UNIVERSITY OF CALIFORNIA
RIVERSIDE

IMPRESS: Improving Multicore Performance and Reliability via Efficient Support for Software Monitoring

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Vijayanand Nagarajan

December 2009

Dissertation Committee:

Professor Rajiv Gupta, Chairperson
Professor Walid Najjar
Professor Frank Vahid

The Dissertation of Vijayanand Nagarajan is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

My sincere thanks to my advisor Prof. Rajiv Gupta, with whom I have worked for the past 5 years. I have learnt a lot about compilers, architecture and research from him during this time. I have also learnt a lot from the ways in which he interacts with his students. His humility, patience and concern for his students are characteristics that have really struck me. I have thoroughly enjoyed the time spent with him solving research problems, and I want to thank him for all the help, advice and encouragement towards this dissertation. I also thank my other committee members Prof. Walid Najjar and Prof. Frank Vahid.

I would also like to thank all the members of my research group including Dr. Dennis Jeffrey, Dr. Sriraman Tallam, Dr. Bengu Li, Dr. Xiangyu Zhang, Dr. Arvind Krishnaswamy, Mr. Chen Tian, Mr. Min Feng and Mr. Changhui Lin for helping me a lot. Special thanks to my friends Balaji, Jeffrey, Arun, Varun, Mubeen, Priya, Divya, Abira, Aishwarya and Gayatri for making my life in Tucson and Riverside that much more happier.

Finally, I would like to thank all my teachers, right from preschool to graduate school. They have given me the necessary skills and knowledge, which has led to this dissertation. Last, but not least, my deepest thanks to my parents for giving me a good life and a good education.

To my parents, Mrs. Prema Nagarajan and Mr. S. Nagarajan.

ABSTRACT OF THE DISSERTATION

IMPRESS: Improving Multicore Performance and Reliability via Efficient Support for Software Monitoring

by

Vijayanand Nagarajan

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2009
Professor Rajiv Gupta, Chairperson

With the advent of multicores, there is demand for monitoring parallel programs running on multicores for enhancing reliability and performance. Debugging tools such as *data race detection* and *deterministic replay debugging* (DRD) require a parallel program to be monitored at runtime. Likewise, techniques such as *dynamic information flow tracking* (DIFT) that are used for preventing security attacks also require runtime monitoring. Furthermore, techniques such as *speculative parallelization* and *speculative optimization*, that strive to expose parallelism and increase performance of programs, also require runtime monitoring – detecting *misspeculation*, which is an integral component of any speculative technique requiring the program to be monitored at runtime.

While each of the above monitoring applications are quite different in their purpose and implementation, they all share a common requirement in the context of monitoring a parallel program running on a multicore – the need to detect and react to *interprocessor shared memory dependences* (ISMD). Current software based monitoring tools, due to their inability to detect and react to ISMDs efficiently, are rendered inapplicable for monitoring

parallel programs running on multicores. On the contrary, hardware based monitoring tools, while applicable in a multicore context, require *specialized* hardware modifications specific for each monitoring task.

This dissertation *IMPRESS* strives to Improve Multicore Performance and and Reliability by providing Efficient Support for enabling Software based monitoring. To enable software based monitoring on multicores, this dissertation proposes *ECMon* – lightweight and general purpose support for exposing cache events to the software, in effect, efficiently exposing ISMDs to the software. Using ECMon, a variety of monitoring applications, which were inapplicable on multicores, can now be used to efficiently monitor parallel program on multicores. More specifically, a class of monitoring applications known as *shadow memory tools* such as DIFT for security, Memcheck and Eraser for debugging, can now monitor parallel programs running on multicores at almost the same execution overhead as monitoring sequential programs, using ECMon support. ECMon can also be used to develop novel monitoring applications for increasing performance and reliability. In particular, ECMon can be used for performing speculative optimizations on parallel programs which results in about 14.5% execution time reduction in a set of seven parallel programs considered. Finally ECMon can be used by servers to recover from memory errors without requiring heavy-weight checkpointing or rollback.

To summarize, this dissertation proposes light-weight and general purpose support in the form of exposing cache events to the software. Using this support, it is shown how parallel programs running on multicores can be monitored efficiently for increasing performance and enhancing reliability.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Processor designers can no longer rely on frequency scaling for increased performance, with the *power wall* thwarting further increases in processor frequency. This has forced industry to turn to parallel processing, in the form of multicores, for increased performance. However, extracting performance from multicores is far from a solved problem. It is contingent on programmers writing parallel software that can take advantage of the multicores – a task that is recognized as one of the biggest challenges facing academia and industry currently. In fact, in the words of John Hennessy the above problem is "as hard as any that computer science has faced" [67].

Demand for parallel software has sparked off yet another demand – the demand for *monitoring* parallel programs running on multicores. In other words, there is a demand for efficiently performing a variety of dynamic analyses while a parallel program is running on a multicore for applications in debugging, security and performance. Debugging tools such as *data race detection*, techniques such as *dynamic information flow tracking* (DIFT) that are used for preventing security attacks, and *speculative* techniques that strive to expose

parallelism and increase performance of programs, are some of the tools that require runtime monitoring.

1.1 Applications of Runtime Monitoring

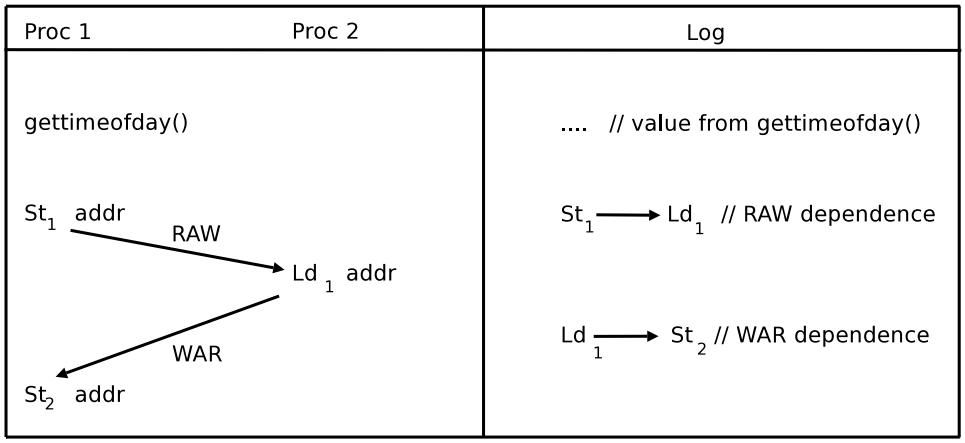


Figure 1.1: Runtime monitoring involved in debugging (DRD): recording a parallel program execution consists of storing the values returned by system calls and remembering dependences enforced in a *log* which is used during replay.

(Debugging) Writing a correct parallel program is hard, and consequently, there is need for tools that assist the programmer in the debugging of parallel programs. Unfortunately, debugging a parallel program is also a cumbersome task. Bugs that manifested during one run, mysteriously disappearing while debugging, are commonplace in parallel and multithreaded programs. These bugs, known as *Heisenbugs* [29], arise due to non-determinism inherent in parallel programs. The first challenge in debugging a parallel program is to *reproduce the bug. Deterministic replay debuggers* (DRD) [61, 85, 83, 95, 111] record the execution of a program and in this process they capture the non-determinism inherent in the program. Once the non-determinism has been captured, the recorded infor-

mation is used to *replay* the program and thus the bug is reproduced. Recording program execution involves capturing events such as system calls during execution. For instance, the linux system call `gettimeofday` returns the current time and hence returns a new value every execution. Consequently, the values returned by these system calls are recorded and stored in a *log*. During replay, the values from the recorded log are used instead of executing the system call again. Another important source of non-determinism in parallel programs involve shared memory races [61, 111]. Unsynchronized memory accesses (involving a memory write) can lead to memory races, which essentially are memory dependences enforced in a non-deterministic fashion. Thus to replay the execution of a parallel program faithfully, the order in which memory dependences are enforced is remembered. As illustrated in Fig. 1.1, the RAW dependence between St_1 and Ld_1 and the WAR dependence between Ld_1 and St_2 are remembered in the log. During replay, it is made sure that these dependences are enforced. In a similar vein, *data race detection tools* [50, 51, 84, 88, 113] also require that memory dependences enforced in the execution of the parallel program be monitored.

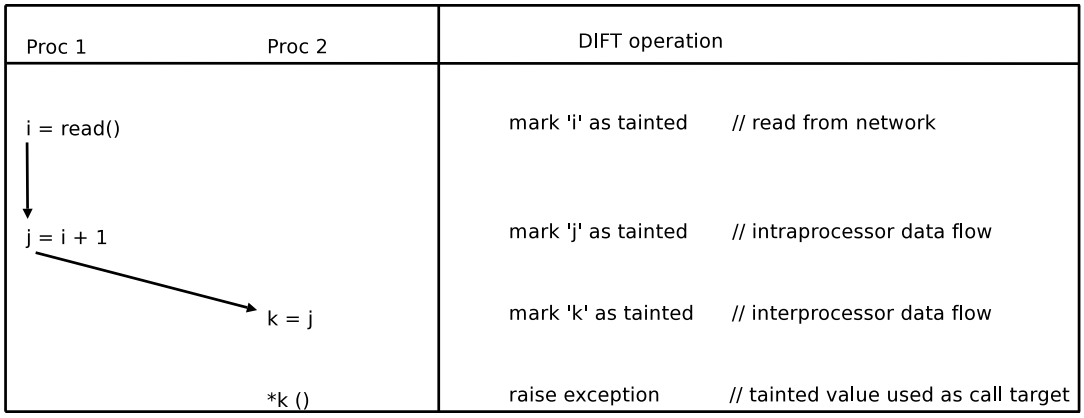


Figure 1.2: Runtime monitoring involved in security (DIFT): tracking information flow in parallel programs consists of tracking intraprocessor and interprocessor data flow

(Security) Runtime monitoring also has applications in security. Software attacks have become increasingly prevalent. US CERT Statistics [2] show that the number of attacks have increased rapidly over the years. At the same time it has become increasingly costly for businesses to deal with worms and viruses. FBI computer crime survey estimates that US businesses spent a total of $67.2 billion in the year 2005 to deal with computer crime, a significant proportion of which was spent in dealing with software attacks (worms and viruses). *Dynamic Information flow tracking* (DIFT) [23, 66, 73, 97] is a promising technique for providing security against malicious software attacks. The basic idea hinges on the fact that an important avenue through which an attacker compromises the system is through input channels. This is a direct consequence of most of the vulnerabilities being *input validation* errors. In fact, 72% of the total vulnerabilities discovered in the year 2006 are attributed to a lack of (proper) input validation [5]. Note that most of the memory errors including buffer overflow, boundary condition and format string errors fall into this category. The main principle of DIFT is as follows. A set of input channels, for example network inputs, are considered insecure. The flow of information from these inputs is tracked and those values that are data dependent on such inputs are in turn marked tainted as shown in Fig. 1.2. Potential attacks are detected upon the suspicious use of tainted values. Since DIFT relies on tracking the flow of values as the program executes, it requires the program executing be monitored during runtime. With the advent of multicores, critical servers programs are now parallelized and being made to run on multicores. Thus, there is a need to ensure that DIFT techniques work for parallel code running on multicores.

(Performance) Finally, runtime monitoring is also essential for techniques that strive to increase performance. With the advent of multicores, there is a great demand

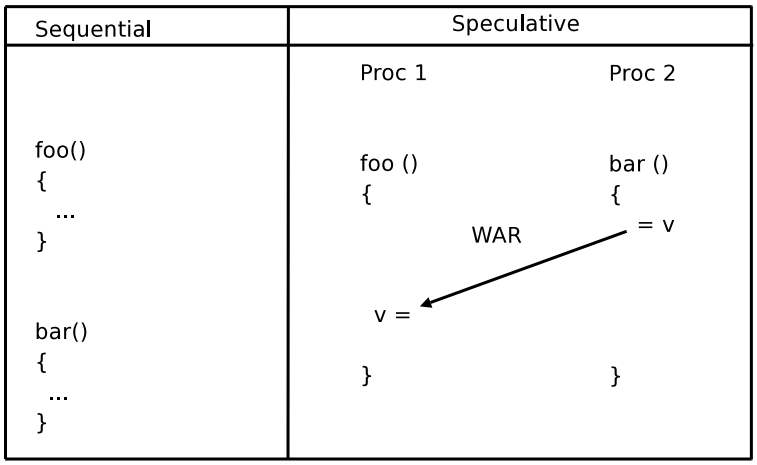


Figure 1.3: Runtime monitoring involved for performance (speculative parallelization): detecting misspeculation involves tracking if a value read speculatively (function bar) is later written into non-speculatively (function foo)

for automatic techniques that parallelize sequential programs. However, extracting parallelism from sequential programs has proven to be a very hard problem. Two sequential functions (for instance functions *foo* and *bar* as shown in Fig. 1.3) can be executed in parallel if, and only if, there is no dependency between them; in other words, they can be executed in parallel only when it can be proven that function *bar* does not read anything written by function *foo*. However, it is hard to precisely guarantee the absence of dependence in the presence of pointer variables [20, 34]. This is because the above requires *interprocedural alias analysis*, which is known to be quite conservative. Even with sophisticated static analyses, we may not be able to expose parallelism in the presence of *infrequent dependences*. For example, even if there is a single dependence between functions *foo* and *bar*, as infrequent as it may be, the two functions still can not be parallelized. Recently, there has been significant interest on techniques that use *speculation* [25, 103] for parallelizing sequential programs. These techniques speculate that there will be *no dependence* between two sequential chunks

of execution, and proceed to *speculatively* execute them in parallel, even if static analysis cannot guarantee the absence of dependence. For example, the sequential functions *foo* and *bar* may be speculatively executed in parallel, as shown in Fig. 1.3. If there is indeed no dependence between the two sequential chunks then the speculation is said to have succeeded and the parallelization is successful. If, however, there is an observed dependence between the two functions during runtime, then there is said to be a *misspeculation* and the speculative state is squashed. The success of speculative parallelization is dependent on two important factors. First, the misspeculation rate, which is the rate at which speculation fails, should be low. Fortunately, profiling techniques can be used to estimate the misspeculation rate and speculation is actually performed only when it is expected to be beneficial [25, 103]. Second, the success of speculation is also contingent upon the efficient detection of misspeculation. Indeed, a key component of every speculation technique is the efficient monitoring of the program execution for the detection of misspeculation. In the above scenario, there is a misspeculation if there is a dependence detected between the speculative and the non-speculative functions (function*foo* and *bar* respectively). Thus runtime monitoring techniques, that ascertain the absence of such dependences is an integral part of speculation.

