, storing a copy of the old non-speculative value in the "XState Log". This approach makes abort a costly operation, though commit can be done very efficiently. UTM requires multiple memory lookups to traverse the log pointer on abort, since it does not cache the log entries, although it could potentially do so. The UTM approach can support most system events, including overflows, context switches, process migration, and paging. Their approach requires significant hardware changes including globally unique virtual addressing.

Hybrid TM

Kumar et al. [50] note that STM have as much as an order of magnitude slowdown versus HTM (measured 2-7 X). However STM is virtualizable and unbounded, so the authors of Hybrid TM[1] (Hy-TM) augments a STM with a cache based Hardware Transactional Memory. They modify Herlihy's et al. [38] Dynamic Software Transactional Memory (DSTM) such that it can run transac-

[1] Note there is another "Hybrid TM" by different authors in Asplos 2006

tional objects either in STM or HTM mode, and provide the means to do cross software/hardware domain conflict detection and reporting. At entry to a transaction the system chooses to run as HTM initially, but will select STM if it runs into cache capacity limitations. They find that Hy-TM runs slower than HTM, anywhere from 1 X (equivalent) to 2.6 X overhead but sometimes as much as 5.5 X slower.

III.C.5 VTM

As we compare our approach to VTM [73] we carefully describe its design. The VTM approach provides an efficient and nearly complete handling of unbounded transactions. The key structures needed to implement VTM are an in-cache hardware transactional memory system, and a set of hardware and software structures to handle transactional overflow and context switching. VTM is oriented towards in-cache TM with eager conflict detection, but is otherwise mostly agnostic about the particulars of the in-cache hardware transactional memory system.

The software structures for VTM consist of transactional state information (XSWs), a table tracking overflowed blocks and their original values (XADT), an overflow counter, and a counting Bloom Filter (XF). Unlike PTM, the addresses tracked by VTM for overflowed blocks are virtual. Instances of the software structures reside in the virtual address spaces of each transactional application, and are shared among the threads. The hardware structures needed for VTM are an XADT walker that performs lookups on overflowed state in the XADT and walks the XADT on commit and abort, and a cache of meta-data for overflowed blocks, called the XADC. The bloom filter XF is used to reduce the frequency of having to access the XADT when doing conflict detection. A set of counters in the XF will be incremented when a cache block is overflowed, and

are decremented lazily during commit or abort. A value of zero means that there is no overflow block, and a non-zero value means that there may be an overflow block.

The XADT log table contains the virtual addresses, transaction state, and data of the overflowed cache lines, buffering all speculative state. VTM uses the old value of the transaction-modified memory, also stored in the XADT, to detect non-transactional code interaction with transactional code. Whenever a transaction encounters a read or write miss, the XF will be consulted to determine if the memory block being accessed may have been overflowed in the past. If so, the corresponding entry, if any, in the XADT will be looked up to resolve the potential conflict. VTM accesses the XADT via the XADT hardware walker.

If no blocks are currently overflowed, then conflict detection beyond the in-cache mechanism consists only of checking the overflow counter. When there are overflows, VTM can avoid the overhead of performing conflict-detection for addresses that have never overflowed by filtering out queries to those addresses using the XF, but it requires XADT look-ups to resolve conflicts for overflowed cache blocks. VTM may cache transactional meta-data and/or transactional state in an XADT Cache (XADC). It stores the most recently accessed evicted transaction blocks, and a pointer to the XADT structure for that block in memory. The meta-data describes what transactions have read the overflowed block, and, if the block was dirty, which transaction wrote it. When a query to the XF says that there may be an overflowed block, we look up the block being loaded in the XADC. If there is a hit, then we have all of the information to determine if there is a conflict, and a pointer to the data blocks in memory to load the speculative block if needed.

VTM stores the new speculative value in their overflow data structure and the memory is updated on transaction commit. This allows fast aborts,

but results in memory-copying overhead at the time of commit. VTM can hide some of this cost by doing a lazy commit, but the memory updates still consume bandwidth, and all the transactions that need to access a memory block modified by a committed transaction, but yet to be updated in memory, have to stall.

We consider a variation of VTM as proposed by Zilles and Baugh [98], to write speculative blocks to memory when a block is evicted, and only store the non-speculative block in the XADT structure. This will make commit fast, which is the expected case. We provide those results in the result subsection III.E.3.

VTM virtualizes the execution of transactions across most system events, which include cache overflows, context switches, process migration and paging. However, they require that the cache blocks touched by the transaction be evicted from caches and invalidated before the transaction is context-switched out. Further, VTM needs to record virtual addresses for locally cached transactional blocks so that it can do the reverse address translation from physical address to virtual address. This enables VTM to evict all the cache blocks read or written by a transaction that is being context-switched out.

III.C.6 Thread-Level-Speculation (TLS)

In this section we briefly described thread speculation techniques, picking a representative design, Multiscalar, to contrast Transactional Memories. Other examples are Stampede [83], and Hydra [32, 33].

Multi-Scalar

The earliest TLS technique was by Sohi, Breach and Vijaykumar [81] in the MultiScalar project. Inspired by contemporaneous out-of-order speculation concepts, their goal was to run speculative threads ahead of the main, non-speculative thread, and perform useful work. Multiscalar used multiple processors

organized as a circular queue running threads ordered youngest to oldest, with the oldest being non-speculative. Memory forwarding happens from older threads to the younger, managed by the Address Resolution Buffer (ARB) that also buffers the speculative memory state. Registers are forwarded in similar fashion. Data mis-speculation may occur when a younger thread consumes data from an older thread that is later overwritten by the older thread (or any other older thread). If this inconsistency is detected, Multiscalar squashes the thread and any younger threads. Task partitioning for Multiscalar occurs in a compiler based on heuristics. In later papers Multiscalar considered memory versioning and forwarding through cache called the Speculative Versioning Cache [27]. Notably TLS is not unbounded. If it runs out of memory capacity or takes an interrupts, it squashes to single threaded execution.

III.D Paged Transactional Memory (PTM)

In this thesis we propose an efficient Transactional Memories called Paged Transactional Memories (PTM) that unlike prior Hardware Transactional Memories (HTM) proposals, supports multiple contexts and unbounded memory capacity. As the name suggests it uses the virtual memory system's to provide the backing store for transactional state, and by doing so provides PTM the ability handle transactional cache overflows to main memory, paging out of transactional data, context-switches, and thread migration. We build the memory versioning and conflict detection in hardware to make both fast. In this section, we describe PTM's policy decision and the mechanisms to implement it.

III.D.1 Structures

Transaction Cache State

For handling bounded transactions, PTM assumes hardware support similar to the architectures proposed in prior work [73, 2]. To support bounded transactions, we need to keep track of the read and the write transactional states for each cache block, and use the coherence mechanism to do an eager conflict detection [73, 2]. The eager conflict detection mechanism checks for a violation on every cache coherence miss. If there is a violation, the oldest transaction always wins the conflict.

In addition to augmenting the cache blocks with the transactional states and supporting eager conflict detection, we also need a checkpoint mechanism to abort and re-execute a transaction. Our approach assumes support for checkpointing the register state when starting the execution of a transaction, similar to the earlier studies [2, 73]. Apart from such basic transactional-memory support in the processor core, PTM does not require any other significant change in the processor core, as most of its functionalities are placed in the memory controller.

In our PTM design, we take care not to adversely impact the performance of transactions whose working sets fit within the transactional cache. As long as the cache blocks accessed by a transaction do not get evicted from the transactional caches, the basic on-chip transactional memory system handles the execution of the transaction, detecting violations, and providing support for committing and aborting of cache blocks. This is similar to how the bounded transactions are handled in prior work [2, 73]. To provide this functionality, we keep a global flag indicating if any blocks have been overflowed or not, for a set of transactions in the same scope. If none of the transaction blocks overflow the cache, then when a thread misses in the cache, a conflict check does not need to be performed by PTM for the miss. The conflict check is instead handled com-

pletely by the on-chip transactional memory system. Only when a transactional block (read or written by a running transaction) has been evicted does our PTM mechanism come into play.

Home and Shadow Pages

A key difference between our approach and the prior techniques is how we maintain the transactional information for the transaction blocks that have been evicted from the cache. A transaction block is a block of memory accessed by a transaction that is still executing.

For an evicted transaction block we need to maintain the following information in a data structure: (1) the speculative data for the block, if it has been written by a transaction, and (2) a list of all the transactions that either read from or wrote to the evicted transaction block, as required for conflict resolution.

We will now describe how we store the speculative data for a transactional block when it gets evicted from the transactional cache. We observe that, for a set of transactions that are currently executing, there can be only one transactional writer to an address at any instant of time (otherwise, a conflict would be detected and one of the two conflicting transactions would have been aborted). Therefore, all that we need for any physical page accessed by a transaction is an additional page that can hold a transactional version of data for the memory blocks in the page. We call the original physical page the *home page*, and the additional physical page allocated as the *shadow page*.

Figure III.5 shows an example of the PTM data structures used to maintain the unbounded transactional memory. On the left side of the figure, we show the page tables used to perform the traditional virtual to physical page translation. We also have another structure called the Shadow Page Table (SPT), which contains one entry for every physical page of memory, and is indexed with

the physical page number. In the SPT entry, we store the address of the allocated shadow page, the home page's address, and some additional information required to maintain the unbounded transactional states. There is a valid bit associated with the shadow page pointer, since not every SPT entry will have a shadow page allocated for it. Since the SPT is indexed by the physical page number, we can access information about the transactional memory block given its physical or virtual address. Given a physical address, we can directly index into the SPT. Given a virtual address, we can use the page table to get the physical address and then access the corresponding SPT entry.