1.2 Detecting and Reacting to ISMDs

While each of the above monitoring tasks are different in their purposes and implementation, they share a common requirement in the context of monitoring parallel programs running on multicores. In each of the monitoring tasks, there is a need to detect *interprocessor shared memory dependences* (ISMD) and react to it. For instance in DRD, ISMDs,

since they are a source of non-determinism in parallel programs, have to be remembered. Consequently, the ISMDs as shown in Fig. 1.1 has to be logged; during replay, the logged dependences are enforced to recreate the original execution. Likewise, ISMDs that are unsynchronized, constitute data races and hence there is a need to detect ISMDs in data race detectors. DIFT technique for ensuring security relies on tracking both intraprocessor and interprocessor data dependences as shown in Fig. 1.2 and hence ISMDs have to monitored. In speculative parallelization, an ISMD signals a misspeculation and so ISMDs have to be monitored, as shown in Fig. 1.3.

Current software monitoring approaches, due to their inability to detect ISMDs efficiently, are rendered inapplicable for monitoring parallel programs running on multicores. Table 1.1 shows some of the current monitoring tools and the scenarios in which they are applicable. As we can see, LIFT [73] and Taint Analysis [66] which are tools for implementing DIFT are only applicable for sequential programs. Flashback [95] and Jockey [85], which are tools for performing DRD, are only applicable for sequential programs and multithreaded programs running on a uniprocessor. However, they are not applicable for multithreaded programs running on a multicore. It is the same with CHESS [50] and Eraser [88] which are data race detection tools, and Valgrind's Memcheck [63] which is a

Software Tool	Purpose	Sequential Programs	Multithreaded on unicore	Multithreaded on multicores
LIFT, Taint Analysis	DIFT	Yes	No	No
Flashback, Jockey	DRD	Yes	Yes	No
Valgrind	Memcheck	Yes	Yes	No
CHESS, Eraser	Race Detection	Yes	Yes	No

Table 1.1: Current software based monitoring tools are not applicable for monitoring parallel programs running on multicore.

tool for detecting memory errors. The inability to detect ISMDs have also affected software based speculative parallelization techniques. Behavior oriented parallelism (BOP) [25] and CORD [103] are two such techniques, which can only expose coarse-grained-parallelism; the high cost of detecting an ISMD for misspeculation making them unsuitable for exposing fine-grained-parallelism.

1.3 Dissertation Overview

This dissertation observes that the ISMDs, while hard to detect in software, can be easily *exposed* to software using lightweight architectural support. To this end, this dissertation proposes *ECMon* [54], support for exposing cache events to the software, in effect, efficiently exposing ISMDs to the software. Whenever the cache controller of a processor receives a cache event (eg. invalidate, data value reply), it interrupts the processor and calls a predefined handler function. The handler function is programmable and is defined based upon the requirement of the monitoring task. Using ECMon it is shown how a variety of monitoring tasks, including novel monitoring tasks, can be implemented efficiently for parallel programs running on multicores.

First, it is shown how ECMon support can be used to record ISMDs efficiently, only incurring about 3 fold execution time reduction [54]. With added support for recording ISMDs, DRD tools like *flashback* and *jockey* can now be used to record execution of parallel programs on multicores. Second, it is shown how a class of monitoring tools known as *shadow memory tools* [63] including DIFT,Eraser and Memcheck from Table 1.1, can be implemented for parallel programs running on multicores using ECMon [55, 53]. Furthermore, it is shown that the above tools can be implemented very efficiently, incurring roughly the

same execution time overhead as monitoring sequential programs. In other words, shadow memory tools, which were previously unable to monitor parallel programs, can now monitor parallel programs running on multicores, with little additional overhead. It is also shown how ECMon can be used for performing a variety of speculative optimizations for parallel programs, which resulted in about 14.5% decrease in execution times for the set of parallel programs considered [54]. Finally it is shown that ECMon support can be used by server programs for recovering from memory errors, without the need for heavy-weight techniques such as checkpointing or rollback [57].

The contributions of this dissertation are as follows:

- ECMon, support for exposing cache events to software, is proposed. By exposing cache events to the software, software is made aware of ISMDs. This enables software based monitoring of parallel programs on multicores.

- ECMon is light-weight, requiring minimal hardware changes. More specifically, ECMon requires no changes to the processor pipeline and the cache coherence protocol.

- ECMon is programmable and general purpose. This is illustrated by implementing a variety of monitoring tools, including novel monitoring tools, using ECMon support.

More specifically, ECMon is used to build the following tools that can be used for increasing performance, enhancing reliability and security of parallel programs on multicores.

- ECMon support is used to record shared memory dependences (ISMDs) in software, incurring only 3 fold execution time reduction [54] in the process, a task which had no prior software based solutions.

- ECMon support is used to efficiently implement a class of monitoring applications known as *shadow memory tools* including DIFT, Memcheck and Eraser for parallel programs running on multicores. Experimental results shows that ECMon is used to implement DIFT (Memcheck, Eraser) at only 5 fold (10 fold , 10 fold) execution time slowdown; whereas without ECMon support, the same monitoring tasks would have incurred 14 fold (22 fold, 24 fold) execution time slowdown.

- ECMon is used, in a novel fashion, as a framework for performing speculative optimizations and exposing fine grained parallelism in parallel programs for increasing performance. By speculatively executing past barrier synchronizations, the execution time is reduced by a 12%, and by speculatively promoting variables to registers in the presence of synhronization operations, execution time is further reduced by 2.5%.

- Finally ECMon is used, in a novel fashion, by server programs to efficiently recover from memory errors without the need for heavy-weight techniques that use checkpointing and rollback. Experimental results shows that server programs are slowed down by only 5% using the above technique.

To summarize, this dissertation *IMPRESS* Improves Multicore Performance and and Reliability by providing Efficient Support for enabling Software based monitoring. The rest of the dissertation is organized as follows. Chapter 2 introduces and describes the architectural support involved in ECMon. Here it is illustrated how this support can be used to capture and record ISMDs efficiently for applications in DRD. Chapter 3 describes how ECMon can be used to implement shadow memory based monitoring tools on multicores. Chapter 4 describes how ECMon can be used for implementing speculative optimizations in

parallel programs. In chapter 5, the concept of self recovery in server programs is discussed, followed by how ECMon can be used to implement self recovery in server programs. While chapter 6 discusses related work, this dissertation concludes in chapter 7.

Chapter 2

ECMon: Support for Exposing Cache Events

In this chapter, the motivation for exposing cache events to the software is first discussed. Then the support involved in exposing cache events to the software is discussed in detail. After describing the system model that is assumed, the main ECMon support, which consists of the exposed cache events and the associated handlers are discussed. Then the ISA support, which consists of the new instructions through which the programmer interacts with ECMon, is discussed in detail. Then, the architectural support involved for enabling ECmon is discussed. The next part of the chapter illustrates the usage of ECMon in deterministic replay debugging (DRD). DRD is a debugging technique for parallel programs , which requires than ISMDs are recorded. This is accomplished using ECMon where the exposed ISMDs are recorded using software handlers. This chapter concludes with experimental results that evaluates the efficiency with which ECMon support can be used to record ISMDs.

2.1 Motivation for Exposing Cache Events

The advent of multicores has given rise to the need for monitoring parallel programs running on multicores for applications in debugging, security and performance. However, each of the above monitoring applications requires ISMDs to be detected. Since software based monitoring techniques are not able to detect ISMDs efficiently, they are rendered inapplicable for monitoring on multicores. On the contrary, hardware based monitoring techniques are able to deal with ISMDs and hence are applicable in a multicore setting. However, hardware based monitoring techniques are specialized, in that, each monitoring task requires its own specific hardware support. This makes hardware based monitoring solutions impractical, as it is unlikely that chip manufacturers would add hardware support for a particular monitoring application. This dissertation makes the case for exposing cache events to the software, in effect, efficiently exposing ISMDs to the software. With this added support, software based monitoring techniques can now be made to monitor parallel programs running on multicores.

2.1.1 Role of ISMDs in monitoring applications

Table 2.1 shows a list of monitoring applications and the role of ISMDs in each of the applications. It also lists the software and the hardware based solutions for each of the monitoring applications. DIFT, which is a technique for detecting software attacks, relies on tracking intraprocessor and interprocessor data flow. Hence ISMDs, which are essentially used to implement interprocessor data flow, have to be tracked to implement DIFT. Software based techniques for implementing DIFT [15, 66, 73] are unable to detect ISMDs efficiently which renders them unsuitable for monitoring parallel programs on multicores.

Monitoring Application	Role of ISMDs	Software	Hardware
Security DIFT (Dynamic Information Flow Tracking)	ISMDs are used to track interprocessor data flow.	Dynamic taint analysis [66], LIFT [73], Taintrace [15] cannot deal with ISMDs and hence inapplicable in a multicore setting.	DIFT [97], Raksha [23], Flexitaint [28] require changes to processor pipeline, caches and memory to track intraprocessor and interprocessor information flow.
Debugging DRD (Deterministic replay debugging)	ISMDs have to be remembered to enable deterministic replay of parallel programs	Flashback [95], Jockey [85] cannot deal with ISMDs and and are applicable only for multithreaded programs running on a uniprocessor.	FDR [111], Bugnet [61], Rerun [35] Delorean [48] require changes to caches and cache coherence system to detect ISMDs.
Debugging Data race detection	ISMDs which are unsynchronized are data races and hence ISMDs have to be detected	CHESS [50] Eraser [88] cannot deal with ISMDs and are applicable only for multithreaded programs running on a uniprocessor.	HARD [113] and SigRace [51] require changes to cache coherence system to detect ISMDs.
Performance Speculative Parallelization	ISMDs signal misspeculation.	Cord [103] BOP [25] cannot detect ISMDs and hence cannot expose fine grained parallelism	TLS [18, 32] and Speculative synchronization [46] Speculative lock elision [74] can expose finegrained parallelism but require changes to caches and cache coherence system to detect ISMDs.

Table 2.1: ISMDs in Monitoring Applications.

On the contrary, hardware based techniques for implementing DIFT [23, 97, 28] are able to detect ISMDs with additional hardware support; the additional hardware consists of changes to the processor pipeline and caches for implementing the tracking operations in hardware. DRD, which is a technique for enabling deterministic replay of parallel programs, requires that the ISMDs be remembered and recorded, so they can be faithfully remembered. Software based implementations of DRD [85, 95], since they are unable to detect ISMDs efficiently can only record and replay multithreaded programs running on a uniprocessor. On the contrary, hardware based techniques for implementing DRD [48, 35, 61, 111] with changes to cache and cache coherence protocol, are able to detect ISMDs are hence are applicable in a multicore setting. Finally speculative parallelization techniques for increasing the performance of programs, also need to detect ISMDs, since they signal a misspeculation. Software based speculative parallelization techniques [25, 103], since they are unable to detect misspeculation efficiently are unable to expose finegrained parallelism. On the contrary hardware based speculative parallelization techniques [18, 32, 46, 74], with changes to caches and coherence protocol, are able to detect misspeculation (ISMDs) efficiently and are able to expose fine grained parallelism in programs.

2.1.2 Approach

In this dissertation it is observed that the ISMDs, while hard to detect in software, can be easily *exposed* to software using lightweight architectural support. This is because the cache coherence messages (along with the *read miss* and *write back* cache events) reveal all the ISMDs. Indeed, several of the hardware based monitoring techniques piggyback meta data along with cache coherence messages to achieve a specific monitoring task. For

example, FDR [111] and Bugnet [61] techniques piggyback time stamps along with cache coherence messages to remember ISMDs. However the above techniques also include specialized hardware support for accomplishing the specific monitoring task. For example, Bugnet utilizes changes to processor pipeline specifically for recording a program's execution. The goal of this work is to identify the *minimal hardware support* for enabling all the above monitoring applications. This follows the design principle of having the common case in hardware and the rest in software. This also greatly enhances the practicality, as the proposed hardware support is general-purpose with several applications. To this end, this dissertation proposes *ECMon* [54], support for *exposing cache events* to the software, in effect, efficiently exposing ISMDs to the software. Software techniques, now aware of ISMDs using ECMon support, can now be used to implement all the above monitoring applications for parallel programs running on multicores.

2.2 System Model

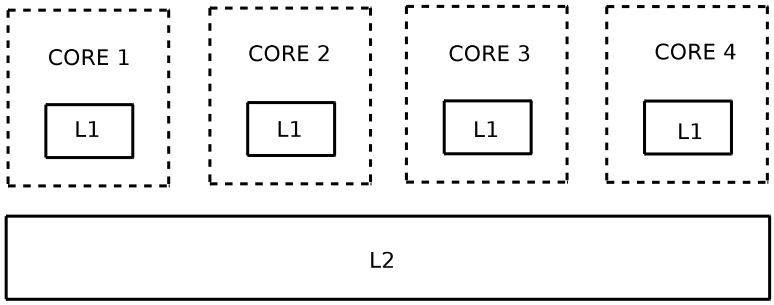


Figure 2.1: System Model: Multicore processor with coherent local L1 caches and a shared L2 cache

For this discussion, a multicore processor with local caches and a shared lower level cache is assumed as shown in Fig. 2.1. Further, it is assumed that the local caches are

writeback caches kept coherent using an *invalidate* based hardware coherence protocol. In other words, it is assumed that a write issued by a processor will first update the value in the processor's local cache; the value in the lower level cache will only be updated (*written back*), when a block is replaced from the local cache. Consequently, when the value in the local cache is written into, it would make the values in other local caches stale. To prevent this situation and to maintain cache coherency, a write to a local cache will *invalidate* all other shared copies in other local caches.

2.3 ECMon

In *ECMon*, cache events are exposed to the software; in effect, efficiently exposing the ISMDs to the programmer.