When a page is allocated, its corresponding allocated physical page entry in the SPT is initialized and marked as valid. When a dirty transactional block is first evicted for a page used within a transaction, PTM allocates a shadow physical page, a pointer to it is stored in the SPT, and the shadow pointer is marked as valid for that SPT entry. For example purposes, we show in Figure III.5 an SPT entry for a physical page address "0x0000000" containing the shadow physical page address "0xFE03000". The corresponding speculative transactional block and the non-speculative block can then be kept track of in the two pages (home and shadow). Note, the physical shadow page that was allocated does not have a valid SPT entry. Only the home physical pages have valid SPT entries, which are marked as valid when the home physical pages are allocated. In addition, not all SPT entries have a valid (allocated) shadow page. If there are transaction blocks evicted from the cache that were only read (not written), then they may have an SPT entry without a shadow page allocated for it. In this case, the SPT entry serves the purpose of finding the transaction access information for the home page (what blocks were read, and by which transaction), which we describe later .

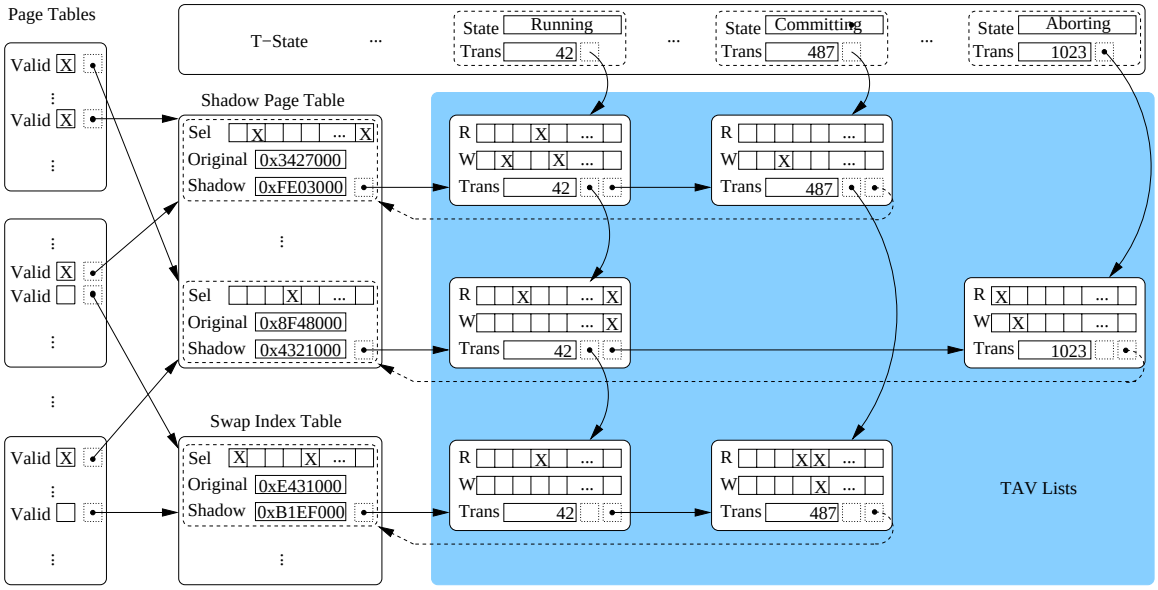

Figure III.5: PTM structures. Physical page numbers and swap file offsets are obtained from the page tables and used to index into the Shadow Page Table (SPT) and the Swap Index Table (SIT) respectively. An entry in the SPT and SIT tables for a page indicate the locations of the shadow page and contain a Selection Vector in which each bit indicates which of the two pages contain the committed version of a block in the page. An entry also points to a Transaction Access Vector (TAV) List, which contain one node per transaction that has accessed the list's page, but was not able to keep the accessed blocks in the cache. The nodes in a TAV list indicate the transactions in question and contain the Read and Write Vectors to mark the accessed blocks that do not stay in the transactions' cache. The T-State table is indexed by a transaction number and contains the state of each transaction. An entry in the T-State table links to a list of TAV nodes that were overflowed by the transaction.

Copy-PTM

Now that we have the shadow page, the question is where to store the speculative transaction blocks that have been evicted from the cache? One could have the policy where the speculative blocks are stored in the shadow page, and on commit they are copied back to the home page. We originally examined this design, but found the cost of commit to be higher than we desired. We hope to see many more commits than aborts when using transactions and we do not want to slow down the execution of transactions that are doing useful work. We therefore want to optimize the performance of committing, and start running the transactions in order if aborts are too frequent.

The first PTM approach we examine is called *Copy-PTM*. In this approach we copy the home block to the shadow page when a dirty transaction block overflows, and then store the speculative block in the home page. This policy enables fast commits, since the blocks that we want to commit are already in the home page. It requires that we make a copy of the non-speculative block from the home page to the shadow page when a dirty transaction block is evicted for the first time in a transaction. Then on abort we have to pay a penalty because we have to copy the non-speculative blocks, which were overwritten in the home page, back from the shadow page to the home page.

Select-PTM

The more aggressive solution we examine is to allow both the home and the shadow page to contain speculative and non-speculative blocks and use *Selection Vectors* to maintain them. We call this *Select-PTM*.

In Select-PTM, both speculative and non-speculative blocks are allowed to exist in either the home or the shadow page. We use a *Selection Vector* to indicate which of the two pages contain the non-speculative block and the

speculative block. The selection bit vector is stored along with the shadow page pointer in the SPT structure as shown in Figure III.5. Each bit in the selection vector represents a memory block in the page. We chose the size of the memory block to be the same as the cache block size of the outermost transactional cache in the processor, but our design does allow for larger or smaller memory blocks to be used.

A bit in the selection vector tells us which of the two pages, the home or the shadow page, contains the current committed data for the memory block for which the bit corresponds to. If a bit in the selection vector is set, then this means that the non-speculative data for that memory block resides in the shadow page and that the home page should be used for holding the speculative version and *vice versa*.

Whenever a transaction modifies a cache block and evicts it, the block is copied to the speculative location in memory, which is either the home or the shadow page depending upon the state of the bit in the selection vector. Similarly, while fetching data from memory we can determine where to find the committed and speculative copies based on the state of the bit in the selection vector.

When a dirty transaction block is evicted, we write the block to the speculative location. We write the speculative block to the home page if the bit is set, or to the shadow page if the bit is clear. When a transaction aborts, nothing needs to be done, since the bits in the selection vectors for the pages touched by the transaction are already pointing to the non-speculative blocks. On commit, however, we must go through the selection vectors and toggle the bit corresponding to the overflowed memory blocks that were written by that transaction, though hardware can accelerate this, which we describe later.

Trade-offs Between Copy-PTM and Select-PTM

The PTM structures required for both Copy-PTM and Select-PTM are the same structures shown in Figure III.5, except that Copy-PTM does not need the selection vector in each SPT entry.

In Select-PTM, the benefit of using a selection vector and allowing committed blocks to reside on either of the two pages (the home or the shadow page) is that it does not have to copy non-speculative blocks during eviction and abort, as described earlier for Copy-PTM. The downside to using the selection vector is that a non-speculative block can now reside in either the home or the shadow page, and we need to have an efficient way of finding the correct physical address to fetch the block, given the virtual address. The policy PTM enforces is that even when a block is fetched from a shadow page, the physical address seen by the cache hierarchy and the TLB structures is the home page physical address corresponding to that block. This allows Select-PTM to only have to perform TLB translation to the home page as in a conventional design. Then Select-PTM will monitor the block addresses at the memory controller to decide where to fetch the correct blocks from (the home or the shadow page). How this is done is described in Section III.D.2. Therefore, the advantage of the Copy-PTM approach is that, since the committed blocks are always on the home page, it does not have to deal with this address translation issue, and it does not have to maintain the selection vectors.

Conflict Detection using Transaction Access Vectors

In addition to keeping track of the speculative data for the overflowed cache blocks, we must also keep track of the information about the list of the transactions that read or write to an overflowed cache block. To accomplish this task, we maintain a Transaction Access Vector (TAV) data structure as shown

in Figure III.5. Each TAV node in the data structure is for a transaction and for a page that a transaction has overflowed. The TAV contains a read vector and a write vector for the page. Each bit in the read/write vectors corresponds to a cache block in the page and it tells us if the cache block was read or written by a transaction.

The read and the write bits are set when the blocks accessed within a transaction are evicted from the cache. For example, Figure III.5 shows that the transaction 42 read the 4th block and wrote the 2nd and 5th block in the virtual page with the physical home address "0x0000000" and the shadow page address "0xFE03000". When a read or write (executed in a transaction's code or even in the non-transactional code) misses the cache, PTM is consulted with the home page's physical address checking these read and write vectors to determine if there is a conflict. Note, we only need to check for conflicts in PTM if there is a live transaction and if a transaction has overflowed the cache.

All the TAV nodes corresponding to a transaction are linked together (vertical links in the Figure III.5). Given the transaction number, we can find all the TAV nodes for that transaction. The TAV nodes corresponding to a page are linked together (horizontal links in the Figure III.5). Thus, for a given TAV we can find its corresponding SPT entry. The same horizontal link is also used to find the TAV nodes of other transactions that have also accessed the same physical page, which enables us to determine the conflicting transactions.

TAV organization: Let us summarize the TAV data structure organization. An entry in the SPT structure contains a pointer to a linked list of these access vectors (transaction access vector (TAV) list). These are the horizontal linked lists in the Figure III.5 and the last node in the list points back to the SPT entry. Each node in the TAV list is for a transaction that had at least one overflowed block for that page in the past. A node in a TAV list contains a trans-

action's read and write access bit vector, where each read/write bit corresponds to an overflowed cache block in the page and tells us if the overflowed cache block was read or written by the transaction. In addition, each entry also contains a transaction identifier, constructed by the TM hardware, which enables us to determine the transaction to which the TAV read and write access vectors belong. A node in a TAV list is updated when a transactional cache block is evicted, and freed when the corresponding transaction either commits or aborts.

Conflict detection using TAV: If a read or write (executed in a transaction or in the non-transaction code) misses the cache, and there exists an overflowed block, PTM is consulted with the physical address to resolve any potential conflict. PTM uses the physical address to index into the SPT structure to get the pointer to the TAV list. Each node in the TAV list corresponds to a transaction that has overflowed a read or write to the page, and has to be examined to determine if there is a conflict. If the current memory operation that triggered a miss is a read, then there is a conflict if there exists a node in the TAV list with the write bit set for the accessed memory block and the transaction identifier is different from the current read's transaction identifier. Conflict detection for a transaction write is similar, and we detect a conflict if there exists a node in the TAV list with either the read or the write bit set for the accessed memory block, and the transaction identifier differs.

In Section II.G, we describe how information in the TAV list can be summarized into one vector and cached in a hardware structure to perform efficient conflict detection. These summary vectors are also used with the selection vector to determine which of the two physical pages to fetch from on a cache miss for Select-PTM.

Commit and Abort

To commit or abort a transaction, we use the vertical links shown in the Figure III.5. The head of the vertical list is maintained in the *T-State* structure and it also contains the transaction identifier along with its current status, which is atomically set to either *committing* or *aborting* before processing the TAV list.

Select-PTM: On commit, we traverse the vertical list for the committing transaction and free the nodes in the list. In addition, we update the selection vectors as needed. This is achieved while traversing each node in the vertical list, where we access the TAV node's corresponding SPT entry by following the horizontal list. We update the selection vector for that SPT entry if the committing transaction has overflowed any dirty block for the page corresponding to the SPT entry. On abort, we also have to traverse the vertical TAV list and free the TAV nodes. But, unlike what we did for commit, we do not have to update the selection vectors.

Copy-PTM: On commit and abort we traverse the vertical list and free the TAV nodes. For commit, we do not need to do any additional work, since there are no selection vectors. On abort however, we need to restore the original non-speculative blocks to the home page, for those overflow blocks that were written by the transaction.

Paging and the Freeing the Shadow Pages

Since the blocks representing a page are split across the home and shadow pages, we need to correctly deal with paging those pages in and out, as well as how to free the shadow pages.

Paging

To deal with the paging out of transaction pages, we actually have two tables, the *Shadow Page Table* (SPT) and the *Swap Index Table* (SIT). The first is indexed using the physical address (when the page is in main memory) and the second is indexed using the swap index number (when the page is swapped out to disk).

The swap index number is the number used by the operating system to keep track of the pages that are swapped out. It is equivalent to the physical page number. The difference between the two is that the swap index number refers to a location on the disk, but the physical page number refers to a location in the main memory. Thus, when a page is swapped out of the main memory to a location in the disk, the swap index number corresponding to that location is stored in place of the physical page number in the page table entry. When an application refers to a swapped-out page, the swap index number is used to locate the paged out data and swap the page back in to the main memory. The new location in main memory referred by a physical page number is stored in the page table entry.

In PTM, the shadow and the home page cannot be swapped out independent of each other. If one of the pages is swapped out, both pages have to be swapped out. The operating system does not consider the shadow pages to be candidates for swap out. The operating system only makes decisions about swapping out home pages. When a page is swapped out, if the page has a valid SPT entry, then it has to be copied to a SIT entry. The index for the SIT entry is the swap index number corresponding to the location in the disk that is allocated to hold the swapped-out home page. If there is a valid shadow page for the home page, then it is also swapped or garbage collected (see below). If swapped out, then its SIT shadow pointer is used to point to where the shadow page is stored

on disk. When a transaction page is swapped back in, the SIT entry is copied to the SPT entry corresponding to the newly allocated physical home page. If the SPT entry has a shadow page, it is also allocated a physical page, and its shadow pointer is updated in the home page's SPT entry.

Freeing Shadow Pages

Copy-PTM frees a shadow page when there are no more transactions using it, which is determined by the NULL TAV Link.

Similarly, for Select-PTM, a shadow page can be freed when there are no more transactions using it. That is, the page has only one version (committed version) for each memory block in the page. However, since the committed blocks can reside in both the home and the shadow page, we need to copy the contents from the shadow page back to the home page before we can free the shadow page.

We examined two different policies for freeing shadow pages for Select-PTM. One approach is to merge the home and the shadow pages together when the home page is swapped out by the operating system. To accomplish this, when a home page is swapped out, if it has a corresponding shadow page and there are currently no transactions using that page (determined by the NULL TAV link), then the operating system stores the valid blocks in the shadow page to the backing store location that is allocated for the home page. The SIT entry is updated to indicate that the page does not have a shadow page anymore and the selection vector is also cleared. This completes the process of freeing a shadow page.

Another approach to free a shadow page for Select-PTM is to lazily migrate the committed blocks to the home page. Whenever a non-speculative dirty block is written back to main memory, we can force it to be written back to the home page, even if the bit in the selection vector points to the shadow page.

After writing back the cache block, the bit in the selection vector is toggled to indicate that the committed copy is in the home page. This allows the memory blocks to be gradually merged back to the home page when they are read and written. Eventually, when the selection vector is completely clear (all the blocks are now in the home page), the shadow page can be freed.

Shared Memory Inter-Process Communication

Since the SPT entry (or SIT entry) and the TAV list are maintained for a physical page (or a swapped out page) rather than a virtual page, conflicts between transactions executing in two different processes accessing the same physical page can be detected. Thus, PTM supports shared memory inter-process communication.

III.D.2 Implementation

This section examines the hardware changes necessary to support PTM. We modify caches and the cache coherence protocol similar to other Hardware Transactional Memory proposals. Paging and swapping are changed in novel ways that the operating system needs to be aware of, and we also add a new Virtual Transactiona Supervisor hardware to the memory controller to cache the transactional state. One nice property of the PTM proposal is it does not require large changes outside of the VTS- for example converting a directory coherence protocol requires the addition of only single bit as described later.

Checkpointing the Registers

PTM does checkpointing at all levels of the memory hiearchy, including the registers at the transaction start. Checkpointing registers is fairly well understood because branch predictions recovers to checkpoint created at the branch

that has the speculation failure. For out-of-order machines, this is accomplish by saving a snapshot of the register renaming map [2, 47] at the checkpoint, instead of saving the entire architectural register state. Registers in the map checkpoint are protected that are no longer references are prevented from being recycled by the register renamer. Inorder processors, may save the entire register state in a single "flash" copy to create the checkpoint [30].

Transactional Cache

Each processor core is largely unaware of the memory controller's PTM hardware. All requests for cache blocks use the home page address. Each core can detect transaction conflicts within its cache through the existing cache coherence mechanism, and cache block versioning to stores speculative and non-speculative memory state [40]. Each cache line contains a valid bit, coherence state bits to support MOESI, a Transaction ID, and bits indicating if the transaction read (TR) or wrote (TW) the block.

When Everything Fits in the Cache

We now describe the operations of the transactional cache. Transactional reads and writes maintain TR and TW bit indicating the block is transactional, and overload the regular cache coherence protocol to detect transactional memory conflicts upon a cache miss. By separating the TR and TW bits from the MOESI bits, the MOESI protocol can essentially be ignorant of the transactional behavior, simplifying the non-transactional cache coherence. On transactional update, we perform memory versioning into a speculative and non-speculative block. We assume the cache is at least two way associative, and obtains a new block through the regular LRU process, making sure not to evict the non-speculative block. If ever either versions must be evicted, then its coun-

terpart is evicted as well.

Transactional cache access behavior

- **Cache Read** When a transactional program reads from the cache, it indexes into the cache and verifies that a valid cache block exists (non-I coherence state and correct address). On miss while fetching the data, the system performances a cache coherence snoop on the other processors' caches. With a valid block, it sets the TR bit to true, if not done already, and for cache reads it is unnecessary to duplicate the block. The cache read returns the data to the requesting CPU.

- **Cache Write** When a transactional program writes from the cache, it indexes into the cache and verifies that a writeable cache block exists (M or E coherence state and valid address). On a block miss or non-writeable coherence (upgrade) miss, the system performances a cache coherence snoop on the other processors' caches. Once the writable permission is available, if the TW bit has not been set already, it duplicates the block and sets the TW bit on one of them to be true. This creates the speculative and non-speculative versions. It then writes the data into the speculative block.

- **Cache Read Snoop** Performs snooping lookup on destination block. If the block's TW bit is marked then signal conflict. Otherwise move any dirty copies to shared (O state), and obtain a copy of block from the owner.

- **Cache Write Snoop** Performs snooping lookup on destination block. If the block has TW or TR bits marked then signal conflict detected. Otherwise, invalidate destination block, flushing dirty data if necessary.

A transaction may complete without overflowing its cache. When a dirty block commits and it has never overflowed the cache, no work needs to be done by PTM. The block is just marked as non-speculative, and at that point it is treated

as a normal cache block. Its non-speculative counterpart will be invalidated at commit. It will continue to reside in the processor core's cache until the cache sets overflow or another core requests the block. When a cache miss results in a conflict with another block in a cache, we use a Virtual Transaction Supervisor (VTS) to arbitrate which transaction to abort. The aborted transaction's cached data is invalidated in the cache.