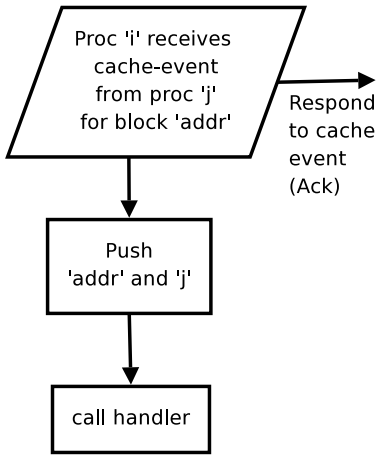


Figure 2.2: ECMon semantics

2.3.1 Events and Handlers

Whenever the cache controller receives a cache event for a processor's local cache, it can be programmed to interrupt the processor, and call a predefined *handler* function, before responding to the event as shown in Fig. 2.2. The cache events exposed for the applications in this work are the following: *(i) a processor sending/receiving an invalidate message*, (ii) *a processor sending/receiving a data value reply*, (iii) *a processor experiencing a read miss for a block uncached in any processor* and (iv) *a processor about to write back a block* as illustrated in Table. 2.2. While the first four events expose ISMDs exercised via the cache coherence network, the last two events expose ISMDs exercised via memory. The handler function is programmable and is defined based upon the requirement. Since the handler resides in user space, the semantics of the call to the handler is similar to a function call; the programmer is responsible for saving and restoring the values of registers that are used in the handler. However, the hardware is responsible for providing values to the handler as *function call parameters* as mentioned in Table 2.2. In general, the *block address* and the *remote proc id* are the parameters. However, for the *send invalidate* event, since there are potentially multiple remote processors, the parameter for this event is the number of remote processors holding the invalidated block. Finally, it is important to note that while the handler function is called, the coherence controller *independently* responds to the cache event, as usual.

2.3.2 ISA Support and Hardware Support

Through the proposed ISA interface, the programmer interacts with the processor and is able to effectively utilize ECMon support. The programmer notifies the hardware

Event	Parameters
receive data value reply	block address, remote proc id
send data value reply	block address, remote proc id
receive invalidate	block address, remote proc id
send invalidate	block address, number of remote procs
read miss	block address
write miss	block address

Table 2.2: Exposed Cache Events.

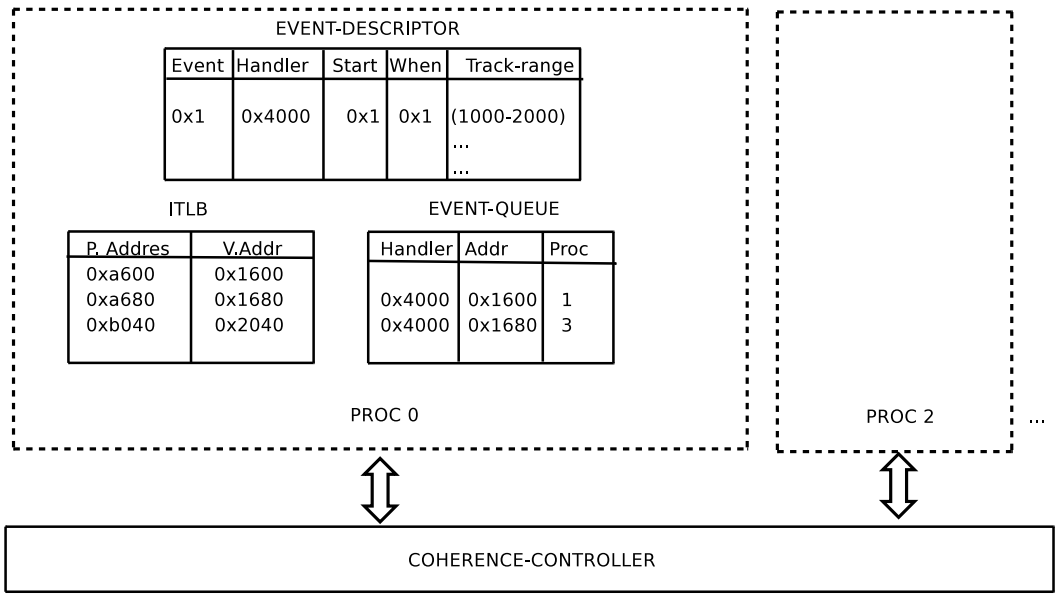


Figure 2.3: Hardware support for enabling ECMon

through the `handler` instruction, which handler to call for what event. While the event is expressed via the predefined *event-code*, the handler is specified with its start address. The programmer is also given the ability to mark regions of code where the *ECMon support is active*, with the `start-handler` and `end-handler` instructions; the handler is actually called upon reception of the event only for execution within these two points. Furthermore, the programmer is given flexibility to specify *when* the handler will be called, upon reception of an event. For this purpose, the `start-handler` instruction takes a *when* bit as one of its operands; a 0 indicates that the handler will be called as soon as the event is received; whereas a 1 indicates that the handler should be called only on *specific points*. If a 1 is specified as the operand, the programmer inserts the `call-handler` instruction (within *start-handler* and *end-handler*). When the processor receives an event, it does not call the handler immediately and only calls the handler when the `call-handler` is encountered. It is very useful for the programmer to control when the handler will be called. For example, in speculative execution we may need to call the handler after the speculation to verify its correctness. Likewise, this feature can be used to handle the *atomicity* problem associated with software monitoring [16, 63], as we shall see later. Finally, through the `track-range` instruction, the programmer is given the ability to specify the *range of block addresses* for which the handler will be called. The rationale for supporting this option is to limit the number of times the handler is called.

Having explained the purpose of each of the new instructions added at a high level, we now describe in detail with an example as shown in Fig. 2.4, the semantics of the instructions.

(**Handler instruction**) The programmer notifies the hardware through the `handler`

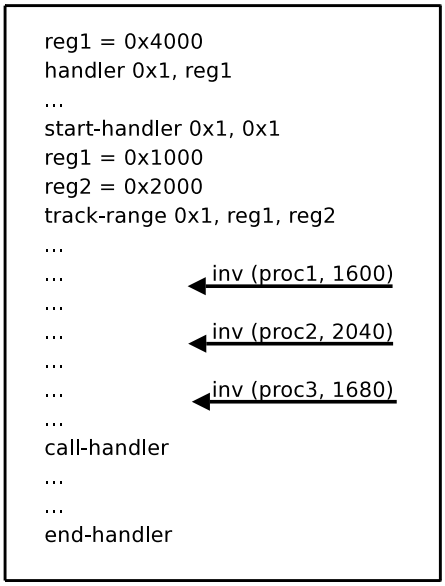


Figure 2.4: An example to illustrate ISA support

instruction, what handler to call for what event. The handler instruction has two operands. The first operand is used to specify the *event code*, which is a predefined code for each cache event. For example, the *event code* $0x1$ may refer to the event when the processor receives an *invalidate* message. Through the second operand which is a register, the programmer specifies the *instruction address* of the handler to the hardware. In the above example, the handler resides in the instruction address *0x4000*. To maintain this information, each core of the multicore processor maintains an *event-descriptor* table. When the `handler` instruction is encountered it adds the *event code* and the handler's instruction address to the *event-descriptor* table as shown in Fig. 2.3.

(**Start-handler and end-handler instructions**) Through the `start-handler` and `end-handler` instructions, the programmer marks the region of code where the *ECMon*

support is active. The handler is actually called upon reception of an event only for execution within these two points. The *start-handler* takes two operands. While the first operand specifies the *event-code*, the second operand is a one bit operand (called the *when* bit) which controls *when the handler should be invoked*. When the *start-handler* instruction is encountered, it sets the *start* bit and the *when* bit in the *event-descriptor*; on the other hand, the *end-handler* clears the *start-bit*.

(**When bit and call-handler instruction**) The *when* bit and the *call-handler* instruction are used to control when the handler is called. A 0 value indicates that the handler will be called *as soon as the event is received*; whereas a 1 indicates that the handler should be called only at *specific points*. If a 1 is specified as the operand, the programmer inserts the `call-handler` instruction at specific points (within *start-handler* and *end-handler*). When the processor receives an event, it does not call the handler immediately and only calls the handler when the `call-handler` is executed. In the above example, the *when* bit is set to 1, which means that the handler will be called only when the *call-handler* instruction is executed. If the processor receives multiple events before encountering the `call-handler`, it is buffered in the *event-queue* and then processed *inorder* when the `call-handler` instruction is finally encountered. In the above example, *proc 0* receives three events before the *call-handler* is encountered, two of which are buffered in the *event-queue*. We will explain why one of them is not buffered after describing the actions for the *track-range* instruction. Finally, when the handler is called, the hardware forces all the *when* bits in the *event-descriptor* table to 1, to make sure that there are no nested handler calls. This ensures that any events that occur when the handler is called are buffered in the *event-queue*. The processor subsequently treats the `return` within the handler like a

`call-handler` and empties the *event-queue*.

(**Track-range instruction**) The programmer is also given the ability to restrict the runtime values of block address for which the handler is called. The `track-range` instruction is used to accomplish this. It has two operands: the *start value* and the *end value*. The handler is called for any value that lies in the range (between *start* and *end*). Once the `track-range` instruction is encountered, the ranges specified in the instruction are stored in the *event-descriptor*. For each entry in the *event-descriptor*, the storage of four such ranges is supported. When the event is encountered, the hardware checks if the block address of the event fall into these range(s). If not, the handler is not called for that event. In the above example, the `track-range` instruction specifies the range as addresses between $0x1000$ and $0x2000$. The second invalidate message with the address $0x2040$ was not buffered in the *event-queue* since it did not lie within this range.

2.3.3 Hardware Support

Fig. 2.3 illustrates the hardware support required for enabling ECMon. Three simple hardware structures are added to each core of the multicore processor: the *event-descriptor*, the *event-queue* and an *itlb*. As discussed earlier, the purpose of the event-descriptor is to store information about the events and its associated handlers, and the purpose of the event-queue is to buffer events, in case the handler call needs to be delayed. Finally, the purpose of the itlb (inverted TLB) is to store the mapping between physical addresses and the virtual addresses. The addresses that the programmer uses (for example in the track range instruction) are all virtual addresses. However, the addresses associated with the coherence events are physical addresses. Thus, the itlb is added as a means of

converting the physical address into virtual addresses.

2.3.4 Completeness

ECMon is complete, in that it is guaranteed to expose *all* ISMDs. ISMDs consisting of RAW, WAW and WAR dependences can be exercised via two modes: through the cache coherence system or through the memory. By exposing all *invalidate* events, all WAR dependences exercised through the coherence system are exposed. Similarly, by exposing all *data value reply* events, all RAW and WAW dependences exercised through coherence are exposed. However, not all dependences are exercised through cache coherence system; some are exercised through the main memory due to cache block replacements. Let us see how the various dependences exercised via memory are exposed:

WAR dependency: *Proc 1* holding a shared block in its local cache (due to a prior read to that block) can later replace that block. A write to the same block by another processor, *proc 2*, results in a WAR dependency between *proc 2* and *proc 1*. If the *bus based* coherence protocol is used *proc 1* would still get the invalidate message from *proc 2* , thereby exposing the WAR dependency. If the *directory* based coherence protocol is used, we make sure that local caches do not notify the directory on shared block replacements, This ensures that *proc 1* will still get the invalidate message from *proc 1*, although *proc 2* has replaced the block, thereby exposing the WAR dependency.

RAW, WAW dependences: *Proc 1* holding a block in exclusive state (due to a prior write to that block) can later write it back to the memory. A read (or write) to the same block by a different processor, *proc 2*, results in a RAW (or WAW) dependency between *proc 2* and *proc 1*, which is exercised through the memory. Two additional events

are exposed to help detect the above dependence. When a processor is about to *write-back an exclusively held block* to the directory, the *write-back* event is exposed to the processor causing the *write-back*. Later when a different processor requests the block, it sends a read-miss to the directory. If such a *read miss* request is received by the directory and is *uncached* in any of the processors, we expose the above *read-miss* event to the processor causing the read miss. To summarize, RAW (WAR) dependences that happen through the memory are detected by exposing the appropriate *write-back* and *read-miss* events.

2.3.5 Correctness

In the design of ECMon, the coherence controller merely calls the handler for specified cache events. Since there is *no change to the coherence protocol itself*, the correctness of the original coherence protocol is retained.

2.4 Recording ISMDs using ECMon

This sections describes an application of ECMon in recording ISMDs which is useful for debugging parallel programs. *Deterministic Replay Debugging* (DRD) [61, 95, 111], is a technique that helps programmers debug their program by replaying the exact execution that causes the bug to manifest itself. Naturally, the first step of DRD is the recording of program information as it executes, so that replay can be enabled. Recording the execution of multithreaded programs involves the recording of ISMDs, since the order in which the ISMDs are exercised are ,in general, non-deterministic. Software based approaches [85, 95] are unable to record these shared memory dependences for multithreaded programs executing in multicores. On the contrary, specialized hardware support, involving changes to

the cache, cache coherency and processor pipeline, has been proposed in prior work [61, 111] to efficiently record these dependences. In this section, ECMon support is used to record these dependences. First, the hardware techniques for recording ISMDs are reviewed, and from this the ECMon based approach is derived. Experimental results show that ISMDs can be recorded efficiently causing only 3 fold execution time slowdown.

2.4.1 HW based recording of ISMDs

The steps involved in recording dependences in hardware recording systems is shown in Fig. 2.5(a). Each processor keeps track of its instruction count in an on-chip counter *instr-count*. Furthermore, each cache block is augmented with space to hold the instruction count in *cache-block[addr]*. Whenever the processor accesses that memory block, it writes the current instruction to it. This is done so that when that memory block is involved in an inter-processor dependence, the time in which it was last accessed can be remembered. Dependences are expressed as edges between processor ids along with each of their instruction counts [61, 111]. The key idea of recording ISMDs is based on the observation that in a multiprocessor with local caches, the cache coherence protocol is actually responsible for enforcing the above dependences and thus the coherence messages reveal the dependences. Thus hardware recorders piggyback cache coherence messages with instruction counts and the hardware takes care of recording these edges. Finally, before recording the dependency the hardware checks if the dependency that is currently recorded is automatically implied by a previously recorded dependency; if so, it will not log the current dependency. We illustrate the above steps with the following simple example with two processors.

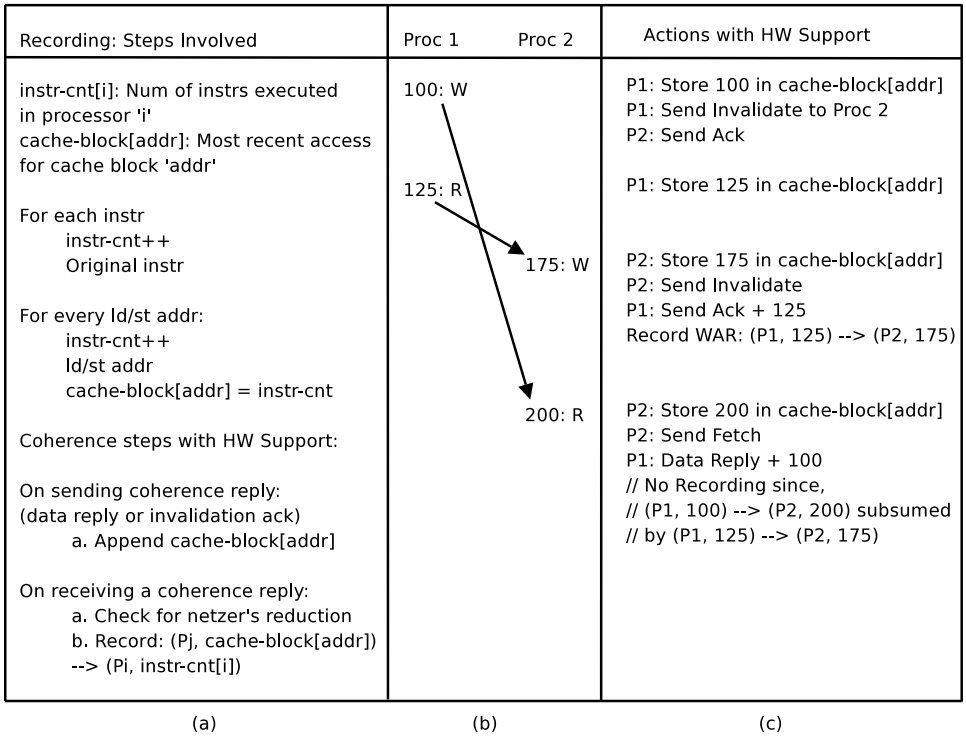


Figure 2.5: Recording ISMDs using HW support

An Example Fig. 2.5(b) shows read and write operations prefixed by their dynamic instruction counts and also indicates the dependency exercised between processors. There are two dependences exercised in the above example: a WAR dependency with the write from processor 2 at instruction count 175 and the read from *proc 1* at instruction count 125. Likewise, there is a RAW dependency with the write from *proc 1* at instruction count 100 and the read from *proc 2* at instruction count 200. Recording ISMDs involves remembering the instruction counts at the time of the write and read, so that the same dependences can be enforced during replay. First, the actions required for recording the WAR dependency are considered. When the read occurs in *proc 1*, the instruction count at the time of the read needs to be remembered, so that the RAW dependency can be recorded when it is subsequently written in processor 2. To remember this count, an instruction count is added to every cache block. Thus as the read is executed, the value 125 is stored in the cache block associated with the read address. When the write is executed in processor 2, the instruction count (175) is similarly remembered in the cache block. Since it is a write, the coherence controller sends an invalidate message to invalidate shared copies of the block in other processors. Consequently, the same cache block that already resides in *proc 1* is invalidated. Once it is invalidated, *proc 1* sends an invalidate acknowledgment, *appending to it* the instruction count when it was last read. Using this, the dependence (P1, 125) $\rightarrow$ (P2, 175) is recorded.

Now, the outer RAW dependency is considered. When the write in *proc 1* is executed, the instruction count (100) is remembered in the cache block. It then proceeds to invalidate shared copies in *proc 2*. Thus when there is a read in *proc 2*, there is a read miss and the value is sent as a *data reply* from *proc 1*, as usual appended with it

the instruction count when the block was written (100). However, before recording the dependency, hardware recorders perform the *Netzer's transitive reduction* [65] test, in which they basically check if the current dependence is actually implied transitively by previous dependences. In fact, in the above example, there is no need to record the outer RAW dependence since it is automatically enforced by the WAR dependence that has already been recorded. The recording of WAW dependences (not shown) takes place similar to RAW dependences.

2.4.2 ECMon for recording

Recording using ECMon is motivated by the fact that all the steps involved in recording dependences, excluding the ones dealing with coherence messages, can already be performed in software. With ECMon support, even the coherence messages can be dealt with under software control. Using shadow memory support [63], instruction counts (*instr-cnt[p]*) are associated for each processor, which are incremented in software for every memory instruction. Likewise, instruction counts are also maintained for every processor's cache block (*cache-block[addr]*) as well as memory block (*directory[addr]*) in software. All stores and loads in the program are instrumented with instructions that copy the processor's instruction count to the shadow memory associated with the processor's cache block for that memory access. The basic steps involved in maintaining these counters are same as shown in Fig. 2.5; only all the variables are actually stored in memory and the counters are maintained using instructions. ECMon support is then used to detect the ISMDs and the associated software handlers are programmed to record ISMDs.

To illustrate the process at a high level let us consider the RAW dependency ex-

ercised between the two processors in Fig. 2.5(b). When the read, accessing block address, *addr*, occurs in *proc 2*, it sends a fetch message to *proc 1*, which contains the block in exclusive state because of the earlier write that happened in *proc 1*. Upon receiving the fetch message, *proc 1* attempts to send a *data-value reply* to *proc 2*. This triggers the software handler. Within the handler, two values are accessed: the instruction count corresponding to the block address, *cache-block[addr]*, and the current instruction count of *proc 2*, *instr-count[2]*. Upon accessing the values, the ISMD is either recorded or skipped after checking for Netzer's transitivity condition. Recording of the WAR dependency in Fig. 2.5(b) proceeds among similar lines. When the write accessing block address, *addr*, occurs in *proc 2*, it sends an invalidate message to *proc 1*. Upon receiving the invalidate message, the software handler is triggered. Within the handler, two values are accessed: the instruction count corresponding to the block address, *cache-block[addr]*, and the current instruction count of *proc 2*, *instr-count[2]*. Upon accessing the values, the ISMD is recorded, after checking for Netzer's transitivity condition.

2.4.3 Correctness Issues

There are some issues that threaten the correctness of the recorded dependences.

(**Atomicity**) The first issue concerns the atomicity of the original and shadow memory operations [16, 63]. Recall that a separate store (denoted W_1') is needed to update the instruction count of *cache-block[addr]*, for the original store, denoted by W_1. This means that these two operations are not atomic anymore. To see how this can cause correctness problems, let us consider the example shown Fig. 5.12(a), which shows the same RAW dependency. Let us first assume that the shadow store is performed after the original

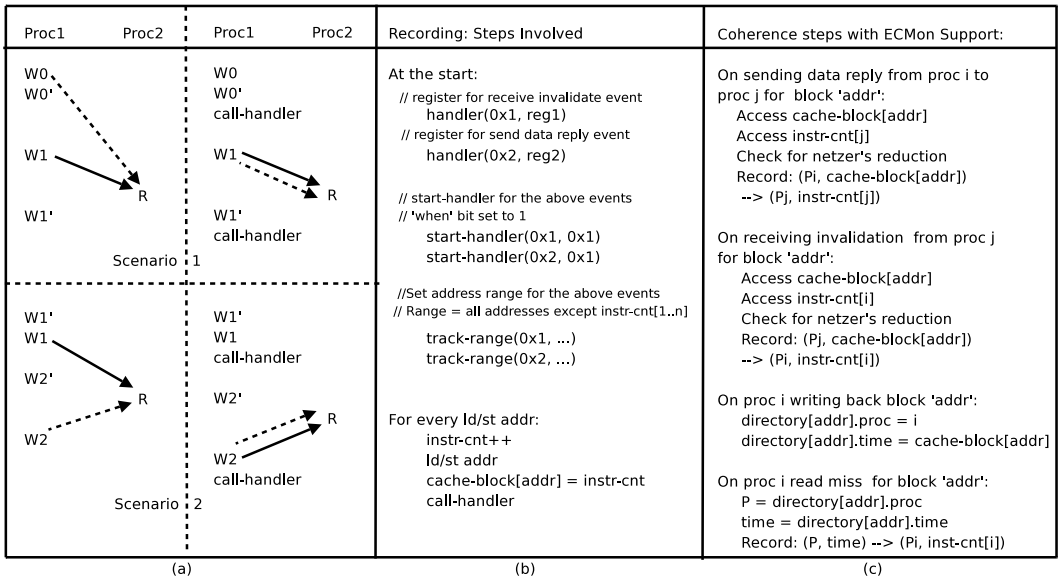


Figure 2.6: (a) Problems due to lack of atomicity and its solution, solid lines represent the exercised dependences, while dotted lines show the recorded dependences (b) Instrumentation involved for recording using ECMon (c) Work done in the software handlers.

store (scenario 1). In the example, let us suppose that the read R from *proc 2* happens before W_1' but after W_1. This implies that *cache-block[addr]* is yet to be updated and contains a stale value. Consequently, $W_0 \rightarrow R$, is recorded instead of $W_1 \rightarrow R$. A similar correctness issue manifests itself, even if the shadow store is performed before the original store, as shown in scenario 2 of Fig. 5.12(a); if read R from *proc 2* happens after the shadow store W_2', but before W_2. This implies that *cache-block[addr]* would have been updated by W_2'; but the read R still gets its value from W_1. Consequently, $W_2 \rightarrow R$, is recorded instead of $W_1 \rightarrow R$. To deal with this issue, the effect of the original and shadow memory instruction executing atomically is simulated via ECMon support. The ECMon facility to control *when* the handler is invoked is used to simulate atomicity. Instead of forcing the handler to be invoked immediately, upon reception of the event, it is invoked at certain key points. More specifically, the `call-handler` instruction is instrumented after every

memory instruction/shadow memory instruction pair W_1/W_1'. This will ensure that if an event is received in between (between W_1 and W_1'), the handler is called only after each of them finish executing. In the above example, Fig. 5.12(a), let us first consider the case where shadow store is performed after original store. When the event is received after W_1 but before W_1', the handler is not called then; it is only called when the `call-handler` is encountered which is after W_1'. This ensures that *cache-block[addr]* is updated and contains the current value. Similarly, in the second scenario, when the event is received after W_2', but before W_2, the handler is not called yet; it is only called after W_2. This means that the data value reply is after W_2 and hence R gets its value from W_2, which is the dependency that is recorded.

Coherence steps with ECMon for correct instruction cnts:

```
On receiving a data reply from proc j
to proc i for block 'addr':
   while (ready(i,j) == 0);
    ready(i,j) = 0;

On sending data reply from proc i to
proc j for  block 'addr':
   ...
   ...
    ready(i,j) =  1;
```

```
On sending invalidation from proc i
shared in 'n' processors for block 'addr':
    while (ready(i,j) !=n);
     ready(i,j) = 0;

On receiving invalidation  from proc j
for block 'addr':
     ...
     ...
     ready(i,j)++;
```

Figure 2.7: Maintaining correct instruction counts

(**Correct Instruction counts**) The second issue concerns the value of *instr-count[2]*, the instruction count of *proc 2* when it is accessed in the handler from *proc 1* for recording the dependency. Recall that *proc 1* is providing a *data value reply* in the first place as *proc 2* experienced a miss due to R. The value of *instr-count[2]* will thus be correct, if it is not changed by a future write, from *proc 2*. To ensure this, *proc 2* is made to wait (using software) until the handler finishes in *proc 1* as shown in Fig. 2.7. To accomplish this

synchronization, a synchronization variable $ready(i, j)$ is maintained for every processor pair i and j. When *proc 2* receives a data reply, a handler is called within which we spin until the $ready(1, 2)$ becomes 1. The variable $ready(1, 2)$ is set to true only when *proc 1* has recorded the dependence and is about to return from the handler. Similarly for WAR dependences we make sure that the processor sending the invalidate, waits until all the processors receiving the invalidate have recorded the dependency.

(**Avoiding nested handler invocations**) It is worth noting that the access of *instr-count[2]* within the handler of *proc 1* will cause a miss, since the variable *instr-count[2]* will be in exclusive state in *proc 2*. Thus, the value of *instr-count[2]* will finally be obtained via a *data value reply* from *proc 2*. However, this *data value reply* should not cause a handler call in *proc 2*. This is because this dependency should not be recorded, since it is not due to the original program. The `track-range` instruction is used to deal with this problem. It is ensured that the range of addresses provided does not include *instr-count[1..n]*. This in turn ensures that the handler is not called for such events.

(**Cache block replacements**) When cache blocks are replaced, the dependences are then exercised through the memory, rather than coherence system. Dependences exercised through memory can be recorded, since ECMon is also able to expose such dependences using the *write-back* and *read-miss* events. First, let us consider the recording of the RAW dependency shown in Fig. 2.5(b). Let us suppose that *proc1* writes back the block to the directory before it is read by *proc 2*. The *write-back* event triggers the software handler, within which the processor id and the instruction count corresponding to the block address, *cache-block[addr]*, is saved into the directory. For this purpose, additional memory is associated with each directory entry in software. Later, when the read occurs in *proc 2*,

it results in a *read-miss*. This triggers the software handler associated with the read-miss within which the RAW dependency is recorded by accessing the information saved in the directory, during the *write-back* event.

(**Thread switches**) When a thread is scheduled out of a processor, the processor can still hold cache blocks accessed by the previous thread. For example, *proc 1* can hold an exclusive block dirtied by *thread 1*, even after *thread 1* is swapped out of *proc 1*. Then later when there is cache miss for the same block in *proc 2* running *thread 2*, the block may be provided by *proc 1* as a data value reply, even though it is currently idle. In this case, the above RAW dependency will not be recorded, since *proc 1* is idle. To handle this case, whenever a thread is scheduled in or out of a processor, the instruction counts of each of the processors are recorded. This dependency, also known as *strata* [60] in prior work, will transitively subsume the missed dependences due to thread switches. The mapping between the thread ids and processor ids is also recorded at the time of a thread switch. Recall that whenever dependences are recorded, the ISMDs between processors are recorded; using this mapping between the processors and threads, the dependences between the threads can then be derived.