VTS Caches

In order for PTM to provide efficient unbounded transactional memory, we provide hardware support to make the following tasks efficient:

- **Fast Conflict Detection** - When a transaction scope has overflowed the cache we need a way to quickly determine a violation when processing a cache miss. We therefore cache in the memory controller, the summary information for the transaction blocks that have been read and written for recently accessed pages.
- **Fast Commit and Abort** - We need to have the ability to quickly commit or abort a transaction, and to let future execution continue, while the overflow data structures used by the transaction are cleaned up.
- **Fast Selection Between Home and Shadow Page for Select-PTM** - We need to be able to quickly choose between the home and the shadow page when fetching a block from memory. To achieve this, the memory controller caches the information needed to correctly choose between the home and shadow page for recently accessed pages.

To provide the above functionality, PTM uses a Virtual Transaction Supervisor, which is shown in Figure III.6. The VTS is part of the memory controller for a snoopy architecture, and part of the directory controller for a

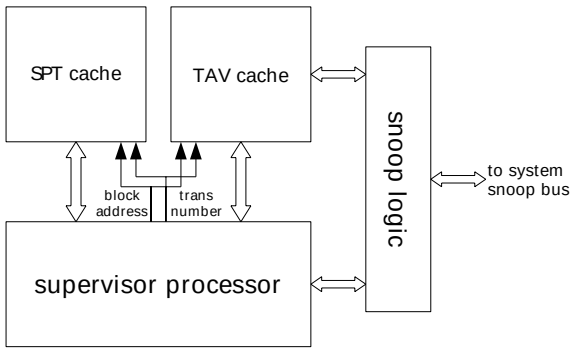

Figure III.6: The Virtual Transaction Supervisor (VTS) has a memory backed cache holding the SPT entries and the TAV nodes.

directory based system. VTS has two main caches. A cache of the shadow page table entries and a cache of the current transaction access vectors. We describe them as if they are updated on-demand, but performance improvements can be had by prefetching data into the caches.

Transaction Access Vector (TAV) Cache

The first cache is called the *Transaction Access Vector* (TAV) cache and is used to hold the nodes in the TAV lists in memory. An entry in the TAV cache corresponds to a TAV node shown in Figure III.5. The TAV cache entry contains the read and write transaction vectors for a page accessed by a transaction. The TAV cache is indexed by the physical page number, and is tagged by the physical page number and the transaction ID. This allows multiple TAV nodes for the same physical page, corresponding to different transactions, to be stored in the cache at the same time. Indexing by the physical page allows PTM to quickly find all of the cached TAV nodes for that page.

The TAV cache is an important component in the PTM architecture for providing fast conflict detection. When there is a cache miss, the resulting memory request may need to determine exactly which transactions were prior

readers or which transaction was a prior writer to the block. In this case, if the TAV nodes for the page of the block are found in the TAV cache, the read and write vectors for the page can be quickly examined to determine the conflicting transactions (if there are any).

When a TAV cache entry is evicted and the access vectors have been updated (the entry is dirty), the access vectors need to be written back to their corresponding TAV entries in memory.

Shadow Page Table (SPT) Cache

The second cache structure, called the *Shadow Page Table* cache, is used to cache the entries in the SPT structure, which was described in Section III.D.1. The SPT cache is indexed by the physical page number, and is used to quickly determine conflicts.

When there are overflowed transactions being executed, we allocate an SPT cache entry for every non-transactional page and home page accessed. This is needed because non-transactional cache misses, which are executing while there are evicted transactional blocks, still need to be checked for conflicts. For non-transactional pages, the SPT cache entry allocated for it is used to quickly identify this.

The contents of an SPT cache entry is shown in Figure III.7. An SPT cache entry contains the shadow page number (if there is a valid one). In addition, it contains a write summary vector and a read summary vector. The write summary bit vector for a page is an OR of all the transaction write access vectors that exist in the TAV list for the page. This provides immediate identification for a cache block that a transaction has speculatively overflowed that block. The read summary vector is a single bit vector where each bit indicates if there has been at least one overflow transaction read for that block.

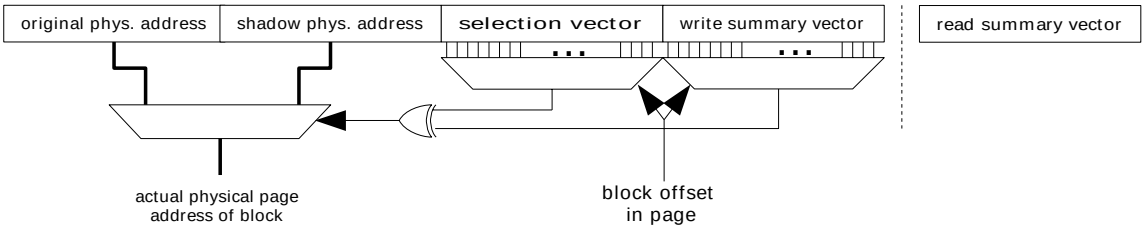

Figure III.7: SPT cache entry. The SPT cache entry stores the selection vector, the write summary vector, and the read summary vector for a page.

For Select-PTM, the SPT entry also has the selection vector as shown in Figure III.5. When an SPT cache entry is evicted, the corresponding selection vector in memory is updated if the SPT cache's selection vector is dirty. The SPT cache entry for Copy-PTM is the same as Select-PTM, but without the selection vector.

The SPT cache stores the information for the most recently accessed pages. A miss in the SPT cache requires the VTS to lookup the shadow page table to find the SPT entry, calculate the write and read summary vector from the TAV list, and then update the SPT cache. While the read and write vectors are calculated from the transaction's read and write access vectors for that page, TAV cache entries are created (if they do not exist) for each TAV node corresponding to that page.

Cache Eviction

When a cache block read/written by a transaction is evicted, the VTS takes action in response to the coherence message triggered as a result of the eviction. The coherence message will contain the physical address of the home page and is also piggy-backed with the transaction identifier. When a block is evicted, we do not need to check for a violation. We only need to check for a violation for the read or write cache miss. The following actions need to be taken

on eviction.

When an *unmodified block* is evicted in the normal MOESI protocol, there is no need to generate a coherence message, but in our case, when a cache block read by a transaction is evicted it has to generate a coherence message to inform VTS to keep track of the overflow information. However, the data block is not written back because the cache block was not modified. When the VTS receives the coherence message for the unmodified transactional block, it will update the read transaction access vector in the TAV cache corresponding to the transaction that accessed the evicted cache block. Also, the read summary vector in the SPT cache for the physical page of the cache block is also updated. Note, when an unmodified non-transactional block is evicted, no coherence message is sent.

When a *modified transaction block* is evicted, we write the transaction access vector in the TAV cache and update the write summary vector in the SPT cache. If a shadow page has not been allocated for the home page, then one is allocated at this time. The modified cache block then needs to be written to the page that is supposed to hold the speculative version. For Select-PTM, the selection vector indicates which page (home or shadow) to write the speculative block to, and the write is done to the speculative location. For Copy-PTM, the block is always written to the home page. For Copy-PTM, we need to determine when we need to copy the non-speculative block to the shadow page on eviction. This is done by checking the write summary vector for the modified block being evicted. If the bit is not set, then this is the first modified overflow of that block, so we first copy the non-speculative block to the shadow page. We can then write the evicted block to the home page and set the write summary vector bit. If the bit is set, and there is no conflict, then we do not have to perform any copy, and the evicted block is written to the home page.

When a *modified non-transaction block* is evicted, we always write the block to the home page for Copy-PTM, and we do not need to do any SPT cache lookup. For Select-PTM, we first need to perform a SPT cache lookup, and use the selection vector to determine which page to write the block to, which is the non-speculative location.

Cache Miss

There are two operations that need to be performed on a cache miss. The first operation identifies from which of the two pages we need to fetch the data to serve the cache miss. The second operation detects any potential conflict. We initiate the fetch for the data block from memory in parallel with the conflict resolution and hold back the coherence reply with the data until the conflict is resolved.

Finding the Block to Fetch on a Miss

To fetch a block in Copy-PTM, we always fetch the block from the home page.

For Select-PTM, on a miss we need to look up the selection vector and write summary vector in the SPT cache. We XOR the bit in the write summary vector and the bit in the selection vector for the current cache block request, and the resulting bit value determines the page (home or shadow) we want to read the block from. This logic is shown in Figure III.7.

Conflict detection

Read Miss: If the memory access is a read to a memory block, then there is a conflict only if there exists an uncommitted transaction that has modified the memory block (RAW conflict). To determine this, first we examine the

bit in the write summary vector that corresponds to the memory block being accessed. If the bit is not set, then there is no conflict. If the bit is set, then there are two possible cases. Either the transaction that is currently accessing the block has itself modified the memory block in the past, or the block has been modified by another transaction. There exists a conflict only in the latter case. To determine which case it is, we look up the block's physical address with the current transaction ID in the TAV cache. If there is a match, then we check to see if the current transaction is the owner of the write. If so, then there is no conflict. If not, there is a conflict and we find the conflicting transaction. If we get a miss in the TAV cache, the VTS has to perform a hardware walk on the TAV list, starting from the shadow page table entry, to find out the conflicting transaction, and the TAV structures found are put into the TAV cache.