2.4.4 Handler Instrumentation

The instrumentation involved for recording ISMDs using ECMon is summarized in Fig. 5.12(b). At the start of the program handlers are registered for two events: *(i) a processor receiving an invalidate message*, (ii) *a processor about to send the data value reply*. The the `start-handler` instruction is added, with the *when* bit set to 1, to handle the atomicity problem. The *track-range* instruction is added, so that addresses involving

accesses to *instr-cnt[1..n]* do not cause handlers to be called. As discussed earlier, *instr-cnt* is incremented and made to update *cache-block[addr]* for every memory instruction. For the purpose of simulating the effect of atomicity, *call-handler* instruction is instrumented after this update. The handlers for the events are summarized in Fig. 5.12(c) and are self-explanatory. The first two events are for the purpose of recording dependences exercised through coherence controller, while the last two events are for the purpose of recording dependences exercised through memory.

2.4.5 Experimental Evaluation

In this section the results of our experimental evaluation of ECMon support in recording ISMDs is discussed. Before presenting the results of the experimental evaluation, the implementation is discussed.

Processor	8 processor, in order
L1 Cache	32 KB 4 way 1 cycle latency
L2 Cache	shared 512 KB 8 way 9 cycle latency
Memory latency	200 cycle latency
Coherence	MOSI Directory

Table 2.3: Architectural Parameters.

Implementation

The ECMon support was implemented in the SESC [80] simulator, targeting the MIPS architecture. The simulator is a cycle accurate multiprocessor (supports multicore mode) simulator which also simulates primary functions of the OS including memory allocation and TLB handling. To implement ISA changes, the unused opcodes of the MIPS instruction were used. The decoder of the simulator was then modified to decode the new

instructions and their semantics were implemented by adding the hardware structures to the simulator. Finally, the ECMon support was implemented for a MOSI based directory protocol for an 8 core processor with a shared L2 cache, which holds the directory entries. In the implementation of the coherence protocol, the L1 cache does not notify the directory on shared cache replacements. The architectural parameters for our implementation are presented in Table 4.14.

Programs	LOC	Input	Description
BARNES	2.0K	8192	Barnes-Hut alg.
FMM	3.2K	256	fast multipole alg.
OCEAN	2.6K	258×258	ocean simulation
RADIOSITY	8.2K	batch	diffuse radiosity alg.
RAYTRACE	6.1K	tea	ray tracing alg.
WATER-NSQ	1.2K	512	nsquared
WATER-SP	1.6K	512	spatial

Table 2.4: SPLASH-2 Benchmarks Description.

The SPLASH [110] benchmark suite (in Table 2.4.5) was chosen to evaluate our ECMon support for helping in recording ISMDs. Instrumentation was performed by modifying the assembler output generated by the gcc-4.1 compiler. The dependences were not output to a file, but maintained in a circular buffer, similar to Bugnet [61]. Time stamps by instrumenting loads and stores with additional instructions that incremented a counter and stored the time stamps in shadow memory. One register was specifically for maintaining these time stamps, so that it need not be spilled and restored for every memory instruction. Finally, the program VOLREND could not be compiled using the compiler infrastructure that targets the simulator and hence was omitted from our experiments.

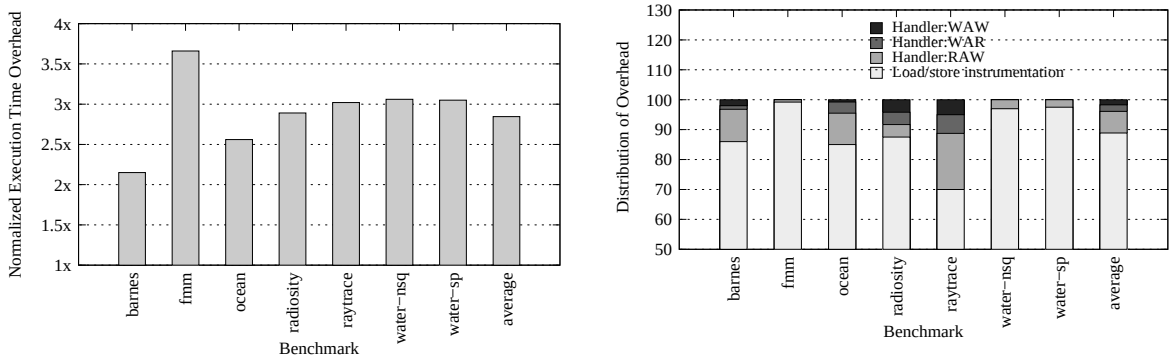


Figure 2.8: Dependence Recording Overhead and Break up of Overheads

Performance: Recording Dependences

In this experiment, the execution time overhead of performing recording in software with ECMon support was measured. As we can see from Fig. 2.8, the overhead for performing recording ranges from 2.2 fold execution time overhead for the *BARNES* benchmark through 3.6 fold overhead for the *FMM* benchmark. On an average (harmonic mean), recording causes a 2.8 fold execution time overhead.

To understand the causes for this overhead, the overhead was split into several contributing categories as shown in Fig. 2.8. The first category is the overhead due to the execution of additional instructions to maintain instruction counts; recall that in the software version we actually needed to instrument loads and stores to maintain these counts. As we can see, this is the major contributing factor of the overhead, accounting for 89% of the overhead on an average. Now the high overhead of the *FMM* can be explained. Since the FMM program has a large percentage of memory instructions (around 50%), significant time was spent to maintain the instruction counts. On the contrary, *BARNES* and *OCEAN* programs, with relatively fewer memory instructions (around 30%), causes lesser overhead.

It is important to note that, since most of time is spent on instrumentation, *only 11%* of the execution time is spent executing the handler recording the ISMDs. This vindicates ECMon's main motivation: efficient support to expose dependences to the software. The time spent in the handler was additionally split into 3 categories: time spent to log RAW, WAR and WAW dependences. As we can see, most of the time (out of 11% time spent recording dependences) is spent recording RAW dependences.

From the above set of experiments, it is clear that with ECMon support, ISMDs can be efficiently recorded at only 2.8 fold execution time overhead. It is worth noting that without ECMon support it would be prohibitively expensive to perform recording; several additional instructions are needed for each memory instruction to derive dependences, in addition to thread serialization to handle the atomicity problems which could result in overhead at least an order of higher magnitude [55, 63].

2.5 Summary

Current software monitoring tools are rendered inapplicable for monitoring parallel programs running on multicores, since they are unable to detect ISMDs efficiently. In this chapter ECMon, support for exposing cache events to the software was proposed. With ECMon support, software is made aware of ISMDs which enables monitoring of parallel programs on multicores. An application of ECMon in which it was used to record ISMDs was discussed. Experimental results showed that ISMDs could be recorded very efficiently, incurring on an average only 3 fold execution time slowdown. By recording ISMDs with ECMon support, current software tools for implementing deterministic replay debugging are rendered applicable for parallel programs running on multicores. In the subsequent

chapters, ECMon support is used to enable several other existing and novel monitoring applications for parallel programs running on multicores.

Chapter 3

Shadow Memory Applications using ECMon

A class of monitoring applications requires the maintenance of information associated with each of the application's original memory location, which is held in corresponding *shadow memory* locations. Consequently, each original memory instruction (OMI) is associated with shadow memory instructions (SMIs) for maintaining the meta data associated with the memory location accessed. The above applications are known as *shadow memory* applications. For multithreaded programs, it is essential that OMIs and the SMIs that accompany them be carried out atomically, in order to correctly maintain shadow values. Existing software monitoring schemes [64, 63] prevent race conditions that can lead to incorrect shadow values by ensuring that a thread switch does not occur in the middle of execution of OMI and its corresponding SMI. Unfortunately, the problem still exists when a multithreaded program is being run on multicores. In this chapter we ensure that OMIs and SMIs are executed atomically even when they are run on multicores, using ECMon

support proposed in the previous chapter. Another important issue in shadow memory design that affects the speed and the robustness of the shadow memory implementation, is the organization of the shadow memory in the address space of the application. Using minimal OS support, in the form of coupled allocation of original and shadow pages, we propose a robust implementation of shadow memory that is also very efficient.

3.1 Overview

There has been significant research on the online monitoring of running programs using various dynamic analyses for a variety of purposes. For example, *LIFT* [73] and *TaintCheck* [66] are software tools that perform taint analysis to ensure the execution of a program is not compromised by harmful inputs; *Memcheck* [63] is a popular memory checking tool that is widely used to detect memory bugs; and *Eraser* [88] is a tool for detecting data races. A common element among these tools is that they make use of *shadow memory* [63]. With each memory location used by the application, a *shadow* memory location is associated to store information about that memory location. Original instructions in the application that manipulate memory locations are accompanied by instructions that manipulate corresponding shadow memory locations. For example, in taint analysis, with every memory location a *taint* value is associated that indicates whether that memory location is data dependent on an (tainted) input. Each original instruction that stores the value of a register into a memory location is accompanied by an additional store that moves the taint value of the register into the shadow memory location. Similarly each original instruction that loads a value from a memory location to a register is accompanied by an instruction that loads the corresponding taint value from shadow memory location. Thus,

monitoring requires that loads and stores present in an application be accompanied by shadow memory loads and stores.

Although the need for shadow memory support across variety of monitoring tasks is well recognized, supporting robust shadow memory that can be efficiently accessed and manipulated remains a challenge that has not been successfully addressed. There are two key issues at the heart of this challenge:

Atomic Updates For multithreaded programs, it is essential that original memory instructions (OMIs) and the shadow memory instructions (SMIs) accompanying them be carried out atomically in order to correctly maintain the shadow values. Since OMIs and SMIs are really separate instructions, maintaining atomicity incurs an additional cost. Existing software monitoring schemes [64, 63] prevent race conditions that can lead to incorrect shadow values by ensuring that a thread switch does not occur in the middle of execution of OMI and its corresponding SMI. Unfortunately, the problem still exists when a multithreaded program is being run on, the now ubiquitous, multicores. To overcome this problem of concurrent updates on multicores, threads can be *serialized* and made to run on one core [64]. However, this is clearly inefficient as parallelism is sacrificed. Alternatively, in the *fine grain locking* approach, the thread that wants to perform a SMI along with the OMI, grabs a lock associated with that memory region and releases the lock after completion. However, this approach suffers from the overhead of executing additional instructions including the expensive atomic instructions.

Shadow Memory Management An important issue in shadow memory design, that affects the speed and the robustness of the shadow memory implementation, is the organization of

the shadow memory in the address space of the application process [63]. A simple *half-and-half* scheme [15, 73] roughly divides the virtual memory into two halves, the original memory and the corresponding shadow memory. While this has the advantage of a fast translation of original addresses into corresponding shadow memory addresses, its less flexible layout means that it fails for some programs in linux and is incompatible with operating systems with restrictive layouts [63]. Moreover, it does not scale when we need to associate more than one shadow value per memory location. To improve robustness, Valgrind's *Memcheck* tool [63] implements a two-level page table in software. Although, several optimizations are proposed, the slowdown can still be as high as 22x for *SPEC* programs, about half of which may be due to shadow memory accesses [63].

In this chapter, a robust shadow memory implementation for multicores that addresses the above challenges of *atomic updates* and *efficient address translation* is presented. Our shadow memory implementation uses the ECMon support to ensure atomicity and proposes OS support for a robust shadow memory implementation that ensures efficient address translation.

3.2 Runtime Monitoring: Applications and Costs

Runtime monitoring serves as a foundation of a variety of tasks aimed at providing security, performing debugging, and improving performance of applications. In this section the role of shadow memory in context of four popular monitoring tasks. In addition, the execution time overheads of runtime monitoring, as well as degree to which various factors contribute to this overhead are analyzed.

Table 3.1 describes four popular monitoring tasks: DIFT [66] for runtime mon-

Monitoring Application	Meta Data Tracked by Shadow Memory	Code Instrumentation Required
DIFT [73, 66, 15] (Dynamic Information Flow Tracking) is used to track whether contents of memory locations are data dependent upon insecure inputs.	With each memory location (byte) a *taint* bit is associated, which indicates whether that memory location is data dependent upon an insecure input. Consequently, the taint bit has to be manipulated for every memory instruction.	(Loads) For every load, the taint bit corresponding to the loaded memory location has to be read; (Stores) For every store, the taint bit corresponding to the stored memory location has to be updated.
Eraser [88] is used to track information to enable data race detection.	With every memory word Eraser associates the *status* and the *lockset*. The *status* tells if the current word is shared across threads or exclusive to one thread, while the *lockset* indicates the set of locks used to access that memory location.	(Loads/Stores) Each memory access, either by a load or a store, must be accompanied with reading and writing of both *status* and *lock-set*.
Memcheck [63] is used for debugging memory bugs.	Every location is associated with two values, the *A bit* and the *V bits*. While the *A bit* indicates if that particular memory location is addressable, the *V bits* indicate whether the corresponding bits in the memory location have been defined.	(Loads) The *A bit* is read and updated while *V bits* are read on every load; (Stores) The *A bit* and the *V bits* are read and updated on every store.
MemProfile [8] is a simple memory profiler that keeps count of number of reads and writes to each memory location.	With each memory location two counts are associated: *ReadCount* and *WriteCount*.	(Loads) The *ReadCount* is read and updated on every load; (Stores) The *WriteCount* is read and updated on every store.

Table 3.1: Applications Requiring Runtime Monitoring.

itoring of software attacks, Memcheck [63] a tool for runtime checking of memory errors, Eraser [88] for runtime detection of data races, and Memprofile a runtime memory profiler [8]. Each of these monitoring tasks require the following:

- With each data memory location, shadow memory location(s) are associated to track the meta data required by the monitoring task. The second column of Table 3.1 describes the meta data maintained by these applications. The number of distinct items of information to be associated with a memory location can vary. While *DIFT* associates just one value, the taint bit, for every memory location, *Eraser* and *Memcheck* associate two values per memory location. Thus, in general, capability of associating *multiple* shadow values for every memory location is needed.

- Application code must be instrumented by associating operations for maintaining the meta data with the memory operations (loads and stores) in the application. The third column of Table 3.1 describes the function of shadow memory instructions (SMIs) that instrument each original memory instruction (OMI) for each of the monitoring tasks.

- An OMI and its associated SMI(s) must be performed *atomically*. For example, if during DIFT a value in an original memory location and its taint bit are read, atomicity must guarantee that the taint bit corresponds to the value read from the original memory location and not to some old value that once resided in the memory location. Note that the SMI in *DIFT* is *symmetric*, i.e. for every original *load* there is an associated *shadow load* and for every original *store*, there is a *shadow store*. However, in *general*, for every original memory access (*load, store*), the associated *shadow memory* may need to be both *read* and *updated*. In fact this is the case for *Eraser*.