We assume MOESI protocol. In PTM, a transactional read miss request to a block that has already been overflown by a different transaction is not granted exclusive permission even if there are no sharers in the system (that is, no processor in the system has read permission). This is required because the transaction that gets the block might later write to it and at that time we have to make sure to resolve any potential conflict that may exist with that write. However, if there are no transactional read overflows to the block and if there are no other sharers in the system, then the read miss request can be granted exclusive permission.

Write Miss: If the memory access is a write to a memory block, then there is a conflict if there exists an uncommitted transaction that had read (WAR conflict) or written the memory block (WAW conflict). An SPT cache lookup is performed to examine the write and read summary bit vectors. If the write summary bit is not set, and if the read summary bit is not set, then we know there is no conflict.

If the write summary bit is set, we need to lookup the write access vector in the TAV cache to see who the writer was. If the TAV write vector shows that the same transaction was the prior overflowed writer, then there is no conflict. If not, then we know there is a WAW conflict, and one of the transactions must be aborted.

If the write summary bit is not set, but the read summary bit is set, then we look through the TAV list to see who the readers are. If the current transaction is the only reader, then there is no conflict, otherwise there is a WAR conflict, and one of the transactions has to be aborted.

Arbitration

When conflicts are detected, the oldest transaction wins the arbitration causing the younger conflicting transactions to abort, thereby guaranteeing forward progress, as any long waiting thread eventually becomes the oldest. Unique transaction identifiers generated sequentially at the transaction start allows us to determine the age of the transaction. This also supports ordered transactions described in Section III.B.5, by assigning the identifiers to match the program defined ordering. When a transaction is aborted and restarted, it maintains the transaction identifier that was originally assigned to it.

Commit and Abort

On commit, all of the cache blocks with the transaction ID are specified as no longer being speculative, and the transaction ID is cleared. On abort, all of the cache blocks with the transaction ID that are dirty are invalidated. Those that are not dirty just have their transaction ID cleared.

To process the PTM state on commit or abort, the VTS will first atomically change the status of the transaction in the T-State structure shown in the

Figure III.5. This is referred to as the logical commit/abort by VTM [73]. Once the transaction has been logically committed or aborted, the thread can continue its execution. The TAVs of the transaction are lazily freed on commit and abort. Before freeing a TAV node, we update the read and write summary vectors in the SPT cache as necessary. During this lazy commit, if another transaction accesses a "not-yet-committed" memory block (in cache or in main memory) it sees that there might be a conflict. However, while resolving the conflict, PTM knows that the conflicting transaction ID has already committed, when it looks up cached T-State structure in VTS. The transaction that has the outstanding miss is made to wait until the commit for that page finishes. After the commit for the conflicting transaction is over, the stalled transaction can continue its execution with the committed data block. After abort or commit, if the shadow page does not contain any committed blocks, then the shadow page is put on the free list and the SPT entry is updated.

For Select-PTM, as the TAV structures are committed for a transaction, the corresponding pages in the SPT cache and TAV cache are processed to correctly update the selection vector in the cache (if there is an SPT cache hit) and in memory (if there is a SPT cache miss). On abort, the selection vectors do not need to be update.

In the case of Copy-PTM, on abort we need to restore the original cache blocks that were overwritten by the transaction in the home page from the shadow page. We walk the TAV list and use the write vector to determine which blocks to restore from the shadow page to the home page. On commit, no data needs to be copied.

VTS Implementation for Snoopy-based and Directory-based Systems

To implement VTS as part of a snoopy architecture we integrate VTS into the memory controller. This is straightforward for a centralized controller, but it is also possible if there are multiple memory controllers. For multiple memory controllers, if the memory controllers are associated with particular regions of physical memory, this means a partitioned and distributed SPT cache and TAV cache. If instead the memory controllers are associated with particular cores rather than memory regions, this means distributed SPT and TAV caches with a dedicated coherence network among them.

For a directory protocol, the VTS would be distributed among the directories and implemented in the directory controller. Essentially, the SPT cache and the TAV cache in a directory will be caching the information corresponding to the physical pages maintained by that directory. The directory based VTS implementation requires some additional hardware support to perform arbitration to resolve conflicts. The additional support is required to ensure that all commits and aborts will be serialized correctly to guarantee atomic commit and aborts. Each directory entry has an overflow bit, which is set when the corresponding memory block overflows. Cache overflow due to a cache miss triggers a coherence request. When a cache miss coherence message reaches a directory, and if the overflow bit is set, the VTS associated with the directory is consulted to resolve conflicts. Thus, selecting between the home and the shadow page and resolving the conflicts can all be done as before. In addition, the shadow page for a home page is allocated so that they reside in the same directory controller.

Processing cache overflows and non-conflicting cache misses does not involve the supervisor processor, unless there is a miss in the SPT cache or TAV cache. If that is the case, then the supervisor processor needs to fill in the entries. The only other main functionality the supervisor processor does is to perform

the TAV list walks on commit or abort. For snoopy and directory, we make sure that all of the TAV entries for a transaction are to the same memory/directory controller. An issue to keep in mind here is that the supervisor processor needs to have low enough occupancy to not become a bottleneck.

Efficient Context Switching

Context switches can be handled by just forcing an overflow of all the cache blocks read/written by a transaction. We assume physically indexed caches. Prior schemes like VTM [73] require the ability to translate the physical address to the virtual address as their overflow structures are virtually indexed. In comparison, PTM can update SPT entries and TAV entries using just the physical address.

On context switches, we avoid overflowing the cache blocks by tagging the transactional cache blocks with the transaction identifiers. In this case, the normal cache coherency conflict detection mechanism will be able to identify conflicts with the cache blocks that were not overflowed when the transaction was context switched out.

When a transaction begins, PTM takes a checkpoint of the architectural register states in the processor so that on an abort they can be restored. To support context switches for a transaction, we save and restore the transaction's checkpointed register state. In PTM, the T-State, which contains an entry for each transaction is used to save the checkpointed register state of a transaction when it is context switched out.

III.E Evaluation

This section evaluates the performance of PTM, demonstrating that it efficiently supports virtual transactional systems without incurring high overhead.

III.E.1 Simulation Platform

We modeled a CMP system using Virtutech Simics [52] based on Enterprise machines running RedHat Linux 7.3, and extended the model to simulate PTM and VTM. The entire system has 4 nodes, each with two levels of private cache. The L1 cache is 16KB direct-mapped with a 1-cycle latency, while the L2 cache is 256 KB 4-way set associative with 6-cycle latency. Coherency is maintained at the L2 cache using a snoopy-based MOESI protocol. The augmented L2 cache blocks contain transactional read and write bits that are used to track transactional read and write accesses similar to prior work [33, 57]. In addition, each cache block contains a transaction ID, a valid bit and the bits to implement MOESI protocol. Each node in the system is a single-issue in-order pipeline. We simulate a 512 entry fully associative TLB where each page is of size 4 KB.

We added features to Simics to support a transactional memory system. In particular, we modified the Simics instruction decoder to recognize the instructions *Begin* and *End*, which are used in the program to specify the begin and end of the transactions respectively. Our simulation of PTM and VTM assumes that the processor has a fast register checkpointing mechanism.

The Chip-Multiprocessor (CMP) memory hierarchy is supported by a high speed on-chip bus and a low speed main memory bus. We simulate a high speed on-chip bus connecting the four CPUs and the on-chip memory controller with a minimum round-trip latency of 20 cycles. The memory controller contains the PTM caches and the ancillary hardware. In the MOESI protocol that we model, a cache miss request can be sourced from other caches containing a valid copy instead of having to access the much slower external memory. We assume access to main memory has a minimum latency of 200 cycles, but up to three requests can be pipelined simultaneously.

PTM Modeling

The PTM hardware in the memory controller handles transactional coherence requests using the SPT cache and TAV cache to speed up the process. We simulate a 512 entry SPT cache and 2048 entry TAV cache. Both are fully associative. A miss requires that we access the shadow page table in memory. From the shadow page table we can get access to all of the TAV structures for that page. To ensure fairness, in our simulation of VTM, we use an XADC [73] of capacity equal to the combined capacities of the SPT and TAV cache. The victim cache, and the hardware resources to implement it, are used only for the Victim-VTM results. Those extra hardware resources not used by PTM.

VTM Modeling

In order to evaluate the performance of PTM, we constructed a VTM model based on the description in [73]. We use the same in-cache hardware transactional memory model for both PTM and VTM. This is a more optimistic model for VTM than that featured in [73]. We assume the presence of transaction IDs in the cache, which can be used to avoid having to flush all transactional data on every context switch. We also assume for the VTM model that the XF counting Bloom filter has been implemented in dedicated hardware. We model an XF with 1.6 million entries. We also assume an XADC to cache the meta-data for the overflowed blocks.

When checking for conflicts, if all of the block's XADT entries have their meta-data cached in the XADC, then the conflict resolution is done in the time it takes to do the cache lookups. If there is an XADC miss, it requires a reconstruction of meta-data via traversal of the XADT, similar to creating a SPT cache entry from our TAV structures for PTM. When walking the XADT for commit or abort, we assume that each XADT entry lookup requires a single

main memory access, and that the number of memory accesses is equal to the number of XADT entries traversed.

VTM, like PTM, supports a lazy commit, changing the status of a transaction atomically via an atomic memory operation on the transaction's status word XSW and updating all other data and structures lazily. However, since it has buffered all overflowed speculative values in the XADT, VTM must actually copy the speculative data to the original memory location on commit. This occupies bus resources, even when doing the commit lazily. As bus contention in our memory model leads to performance degradation, we also consider adding data buffering to the XADC to hold the speculative and non-speculative block in addition to the meta-data. Because this secondary cache acts like a victim cache, we refer to this variant as Victim-VTM (VC-VTM) in our results, with the baseline VTM labeled simply as VTM. Blocks in the victim cache are marked as being committed instantly, and later written back to memory when evicted from the cache. Currently executing transactions can then use the blocks found in the victim cache, instead of having to wait for them to be committed. We found this to significantly reduce the commit delay penalty for VTM.

III.E.2 Characterizing Transactional Applications

We studied the behavior of the transactional memory regions by using Splash-2 [93] programs. We first removed all the locks from the programs. We then parallelized each program using transactions. We made use of two instructions, `Begin` and `End`, which specify the begin and end of a transaction respectively. To parallelize the code, we focused on creating critical transaction regions similar to how the average programmer might go about doing this. We wrapped each loop body with a transaction, so that each iteration of the loop can be executed in parallel. If there are loop carried dependencies, we used ordered

Table III.4: Transactional memory execution behavior for loop regions in the SPLASH-2 programs. The entries in the table are organized in three sets. The first set describes the transactional behavior of the applications, the second set describes the system behavior, and the third set provides information about the memory footprint of the transactions.

Apps	Transactions		System		Memory				
	commit	abort	exception	context-switch	pages	pg-x-wr	conservative	ideal	mop/evict
fft	34	5	595	52	1041	551	52.9%	9.5%	87.5
lu	656	0	17754	1079	2311	2130	92.2%	3.6%	95.3
radix	70	17	615	116	771	629	81.6%	2.0%	246.3
ocean	877	282	7417	1421	14966	6769	45.2%	0.2%	15.8

transactions to enforce correct dependencies.

Various program characteristics relevant to PTM are presented in Table III.4. The second column in the table indicates the number of committed transactions per application, and the third column presents the number of aborted transactions. Both these results demonstrate the significant amount of transactional activity in our benchmarks. We present the results for system effects in the fourth and fifth column of Table III.4, listing the number of exceptions and context switches seen by the program – the fact these system effects exist is a motivation for our proposal's support for virtualizing unbounded transactions.

The sixth column, titled "pages", presents the memory footprint in terms of the number of unique pages accessed during the course of entire program execution by both transactional code as well as non-transactional code. This does not include the shadow pages used. The seventh column "pg-x-wr" shows the total number of unique pages updated by just the transactional writes.

We estimate the worst case upper bound on additional pages allocated due to allocation of the shadow pages. The upper bound is shown in column eight with the title "conservative". The upper bound is computed as the fraction

of transaction's footprint (shown in column six) and the entire program execution's footprint (shown in column seven). The column "ideal" shows the percent increase in the number of pages if all of the shadow pages created for a transaction were instantaneously committed or garbage collected when a transaction commits. To calculate this number we determine the average number of pages that are live at any instant for the transactions. We treat this number to be the additional number of shadow pages that are live at any instant and calculate the increase in page overhead accordingly.

The last column "mop/evict" in the table describes the frequency of cache block evictions. The results are shown in terms of how many memory operations occur between evictions. For example, `radix` shows that it evicts a block every 246 memory operations. This is one measure of how much work the overflow transactional memory has to perform. In the worst case (`ocean`), we see that a cache block is evicted for every 16 memory operations.

III.E.3 PTM Performance Comparisons

To determine the usefulness of the proposed PTM, we simulated the performance of PTM comparing it against the prior technique VTM and lock-based multi-threaded execution. Figure III.8 shows the speedup over a single thread of execution for five SPLASH-2 benchmarks. In this and our other figures, we abbreviate Select-PTM as *Sel-PTM*. We first show the speedup of using the default p-thread locks. Using fine grain locks we can achieve a speedup of 103% on average. This approach does not have the overhead of the transactional execution, speculative aborts, and the overhead of buffering the overflowed blocks, although lock-based execution lacks the deadlock-free execution guarantees that transactional memories provide.

The baseline VTM shows decent speedups for three of the benchmarks,

but we do not see any speedup for VTM on `fft` and `ocean`, due to the overhead of commits. In comparison, if a victim cache is used with the XADC to hold the recently evicted transaction blocks, we achieve speedup for all benchmarks over single threaded execution. This is because currently executing transactions can access the overflowed but not-yet-committed blocks from the victim cache, while those blocks are being committed. Thus, for VC-VTM we see an average speedup of 26% reflecting the benefits of overlapping execution with physical commit to reduce the commit cost. We also investigated performing copying on abort, which found 24% speedup on average. Modifying VTM slightly would obtain a significant improvement.

Our results for Copy-PTM show an average speedup of 68% and for Select-PTM we observe 75% speedup. The difference between the two is directly attributable to the additional overhead Copy-PTM incurs for copying blocks to the home block on evictions and restoring them on aborts. Note that we do not use a victim cache for the PTM results. One of the main differences between VTM and Copy-PTM is that Copy-PTM incurs a penalty on abort, whereas VTM incurs a penalty on commit. In the future we plan to compare against a variant of VTM that does in-place speculative updates, so that the main penalty is due to abort and not commit. We expect this approach to perform closer to Copy-PTM.

Since coherence is done at the cache block granularity, there can be false conflicts detected due to false sharing. This can lead to unnecessary aborts, which incur extra run-time overhead [54]. It has been shown that this overhead can be reduced by for tracking conflicts at the word granularity [34, 33].

For the results we discussed thus far, we used a cache block of size 64 bytes. Let us say a transaction read/wrote to one of the words in the 64 bytes, and then it was followed by another transaction that tried to write to a different

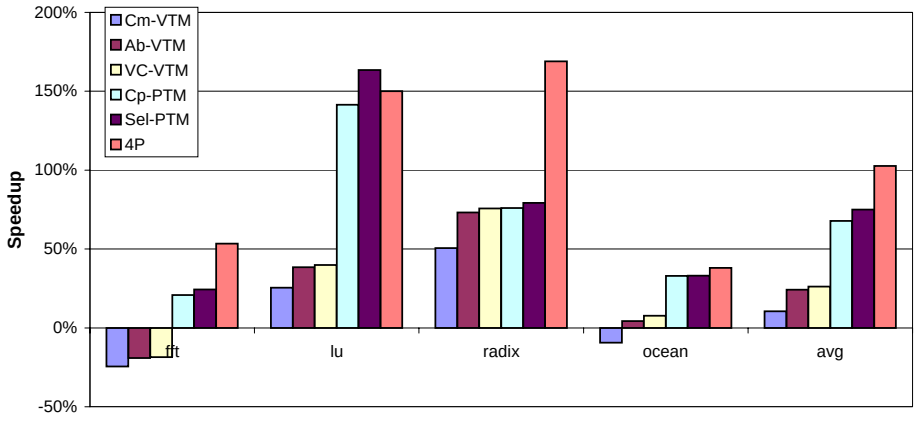

Figure III.8: Comparing TM speedup for lock-based multi-threading, (base) VTM, Victim-Cache VTM, Copy-PTM and Select-PTM. Speedup is over single threaded execution.

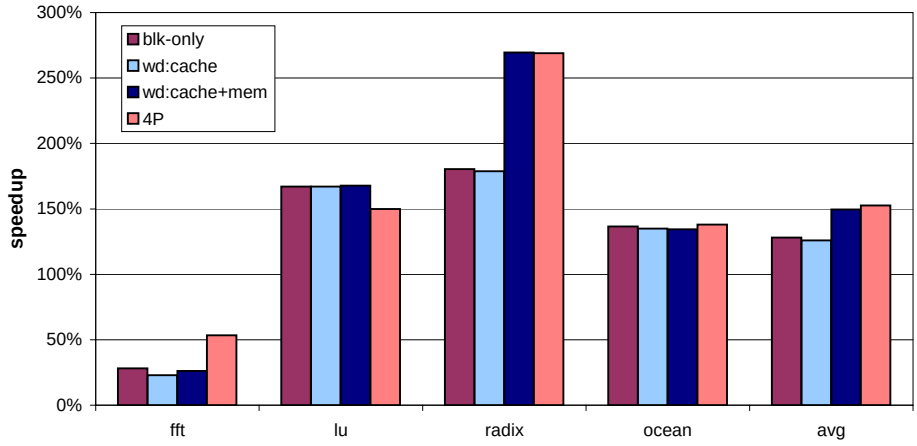

Figure III.9: Advantage of conflict detection at the word granularity.

word in the same 64 byte cache block. Clearly, there was no conflict. However, our conflict detection mechanism based on block sized coherence messages and PTM data structures would detect a false conflict and unnecessarily abort one of the transactions, because the conflict mechanism operates at the cache block granularity.

Figure III.9 shows the performance of modeling conflicts at the word granularity compared to Select-PTM. Results are compared against only using block granularity `blk-only`, and using p-threads locks. The first approach we examine, `wd:cache`, performs cache coherence at the word granularity, but still keeps track of transactional information for overflowed blocks at the block granularity (64 bytes). As a result, this leads to more coherence traffic, which we modeled, and also adds additional complexity to a directory system. This resulted in only minor speedups, because evicting a block with multiple writers would cause an abort, since the overflowed PTM structures would only kept track of one writer per block.

We then examined keeping track of transactional information even for the overflowed blocks in PTM at the word granularity, which is `wd:cache+mem` in Figure III.9.For `radix`, this resulted in 169% speedup over single threaded execution, which is a significant improvement over 80% speedup from tracking all conflicts at the block level.