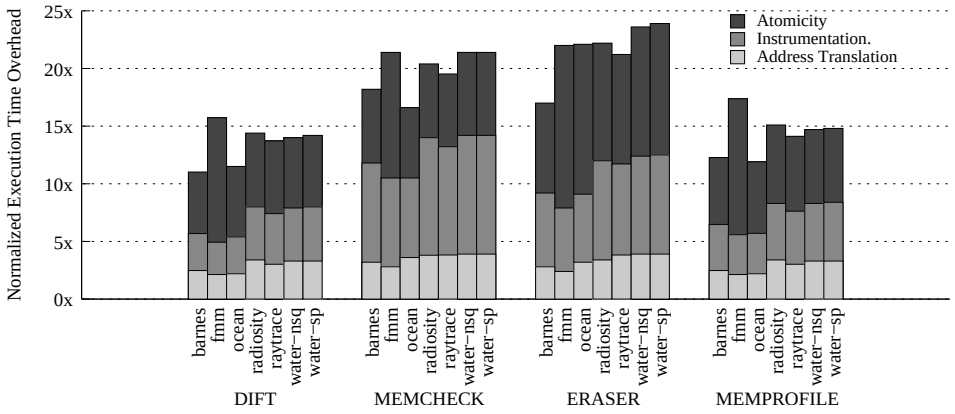


Figure 3.1: Overhead Imposed by Current Shadow Memory Tools.

To get an idea of the performance overhead imposed by the current shadow memory tools, the overhead of performing the above monitoring tasks for the SPLASH [110] benchmarks on a 4 core processor was measured. As shown in Fig. 3.1, the runtime overhead of monitoring was broken down into three components: the overhead for performing address translation, overhead for maintaining atomicity of OMIs and SMIs, and finally the overhead due to execution of instrumentation code required by the monitoring task. As we can see, the overhead in performing the monitoring tasks can be as high as 25x, with a significant percentage of overhead (about 50%) spent in performing address translation and enforcing atomicity.

The goal of OASES is to reduce the runtime overhead of monitoring tasks. For this purpose, we focus on providing support that reduces the overhead due to the two components that are common to all the monitoring tasks, i.e. address translation of shadow memory references and enforcing atomicity of OMIs and SMIs. The third component, code instrumentation, varies from one monitoring task to another. Thus, the ability to program the instrumentation to accommodate the requirements of different monitoring tasks must

be maintained. We do not provide any specialized hardware support to reduce the cost of executing the instrumentation code as different monitoring applications will require different hardware support.

3.3 Shadow Memory Design for Multicores

This sections begin with a brief overview of the approach for efficiently enforcing atomicity and performing address translation. Then in subsequent sections the solutions are presented in full detail. First, the problem of performing atomic updates of original memory locations and corresponding shadow memory locations is considered.

Atomic Updates. First let us see why an OMI and its SMI(s) must be performed atomically. Consider the example shown in Fig. 3.2. Processor A executes two store instructions (St_1 and St_2) and their corresponding shadow store instructions (SSt_1 and SSt_2) while Processor B executes a load instruction Ld and its corresponding shadow load SLd. We assume that all these instructions target the same virtual address. As we can see in Fig. 3.2a, if no special care is taken, Ld in Processor B may see the value produced by St_1 while SLd may see a value produced at SSt_2. Atomic SMIs will guarantee that Ld and SLd see either values produced by (St_1, SSt_1) or (St_2,SSt_2) as shown in Fig. 3.2b. Prior solutions have used thread serialization or fine grained locking to ensure atomicity of OMIs and SMIs. However, they are inefficient as we saw earlier: while thread serialization is clearly inefficient since it compromises on concurrency, the expensive atomic instructions and memory fences involved in locking are also inefficient. Although the above example illustrates a scenario in which there is a race in the original program, the same problem can manifest itself even

if the original program is devoid of races; the introduction of SM can break the assumed atomicity of instructions such as compare-and-swap which are used to implement a variety of synchronization primitives and lock-free data structures. [16]

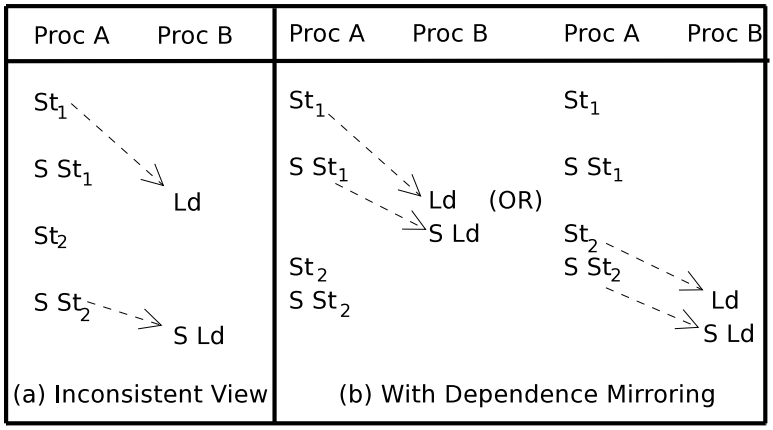


Figure 3.2: Atomic Updates of Shadow Memory.

The solution for enforcing atomicity is based upon the following key observations. First, given a memory location and a corresponding shadow value, we must maintain multiple memory locations for this shadow value. More specifically, a distinct shadow location must be provided for each distinct place where the shadow value can reside, i.e. corresponding to each processor's cache we must provide a shadow memory location and corresponding to the memory we must provide a shadow memory location. Second, we must provide a protocol for updating the shadow values in a manner that guarantees atomicity. We name this protocol as the *Coupled Shadow Coherence* (CSC) protocol because it couples the coherence of shadow values with coherence actions of the original values to achieve the effect of atomicity.

The need for maintaining multiple shadow locations to implement a single shadow value and the requirements placed on the CSC protocol for maintaining these shadow loca-

tions are illustrated by two scenarios shown in Fig. 3.3. Let us consider the first scenario, Fig. 3.3(a), in which the *Ld* and *SLd* performed by processor B must access values v and v' respectively. However, the execution of *Ld* and *SLd* at Processor B is intervened by execution of a *St* and *SSt* at Processor A that update values v and v' to w and w' respectively. The contents of the memory location and the two corresponding shadow locations for the two processors are shown in the figure as the execution proceeds. It should be noted that to guarantee atomicity of *Ld* and *SLd* at Processor B, the following must be done. After the execution of *SSt* at Processor A, although the contents of shadow location for Processor A are changed to w', the contents of the shadow location for Processor B must remain unchanged as v' till v' has been read by *SLd* at Processor B. While this scenario shows that an update of a shadow location may need to be delayed till a *SLd* had been executed, the second scenario in Fig. 3.3b shows the reverse situation, i.e. the execution of *SLd* must be stalled till the shadow location has been updated.

In the first scenario, there is a period of time during which the shadow values at the two processors must be different. This justifies the need for separate shadow locations for the two processors. The requirements of delaying the update of a shadow location (first scenario) and waiting for the update of a shadow location (second scenario) must be enforced by the CSC protocol that will be implemented in software. For example, in the first scenario, following Processor B's execution of its *SLd* operation, any future references by Processor B to the memory location and its shadow location should result in the delivery of values w and w' respectively. While delivery of w is guaranteed by the hardware cache coherence mechanism, the delivery of w' requires that this value be copied from the shadow location for Processor A to the shadow location for Processor B. The CSC protocol will be

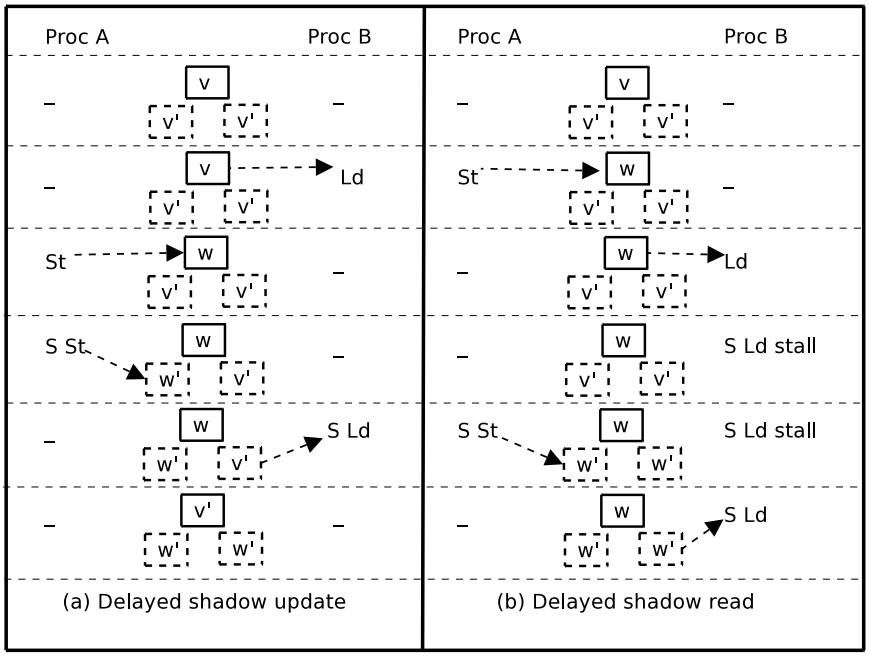


Figure 3.3: Timing of Shadow Value Updates.

responsible for ensuring that this copying operation is performed. Similarly, in the second scenario the CSC protocol will cause the execution of *SLd* to stall till it is able to copy the value w' from shadow location for Processor A to shadow location for Processor B.

The actions performed by the CSC protocol to maintain the consistency of shadow memory locations are coupled with the actions performed by the cache coherence protocol to maintain the consistency of the memory locations cached at various processors. The example in Fig. 3.4 shows how cache events trigger corresponding CSC actions. To implement the CSC protocol, ECMon support for exposing the cache events to the software, is utilized. Whenever the cache controller of a processor receives a cache-coherence event, such as a data value reply, it interrupts the processor and passes the control to the CSC protocol. The CSC protocol is implemented as a sent of handler functions – one handler for each distinct cache coherence event. One key aspect of the CSC protocol is that there are no

changes to the original hardware cache coherence protocol.

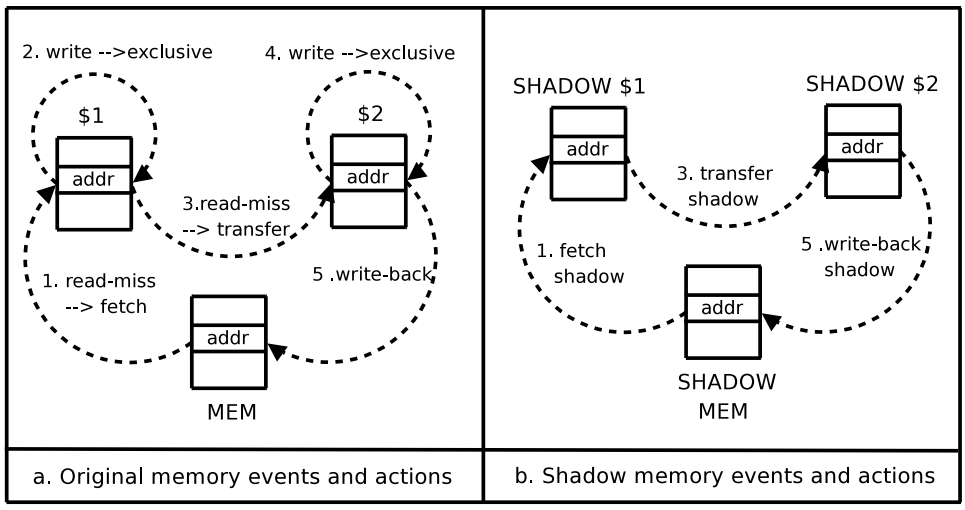


Figure 3.4: Coupled Shadow Coherence.

Efficient Address Translation. The process of addressing shadow memory needs to be both robust and efficient. The same virtual address is used to reference both an original memory location and the corresponding shadow memory location. During translation to physical addresses, different physical addresses are produced for the original and SMIs referring to the same virtual address. In particular, for every original page there are corresponding shadow memory pages and during page translation virtual page is translated to different appropriate physical pages. This approach is robust as unlike the *half-and-half* strategy it does not require an application to reserve half of its virtual address space for shadow memory. To enable efficient translation of original memory addresses into shadow memory addresses the following approach is followed. A page of memory belonging to the application and the corresponding shadow memory pages are all allocated consecutive physical memory pages. Thus, from the address of a original memory location, the address of corresponding

shadow memory locations can be efficiently computed. Furthermore, it is ensured that at any point in time if an original memory page resides in main memory then the corresponding shadow memory pages also resides in main memory. Thus, while page table entries are created for original memory pages, no additional page table entries are required for the corresponding shadow memory pages.

In the remainder of this section, the detailed design and implementation of the solutions outlined above are described. First the instruction set support for identifying memory instructions that must be executed atomically, as well as distinguishing an OMI from its SMIs is described. Next the details of the OS and architectural support for efficient translation of original memory addresses to shadow memory addresses is presented. Finally, the details of the CSC protocol that ensures atomic updates of original memory locations and corresponding shadow memory locations, is described.

3.3.1 Instruction Set Support

Instruction set support is needed for two purposes. First, since each OMI and all of its SMIs must execute atomically, a mechanism for identifying them as an *atomic block* is needed. Second, since the same virtual address is specified in addressing a memory location and its corresponding shadow locations, for correct address translation there is a need to provide a means for *distinguishing* the OMI and SMIs for various shadow values. Two new instructions that simultaneously meet the above requirements are added. As shown below, the two new instructions, `shadow-start` and `shadow-end`, are used to define an atomic block. The operands of the `shadow-start` instruction, *init-SVC* and *pid*, allow us to distinguish between OMI and SMIs.

```
shadow-start init-SVC, pid
...
shadow-end
```

The *pid* operand identifies the processor id of the processor whose copy of a shadow value is to be accessed. The *pid* operand is an optional operand. If no value is specified as the processor id, the processor id is implicitly assumed to be the current processor's processor id. The operand *init-SVC* enables us to distinguish between the OMI and various SMIs within an atomic block. All memory instructions in the atomic block that access the same virtual address as the OMI are recognized as SMIs. If *init-SVC* is specified as 0, the first memory operation in the atomic block is treated as the OMI and subsequent memory operations that access the same virtual address are treated as SMIs. Moreover, the second memory operation refers to the first shadow value, the third memory operation refers to the second shadow value and so on. However, a non zero *init-SVC* is used to handle situations in which only shadow values need to be accessed without the accessing original values. For example, if *init-SVC* is 1, the first memory access refers to the first shadow value and so on. In other words, *initSVC* is specified as a parameter to give us additional flexibility in accessing the shadow values. It should be noted that there is an implicit assumption that multiple shadow reads (writes) correspond to different shadow values. This is done since each shadow memory location is read and written once in an atomic block. It is not necessary to explicitly read (or write) to the same shadow memory location more than once inside the atomic block – the shadow memory value can be copied on to the stack, manipulated and then copied back.