While the problem of false conflicts due to detection granularity is highly benchmark dependent and not universal, it does affect programs like `radix` dramatically. Techniques explored in prior work should help reduce false conflicts, either by changing data structure alignments [86] via the compiler, or allowing more than one processor to own sub-partitions of the cache block [23].

III.F Conclusion

With the advent of multi-cores, extracting task level parallelism is going to be crucial. To meet this goal, transactions can help common programmers to write multi-threaded programs. Transactions Memories can eliminate introduction of deadlocks and livelocks through synchronization. Ordered transactions can eliminate non-determinism (general-races) from multi-threaded programs. However, support for unbounded transactions is crucial to develop a good transactional programming model.

We proposed a system design called PTM that extends existing virtual memory support to support unbounded transactions. In PTM, when a transactionally modified cache block is evicted, we allocate a shadow page, which can be used to hold the speculative block. In addition, we aggregate and maintain all of the transactional information on a page-level granularity. The PTM structures are integrated with the virtual memory system, allowing direct access to the transactional data for a page with both the virtual and physical address of the page.

The first approach we examined is Copy-PTM, in which on a transactional dirty block overflow, a copy of non-speculative block is first backed up in the shadow page. On commit, the backed up copy can be discarded, but on abort it has to be restored in the home page. This allows commits to be fast, but aborts can be slow. We optimized this design in Select-PTM, where the two versions of data are allowed to be spread across the home and the shadow pages. To determine which of the two pages contain the block to be fetched, we used a selection bit vector. Select-PTM is efficient for performing both commit and abort operations, as it does not have to physically copy the data between the two pages. Also, on dirty block eviction the non-speculative data need not be backed up.

III.G Acknowledgement

Sections III.D, and III.E contain material to appear in "Unbounded Page-Based Transactional Memory", in *Proceedings of 12th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS XII)*, W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M.Van Biesbrouck, G. Pokam, O. Colavin and B. Calder. The dissertation author was the primary investigator and author of this paper.

IV

Conclusion and Future Work

Computing historically has taken advantage of an abundance of fast transistors. This enabled incredible growth in the speed and performance of CPU's, and made computers so cheap that they are pervasive and highly interconnected. Such a rich computational "environment" makes tempting targets for hackers to exploit, and by using software bugs, hackers are able to take over the computer for "fun and profit"[1]. The first part of our work attempts to thwart these attacks by reducing or eliminating the vulnerability of these bugs.

We consider two well known software checks known to defeat large classes of Internet attacks- bounds checking and dangling pointer checks. Their primary disadvantage are their run-time performance slowdown. We study the overheads caused by these software checks through CPU performance counters and microarchitectural simulators. These reveal that the primary causes of the slowdown are: (1) instructions executed to manipulate meta-data (2) cache miss effects accessing meta-data and (3) branch misprediction effects due to reduced capacity in the predictor. We propose a set of hardware and software techniques to reduce this overhead.

These are:

[1] Comes from the title of [1] that proliferated the technique for stack overflow.

- Folding together instruction sequences into a single check instruction using a dedicated instruction. One example is the x86 bounds instruction that hitherto has not been investigated.
- Discovering if pointers and array references are dangerous by propagating an externally-tainted property through the scalar and pointer assignment network. Tainted references implies that its buffer data may possibly be written from an external source, and therefore be considered dangerous. These should be checked, while the rest do not need checks.
- Reduce the meta-data copying overhead of pointer-meta-data by moving it to a shared object and accessing it through an extra level of indirection. We use hardware to hide the cost of the indirection.

The future of computing will receive an abundance of relatively slower transistors. In response, microprocessor designers are building multi-core Chip-Multi-Processor (CMP) systems that minimize communication between cores, rather than trying to scale up superscalar processors. As these CMP's scale, their software does not because they are stymied by conservative serialization, deadlock or priority inversion from lock based synchronization. Transactional Memories provides a favorable alternative. It's obstruction-free synchronization between transactional regions prevents data-races yet allows greater concurrency when there is no memory aliasing. A special class of Transactional Memories called Ordered TM can enhance the safety of concurrent programs desiring ordered serialization when there is a data conflict. For example OpenMP compilers are oblivious to loop carried dependencies, but with transactions concurrent loop iterations with loop carried dependencies can be made to serialize properly. Our Page-based Transactional Memory (PTM) work enhances the efficacy of Transactional Memory by making a PTM thread like any other except that has additional safety properties. A PTM transactional thread is unbounded in

speculative memory capacity and may be context switched or paged out in the middle of a transaction. PTM modifies the virtual memory system to support using an extra shadow page for overflowed speculative state, and the memory controller hardware to select between the two pages and detect conflicts. We also provide the first performance measurements of the earlier Virtual Transactional Memory (VTM) proposal.

IV.A Future Direction

Staring into the future its interesting to observe academic interest for both topics. While hardware enhancements for software safety is consistently active in the research community with session on security or debug at all the recent computer architecture conferences, transactional memory has exploded. Groups from the programming languages and hardware community in both academia and industry are rapidly investigating and publishing work on Transactional Memory, with many new possible research directions being explored and many new branches being opened up all the time. There have been workshops dedicated to Transactional Memories[2], and the number of Transactional Memory papers for 2005 has increased to 27 over the previous year of 14.

Here we present some ideas for future research in both software safety check and Transactional Memories.

IV.A.1 Software Safety Checks

While we feel the performance of our hardware acceleration of software checks makes it very practical and deployable, the primary limitation of our approach is its inability to check library code and system calls. Binary translation techniques like that used for Operating system virtualization [63] suggest a way

[2]Synchronization and Concurrency in Object Oriented Languages (SCOOL) 2005, and Workshop on Transactional System 2005

of providing enhanced capabilities with backwards compatibility throughout the runtime of the program. Newsome and Song [62] used this approach to track external Tainted sources but they found a five times slowdown. However another faster approach might be to encrypt data through ISA hardware like what Tuck et.al. [88] with significantly lower overhead. We would use the binary translation to handle retrofitting encrypted pointers into existing binaries.

Another path to investigate is the well known solution of using managed languages such as Java that eliminate explicit pointers and all the possible pointer bugs. The main hindrance is its lack of deterministic performance, and this issue will provide ample opportunity for research.

IV.A.2 Transactional Memory

We feel that our Page-based Transactional Memories provides a complete solution to providing virtual memory support for Transactional Memory. Of course our work already suggests some directions for improvements. First, one should eliminate false conflicts that is a problem with PTM and any other Transactional Memory or Thread-Level-Speculation system. Earlier we noted that false conflicts are caused by the granularity of detection (block) being much larger than the size of speculation (word typically). Other proposals [30, 54] have investigated this as well, so there is basis to start from. Second, it would be interesting to see if the PTM technique could be extended to provide more than one speculative version of memory so as to support TCC TM [30] or TLS [81] that requires them.

In the longer term, we feel that different programming models and perhaps their accompanying hardware support would be a very productive area. For example the OpenMP compiler can partition loops to execute concurrently although not always safely. McDonald et al. [54] recognized that Transactional

Memories can enable safe OpenMP partitioning, though this was mentioned in passing. A deeper more thorough investigation may turn up more possible issues with the interaction between TM and OpenMP.

A second long term area of interest is realizing that general speculative memory versioning and conflict detection hardware may allow novel memory models along with new programming methods. We propose one of the ideas. We can modify the conflict detection on PTM hardware with transactional ordering to implement a completely concurrent memory model which we call Phase Update Memories (PUM). Memory updates occur in phases where updates in a given phase appear simultaneously, and once a phase completes the next phase begins. Consequently unlike transactions each thread's PUM phase update cannot be serialized to create an equivalent execution on a single thread. PUM updates are similar to incrementing a clock in hardware that simultaneously exposes state on all sequential elements. Taking the hardware analog to software, PUM programming model would probably appear similar to hardware CAD languages like Verilog or VHDL, and the obvious application for PUM would be to execute those languages more efficiently.

Bibliography

[1] Aleph One/E. Levy. Smashing the stack for fun and profit. *Phrack*, 7(49), Nov. 1996.

[2] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[3] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[4] A. Anisimov. Defeating microsoft windows xp sp2 heap protection and dep bypass. http://www.maxpatrol.com/defeating-xpsp2-heap-protection.pdf.

[5] T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *Symposium on Programming Language Design and Implementation*, pages 290–301, June 1994.

[6] Avantgarde. Time to live on the network. http://www.avantgarde.com/xxxxttln.pdf.

[7] D. Avots, M. Dalton, V. Livshits, and M. Lam. Improving software security with a c pointer analysis. In *Proceedings of the 27th International Conference on software Engineering*, May 2005.

[8] B. Beizer. *Software testing techniques*. Van Nostrand Reinhold Co. New York, NY, USA, 1990.

[9] M. Blasgen, M. Astrahan, D. Chamberlin, J. Gray, W. King, B. Lindsay, R. Lorie, J. Mehl, T. Price, G. Putzolu, P. Sellinger, D. Slutz, H. Strong, I. Traiger, B. Wade, R. Yost, and and. System R: an architectural overview. *IBM Systems Journal*, 20(1):41–62, 1981.

[10] C. Blundell, E. Lewis, and M. Martin. Deconstructing transactional semantics: The subtleties of atomicit. In *4th Annual Workshop on Duplicating, Deconstructing and Debunking*, New York, NY, USA, June 2005. ACM Press.

[11] R. Bodik, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 321–333, 2000.

[12] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[13] G. Candea. Enemies of dependability I: Software. Stanford CS Lecture Notes: CS444a, Fall 2003.