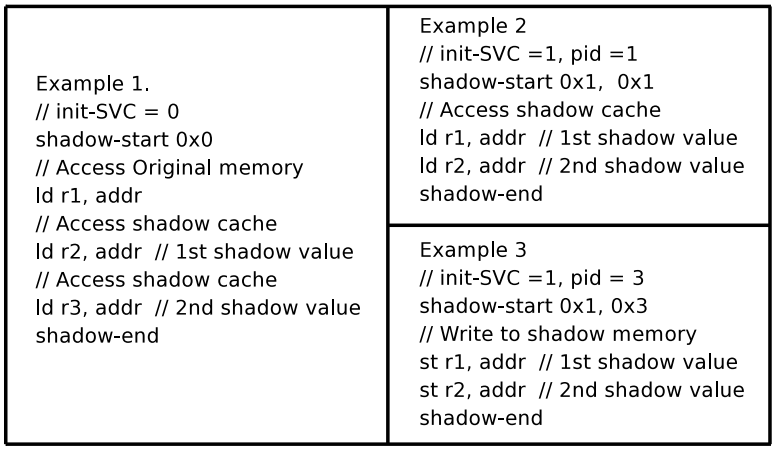


Figure 3.5: Some Code Sequences for Accessing Shadow Values.

Given the above interpretation of *init-SVC*, the compiler must generate instructions within an atomic block in the appropriate order. Fig. 3.5 shows some examples that show how the compiler generates code for accessing various shadow values. For the purpose of this example, let us assume that there are 2 processors with 2 shadow values. The first scenario shows the inlined instrumentation for accessing both original and shadow memory values. Since original memory values are involved, the value of *init-SVC* is set to 0, specified as an operand to *shadow-start* instruction. Accordingly, the first memory access is an original memory access and subsequent accesses are for shadow values. Since the *shadow-start* instruction does not specify any *pid* operands, the current processor id is used in the translation process and so the shadow cache contents of the current processor are accessed. The second scenario shows code generated for the handler. The purpose of this handler is to read the contents of shadow cache of processor 1 and write it to shadow memory. Accordingly the first two loads access the shadow cache contents of processor 1. To enable these accesses, *init-SVC* is set to 1 through the `shadow-start` instruction; this is because

there are no original memory accesses involved. Furthermore, by the specifying the *pid* as 1, the shadow cache contents of processor 1 are accessed. Finally, in the last example, the two stores are made to write to the shadow memory contents. This is enabled by specifying the *pid* as 3; since there are only, 2 processors, a pid of 3 denotes shadow memory.

3.3.2 Address Translation for Shadow Accesses

Since the same virtual address is used by the OMI and the corresponding SMIs, an address translation scheme that efficiently translates the virtual address used by SMIs into appropriate physical addresses of shadow locations must be implemented. To ensure that the translation can be performed efficiently, a page layout scheme that fixes the relative location of an original physical page and its corresponding shadow physical pages is used. For every original page, the OS allocates $p + 1$ shadow pages per shadow value, where p is the number of processors. Therefore, if there are n shadow values, the processor allocates $n \times (p + 1)$ shadow pages. Moreover, consecutive set of physical pages are allocated by the OS. Thus, given the physical address of an original memory location, the physical addresses of the various associated shadow values can be easily determined. Given the values of SVC (shadow value count), the *pid*, and N the number of processors, address translation proceeds by multiplying N with the *pid* and adding the result with SVC. The resultant is added to the physical page fetched from the TLB, if it is a shadow memory access (SVC is non zero); if it is an original memory access, the resultant is 0, since the value of SVC is 0 and hence the fetched page from TLB is used. The above page layout and addressing scheme is illustrated in Fig. 3.6 for the scenario where there are 2 processors and 2 shadow values. As we can see, the first page denotes the original page, while the rest denote shadow pages.

The second and third pages denote the shadow cache pages of the first processor, while the fourth and fifth denote the shadow cache pages of the second processor, and finally the last two pages refer to the shadow memory pages.

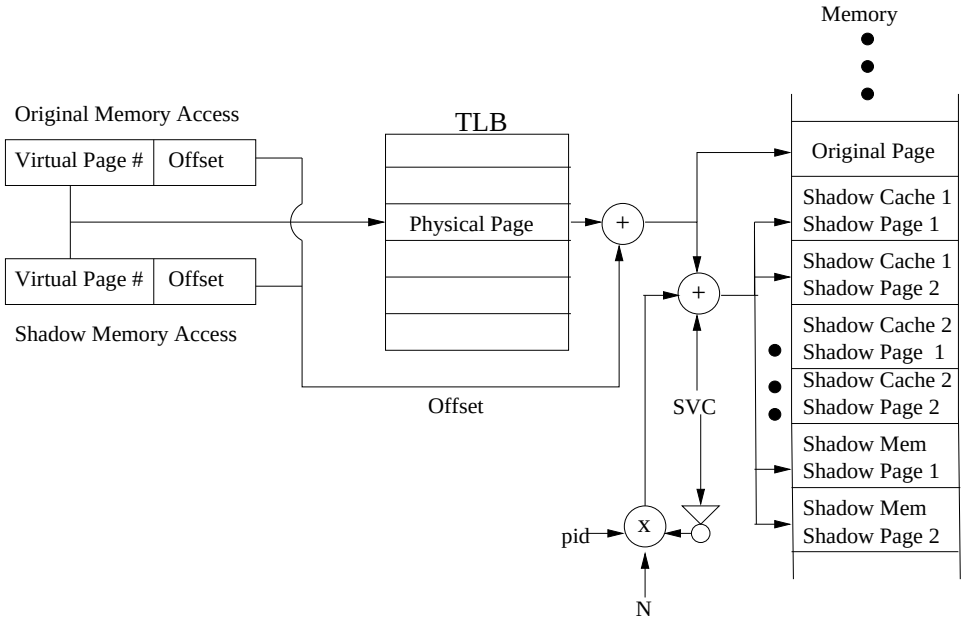


Figure 3.6: Address Translation.

The OS treats every original memory page and its corresponding shadow pages as a single entity. When the OS decides to swap out an original page on to the disk, it also swaps out the associated shadow pages. Similarly, both original page and its associated shadow pages are swapped in together. The above translation process is highly efficient. Another important consequence of this scheme is that shadow memory does not require any additional TLB entries. Finally, since an application may not require monitoring, an extra flag is added to the *process descriptor* which indicates whether that particular process requires shadow memory support. When this flag is set, the OS allocates shadow page(s) along with every original page that it allocates; otherwise no shadow pages are allocated.

Given the manner in which code within atomic blocks is organized, is then shown how this organization can be used to generate the *Shadow Value Count* (SVC) needed for address translation in Fig. 3.6. The state machine in Fig. 3.7 generates the value of *SVC*. The state machine is in initial state "Outside Atomic Block" and when `shadow-start` is encountered it moves to state "Inside Atomic Block" initializing *SVC* to *init-SVC* the value specified as an operand to the `shadow-start` instruction. For now, let us assume that the value of *init-SVC* is a 0, which means the first memory instruction encountered refers to the OMI. When the OMI (load or store) is encountered – the virtual address is remembered in *vaddr*; counts *LoadSVC* and *StoreSVC* are set to *initSVC*; and transition to state "Inside Instrumentation Code" takes place. In this state when a shadow load (store) is encountered, *LoadSVC* (*StoreSVC*) is incremented and its value is assigned to *SVC* for use by address translation logic. If `shadow-end` is encountered, transition to initial state "Outside Atomic Block" occurs.

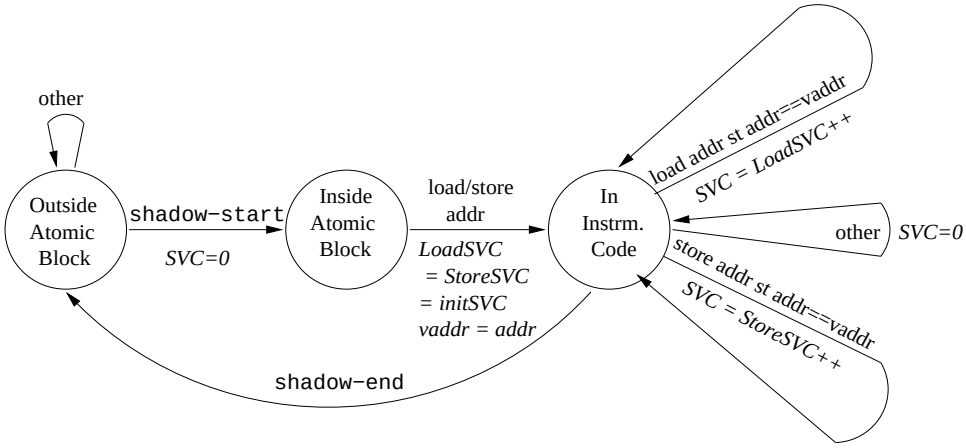


Figure 3.7: Generating Shadow Value Count.

Small Sized Shadow Values. In the above discussion it is assumed that each mem-

ory location used by an application requires equal sized shadow values. For some monitoring tasks, each word of original application does not require an equal size shadow value. For example, in DIFT each memory byte is associated with only a shadow bit. Association of a byte of shadow value with every byte of original application, in this instance, will lead to wastage of memory. It is also possible to extend the scheme to support small-sized shadow values as discussed in [53].

Optimizing Shadow Cache Organization. The memory overhead of maintaining shadow cache can be reduced. This is based on the simple observation that the cache can only hold a fixed amount of data and so the size of the shadow cache can be limited. Thus one way of organizing shadow cache is to reserve a small portion of the virtual memory for the shadow cache. For example, for an L1 cache of size 32KB with 8 processors and 4 shadow values, it is sufficient to allocate 1MB of virtual memory for shadow cache. However, such a scheme will only be applicable for a direct mapped cache; otherwise tag checks that are performed in parallel in hardware will have to performed in software, which can be very expensive.

3.3.3 Atomic Updates of Shadow Memory

In a multithreaded application, an OMI and its corresponding SMI(s) need to be executed atomically. In this section, the CSC protocol to ensure atomicity is first described. Then the software implementation of CSC protocol using ECMon is discussed.

In CSC, the coherence of the shadow memory values is *coupled* with the coherence of the original memory. In particular, to achieve atomicity, the CSC scheme ensures *dependence mirroring* between OMI and SMIs; dependences exercised among SMIs are made

to *mirror* the dependences exercised among OMIs. Let M_1 and M_2 denote a pair of OMIs and SM_1 and SM_2 denote their corresponding SMIs. If M_2 is dependent (e.g., RAW) upon M_1 during an execution, then SM_2 must be similarly dependent upon SM_1. To enforce dependence mirroring, it is ensured that whenever there is a transfer of original memory values from one local cache to memory (or another local cache), it is accompanied by a corresponding *co-transfer* of shadow memory values. To implement this in software, the ECMon support is utilized. Recall that in ECMon, whenever the cache controller receives a pre-specified cache event for a processor's local cache, it can be programmed to interrupt the processor, and call a predefined *handler* function; by suitably programming the handler, dependence mirroring can be enforced in software.

Events. To implement CSC scheme in software the following specific cache events are utilized:

- When a processor exclusively holding a block, is about to send the data value reply
- When a processor receives a data value reply
- When a processor experiences a read miss for a block uncached in any of the processors
- When a processor is about to write back a block

The first two events capture dependences that are exercised through cache coherence network, while the last two events capture dependences exercised via the main memory.

Handler Semantics. The coherence controller interrupts the processor and calls the handler function, when one of the specified events takes place. When the specified event occurs, the current instructions in the pipeline are flushed and a call to handler function is made at once. However, if the processor is in the midst of executing an atomic

block (between `shadow-start` and `shadow-end`), then the call to the handler is delayed until the atomic block is fully executed (`shadow-end` instruction commits). In this regard, the semantics of the `shadow-end` instruction is the same as the `call-handler` instruction introduced in chapter 2.

State	Purpose
shadow (addr)	Shadows the original address 'addr'
shadow-cache (p, addr)	Proc p Shadow cache contents of address 'addr'
ready (i, j)	Synchronizing procs 'i' and 'j' for implementing co-transfer b/w shadow caches
shadow-event-cnt	Ensuring that co-transfer b/w shadow memory and shadow cache takes place along with the original transfer.

Figure 3.8: State Maintained to Implement CSC.

State Maintained for Implementing CSC. Since CSC protocol is implemented in software, shadow coherence state must be maintained in software as shown in Fig. 3.8. Every original memory block, *addr*, that is present in the local cache of each processor p is shadowed by *shadow-cache(p, addr)*; likewise, each original memory address in the main memory is shadowed by *shadow(addr)*. When original memory dependences are enforced via the coherence network, enforcing the dependences of shadow values entails that the two processors involved in the dependency synchronize with each other. For achieving this pair-wise synchronization, a flag value is maintained for each processor pair (i, j) which is referred to as $ready(i, j)$. When dependences are enforced through the memory, it is ensured that co-transfer of shadow values to and from shadow memory, take place in the same order of the transfer of original values to and from shadow memory. For achieving this, a count referred to as *shadow-event-cnt* is maintained to uniquely identify each memory event.

Handlers for CSC. We now explain how the individual steps involved in CSC scheme are implemented within the software handlers, which is shown in Fig. 3.9. For this discussion, macros for reading and writing into the shadow memory (steps 48 through 55) and shadow cache (steps 31 through 45) are assumed.