[14] B. Carlstrom, J. Chung, A. McDonald, H. Chafi, C. Kozyrakis, and K. Olukotun. The Atomos Transactional Programming Language. *ACM Conference on Programming Language Design and Implementation*, 2006.

[15] A. Chang and M. Mergen. 801 storage: Architecture and programming. *ACM Transactions on Computer Systems*, 6(1):28–50, 1988.

[16] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.

[17] M. Corliss, E. Lewis, and A. Roth. Dise: A programmable macro engine for customizing applications. In *30th Annual International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[18] M. Corliss, E. Lewis, and A. Roth. Low-overhead debugging via flexible dynamic instrumentation. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA05)*, San Francisco, CA, Feb. 2005.

[19] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium, Washington DC*, August 2003.

[20] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, Jan. 1998.

[21] J. Crandall and F. Chong. Minos: Control data attack prevention orthogonal to memory model. In *37th International Symposium on Microarchitecture*, Dec. 2004.

[22] M. Dubash. Moore's law is dead, says Gordon Moore. *Techworld.com*, 13 April 2005. http://www.techworld.com/opsys/index.cfm?NewsID=3477.

[23] C. Dubnicki and T. LeBlanc. Adjustable block size coherent caches. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, 1992.

[24] H. Etoh. GCC extension for protecting applications from stack-smashing attacks (ProPolice), 2003. http://www.trl.ibm.com/projects/security/ssp/.

[25] G. McGary. Bounds Checking in C and C++ using Bounded Pointers, 2000. http://gnu.open-mirror.com/software/gcc/projects/bp/main.html.

[26] B. Gates. We can and must do better. Microsoft Memo, January 2002. http://news.com.com/2009-1001-817210.html.

[27] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative versioning cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, Feb. 1998.

[28] J. Gray. The Transaction Concept: Virtues and Limitations. *Proceedings of VLDB*, 81:144–154, 1981.

[29] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1-4):135–150, 1993.

[30] L. Hammond, B. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 1–13, New York, NY, USA, 2004. ACM Press.

[31] L. Hammond, B. D. Carlstrom, V. Wong, M. Chen, C. Kozyrakis, and K. Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *Mico's Top Picks, IEEE Micro*, 24(6), nov/dec 2004.

[32] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra microprocessor. *Proceedings of the 23th Annual International Symposium on Computer Architecture*, pages 67–77, 1996.

[33] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. *ACM SIGOPS Operating Systems Review*, 32(5):58–69, 1998.

[34] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, Jun 2004.

[35] M. Harren and G. C. Necula. Lightweight wrappers for interfacing with binary code in ccured. In *Software Security Symposium (ISSS'03)*, Nov. 2003.

[36] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM Press.

[37] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[38] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III. Software transactional memory for dynamic-sized data structures. *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, 2003.

[39] M. Herlihy and J. Moss. Transactional memory: architectural support for lock-free data struct ures. Technical report, Digital Equipment Corporation Cambridge Research Lab, 1992.

[40] M. Herlihy and J. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[41] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[42] Internation technology roadmap for semiconductors 2005 edition. ITRS, 2005. http://www.itrs.net/Links/2005ITRS/Home2005.htm.

[43] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference*, pages 275–288, June 2002.

[44] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[45] M. Kaempf. Vudo malloc tricks. *Phrack*, 0xB(0x39), Aug. 2001.

[46] A. Kahng. The 2001 ITRS: Roadmap for Deisgn and Shared Brick Walls. *Michigan EECS Dept Talk*, March 2002.

[47] R. Kessler, E. McLellan, and D. Webb. The Alpha 21264 microprocessor architecture. In *International Conference on Computer Design*, Dec. 1998.

[48] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th Usenix Security Symposium (SEC02)*, Aug. 2002.

[49] P. Kolte and M. Wolfe. Elimination of redundant array subscript range checks. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–278, 1995.

[50] S. Kumar, M. Chu, C. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 209–220, 2006.

[51] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[52] S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[53] V. Markstein, J. Cocke, and P. Markstein. Optimization of range checking. In *Symposium on Compiler Construction*, pages 114–119, June 1982.

[54] A. McDonald, J. Chung, H. Chafi, C. Cao Minh, B. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of TCC on chip-multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, Sept 2005.

[55] The journey to trustworthy computing: Microsoft execs report first-year progress. Microsoft PressPass- Information for Journalist, January 2003. http://www.microsoft.com/presspass/features/2003/jan03/01-15twcanniversary.mspx.

[56] D. Moore, C. Shannon, and J. Brown. Code-red: A case study on the spread and victims of an inernet worm. In *Proceedings of the Second ACM Sigcomm Internet Measurement Workshop 2002*, pages 273–284, Marseille, France, November 2002.

[57] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: Log-based transactional memory. In *HPCA '06: Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, Washington, DC, USA, 2006. IEEE Computer Society.

[58] M. Moravan, J. Bobba, K. Moore, L. Yen, M. Hill, B. Liblit, M. Swift, and D. Wood. Supporting nested transactional memory in LogTM. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, New York, NY, USA, October 2006. ACM Press.

[59] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *32th Annual International Symposium on Computer Architecture*, Madison, WI, June 2005.

[60] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[61] R. Netzer and B. Miller. What Are Race Conditions? Some Issues and Formalization. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, 1992.

[62] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium*, Feb. 2005.

[63] A. C. of Software and H. T. for x86 Virtualizatioin. K. adams and o. agesen. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, New York, NY, USA, October 2006. ACM Press.

[64] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. *ACM SIGPLAN Notices*, 31(9):2–11, 1996.

[65] M. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. *Software - Practice and Experience*, 27(1), Jan. 1997.

[66] M. Pettersson. Perfctr. http://user.it.uu.se/ mikpe/linux/perfctr/.

[67] F. Pollack. New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies. *Keynote speech: 32nd International Symposium on Microarchitecture*, 1999.

[68] K. Poulsen. Tracking the blackout bug. Security Focus, April 2004. http://www.securityfocus/news/8412.

[69] M. Prvulovic and J. Torrelas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *30th Annual International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[70] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA05)*, Feb. 2005.

[71] R. Rajwar and J. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture, December*, pages 01–05, 2001.

[72] R. Rajwar and J. Goodman. Transactional lock-free execution of lock-based programs. *Proceedings of the Tenth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, October 2002.

[73] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. *SIGARCH Comput. Archit. News*, 33(2):494–505, 2005.

[74] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and W. S. B. Jr. Enhancing server availability and security through failure-oblivious computing. In *6th Symposium on Operating System Design and Implementation(OSDI)*, Dec. 2004.

[75] RTI. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. NIST, Research Triangle Park, NC, May 2002.

[76] O. Ruwase and M. Lam. A practical dyanmic buffer overflow detector. In *11th Annual Network and Distributed Security Symposium (NDSS 2004)*, pages 159–169, San Diego, California, February 2004.

[77] S. Savage, G. Voelker, and G.Varghese. NSF CyberTrust center proposal: Center for internet epidemiology and defenses, 2004.

[78] C. Shannon and D. Moore. The spread of the Witty worm. *Security & Privacy Magazine, IEEE*, 2(4):46–50, July-August 2004.

[79] M. Shapiro II and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[80] N. Shavit and D. Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[81] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. *Computer Architecture, 1995. Proceedings. 22nd Annual International Symposium on*, pages 414–425, 1995.

[82] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[83] J. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, 1998.

[84] G. Suh, J. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[85] M. Sullivan and R. Chillarege. Software defects and their impact on system availability. In *21st International Symposium on Fault Tolerant Computing*, Montreal, 1991.

[86] J. Torrellas, M. Lam, and J. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Trans. Computers*, 43(6):651–663, 1994.

[87] I. Traiger. Virtual memory management for database systems. *ACM SIGOPS Operating Systems Review*, 16(4):26–48, 1982.

[88] N. Tuck, B. Calder, and G. Varghese. Hardware and binary modification support for code pointer protection against buffer overflows. In *37st International Symposium on Microarchitecture*, Dec. 2004.

[89] Cops take a bite, or maybe a nibble, out of cybercrime. USA-Today, Sept. 2003. http://www.usatoday.com/money/industries/technology/2003-09-01-blaster-cover_x.htm.

[90] D. Wheeler. More than a gigabuck: Estimating gnu/linux's size. http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html.

[91] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, pages 149–162, February 2003.

[92] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proceedings of ASPLOS-X*, Oct 2002.

[93] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *22nd Annual International Symposium on Computer Architecture*, pages 24–36. Association for Computing Machinery, 1995.

[94] L. Wu, C. Weaver, and T. Austin. CryptoManiac: A Fast Flexible Architecture for Secure Communication. *28th Annual International Symposium on Computer Architecture*, pages 110–119, 2001.

[95] M. Xu, R. Bodik, and M. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *30th Annual International Symposium on Computer Architecture*, San Diego, CA, 2003.

[96] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Z. S. Midkiff, and J. Torrellas. Accmon: Automatically detecting memory-related bugs via program counter-based invariants. In *37st International Symposium on Microarchitecture*, Nov. 2004.

[97] P. Zhou, F. Qing, W. Liu, Y. Zhou, and J. Torrellas. iwatcher: Efficient architecture support for software debugging. In *31st annual International Symposium on Computer Architecture (ISCA'04)*, June 2004.

[98] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and nontransactional actions. In *TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[99] C. Zilles and D. Flint. Challenges to providing performance isolation in transactional memories. In *4th Annual Workshop on Duplicating, Deconstructing and Debunking*, New York, NY, USA, June 2005. ACM Press.