(*Co-transfer through coherence.*) Whenever processor i receives a data reply from processor j, processor i is interrupted and the handler is called. Within the handler, the corresponding shadow block is copied from processor j to processor i and thus co-transfer is implemented in software. However, it needs to be ensured that the value in the shadow block copied is consistent. A situation may arise where the shadow block from processor j is yet to update the shadow block (it is in the midst of an atomic block), when it receives a request for the shadow block, as shown in Fig. 3.10(b). Likewise, processor j may have updated the shadow block by a later SMI, by the time it is copied, as shown in Fig. 3.10(a). To deal with these scenarios, it is ensured that processors *synchronize with each other* before co-transfer is performed. To implement this synchronization, whenever a processor receives a request for data reply, it calls a handler (*data-reply-request*). Accordingly, processor j is interrupted and the handler is called when it first receives a request for data reply. If the processor j is in the midst of executing an atomic block, the calling of the handler is *delayed* until the atomic block is completed; this avoids the problem shown in Fig. 3.10(b). Within the handler, the *ready(i,j)* flag is set to be true, meaning that the shadow block is now ready to be copied (step 3-6). Likewise, when processor i receives a data reply and calls the handler, we spin and wait for the *ready(i,j)* flag to be true. Once it becomes true, we proceed with the copying to accomplish the actual co-transfer (step 10-14). Fig. 3.11 shows the CSC actions performed in software for the delayed shadow read scenario.

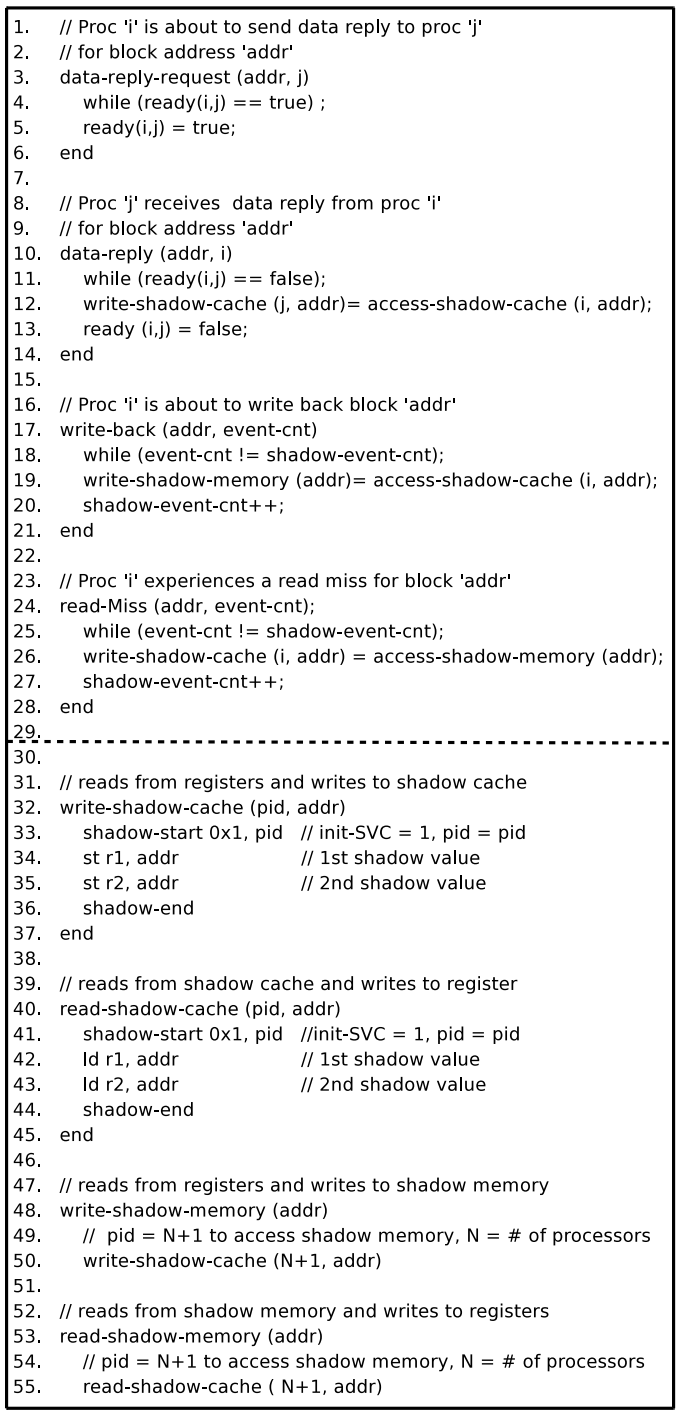


Figure 3.9: Handlers for Various Cache Events.

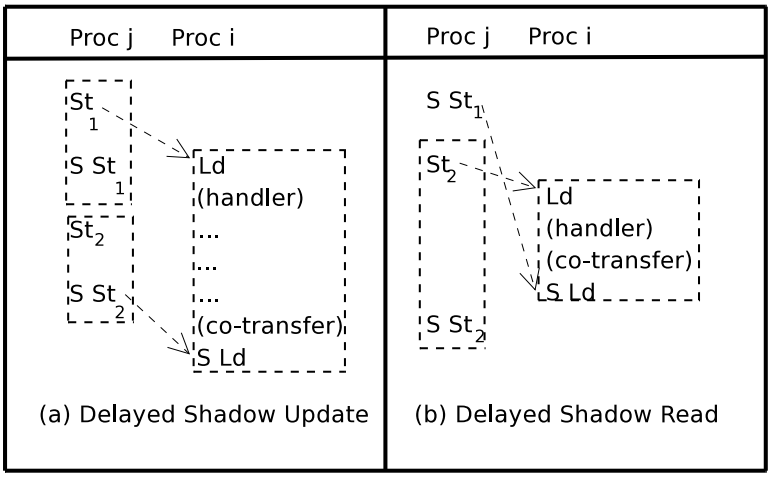


Figure 3.10: Co-transfer Pathological Scenarios.

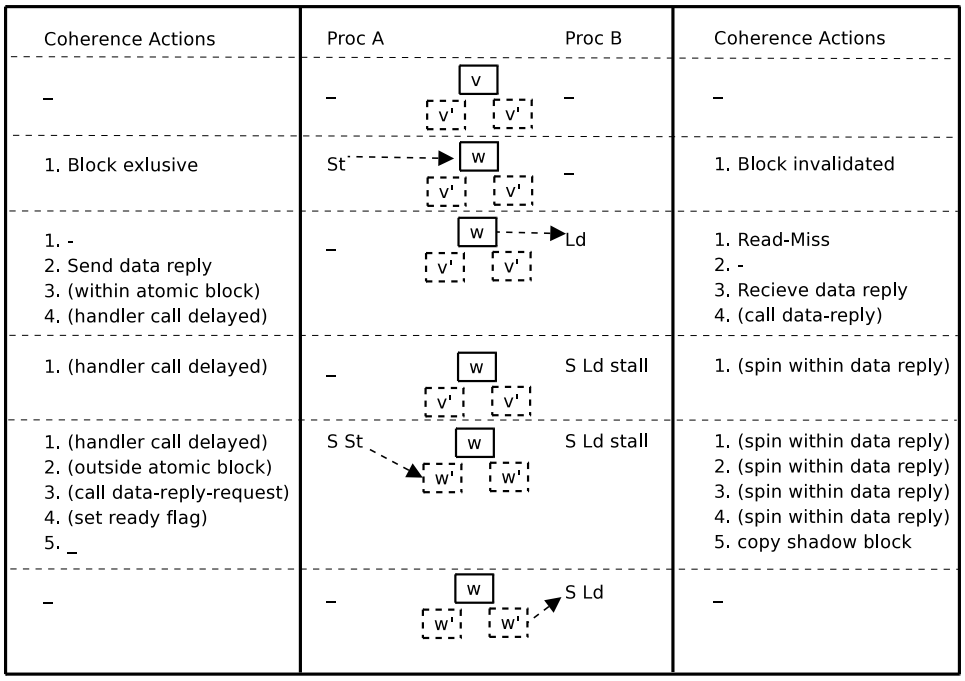


Figure 3.11: Cache Events and CSC Actions.

(*Co-transfer through memory.*) In the CSC scheme, is needs to be ensured that shadow blocks are brought in (and written back) from the memory along with original blocks. To implement this, the *read miss* and *writeback* events are exposed to the software. Whenever, the cache controller performs a write back of an original block, it interrupts the processor and calls a handler. Within the handler the shadow cache contents of the original block is copied to the shadow memory (step 19). Likewise, whenever the cache controller fetches an original block from the main memory, handler is called and within the handler the shadow block is copied from the shadow memory to the shadow cache (step 26). However, it needs to be ensured that the shadow transfers to and from the memory, *take place in the order of the original transfers.* To this end, the coherence controller maintains a *global count* of the total number of *write-back* and *read-miss* events in the *event-cnt* counter. For every *write-back* or *read-miss* event, it increments *event-cnt* count by one and passes it as a parameter to the handler. The handlers in turn maintains a *shadow-event-cnt* counter in software which is incremented by one just before returning from the handler (step 20 and step 27). Additionally, at the start of the handler the value of the *shadow-event-cnt* counter is compared with the *event-cnt* counter that is passed as a parameter (step 18, step 25); *a value match guarantees that all prior handlers have completed executing* and thus ensures that handlers are executed in the order of the original memory transfers.

Preventing Nested Handler Invocations. Nested handler invocations are prevented by ensuring that only OMIs (those which have SVC = 0) inside `shadow-start` and `shadow-end` can cause handler invocations. It should be noted that the handler code does not involve OMIs within `shadow-start` and `shadow-end`. Thus, no nested handler invocations can occur.

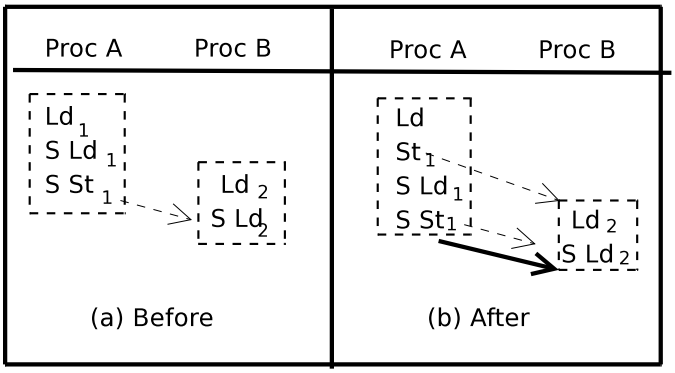


Figure 3.12: Transformation to Handle General SMIs.

Handling general SMIs. Let us discuss how with general SMIs are dealt with, where an original memory load is accompanied by both shadow loads and shadow stores. As we can see from Fig. 3.12(a), the load Ld_1 from processor A is accompanied by shadow load SLd_1 and a shadow store SSt_1. Intuitively, the shadow load from processor B, SLd_2, needs to get its value from the shadow store SSt_1. However, since each of the OMI are loads, there is no transfer of original blocks through the coherence network, which in turn means that co-transfer of shadow blocks is not possible. To enable co-transfer, the original load is converted into a load followed by a (silent) store that writes the same loaded value back to the memory, as shown in Fig. 3.12(b). This will mean that St_1 will invalidate the original block in processor 2 and the original block will be in exclusive state in processor 1. Consequently, Ld_2 will get its value from St_1 through coherence network. This will enable co-transfer and so the shadow load SLd_2 will get its value from SSt_1. It is important to note that this transformation (of loads into a load and a silent store) is not required if the monitoring tool uses symmetric SMIs.

3.4 Experimental Evaluation

In this section, the experimental evaluation of the shadow memory support is discussed is detail. But before that, the implementation details are discussed.

3.4.1 Implementation

The shadow memory support including the OS support and support for ECMon: exposed cache events in the SESC [80] simulator, targeting the MIPS architecture. The simulator is a cycle accurate multicore simulator which also simulates primary functions of the OS including memory allocation and TLB handling. To implement ISA changes, the unused opcodes of the MIPS instruction set were used to implement the newly added instructions. Then the decoder of the simulator was modified to decode the new instructions and implemented their semantics by adding the hardware structures to the simulator. The address translation support was implemented by modifying the OS page allocation algorithm to allocate additionally the shadow pages along with the original pages. The page replacement algorithm was also modified, so that original and shadow pages are considered as a single entity, and replaced together. Finally, the ECMon support was implemented for an invalidate based snooping protocol for a multicore architecture with shared L2 cache. The architectural parameters for our implementation are presented in Table. 3.2. The shadow

Processor	4 processor, out of order
L1 Cache	64 KB 4 way 1 cycle latency
L2 Cache	shared 1024 KB 8 way 9 cycle latency
Memory	4 GB, 500 cycle latency
Coherence	Bus based invalidate

Table 3.2: Architectural Parameters.

memory support was evaluated with four monitoring/profiling applications viz. DIFT[73], Memcheck[63], Eraser[88] and MemProfile[8]. For implementing DIFT, a byte of shadow value kept track of the taintedness for every word of original memory word. The system calls (that were emulated by the simulator) were modified to initialize the taint values. Eraser is a tool for identifying data races. The first part of the algorithm which characterizes each memory word as *virgin*, *exclusive*, *shared* or *shared-modified* was implemented. The second part of the algorithm that then uses this information to maintain the locksets was not implemented. With each memory word, two bytes of information: one byte for maintaining the above four states, and another byte for maintaining the thread-id of the thread that last accessed that memory location was associated. Memcheck-lite, a version of Memcheck in which the register level V-bits propagation is not implemented, was considered. A version that has been optimized for word based memory operations was implemented. For implementing MemProfile, two words of data along with each original memory word was associated, used for maintaining the number of reads and writes to that memory word.

Instrumentation was performed by modifying the assembler output generated by the gcc-4.1 compiler. The SPLASH-2 [110], a standard multithreaded suite (Table 3.4.1) of benchmarks was used for our evaluation. The VOLREND program could not be compiled using the compiler infrastructure that targets the simulator and hence it was omitted from the experiments.

3.4.2 Efficiency of Shadow Memory Support

Recall that shadow memory support has two components: address translation and atomicity. Address translation can be either achieved using a Valgrind style software

Programs	LOC	Input	Description
BARNES	2.0K	8192	Barnes-Hut alg.
FMM	3.2K	256	fast multipole alg.
OCEAN	2.6K	258×258	ocean simulation
RADIOSITY	8.2K	batch	diffuse radiosity alg.
RAYTRACE	6.1K	tea	ray tracing alg.
WATER-NSQ	1.2K	512	nsquared
WATER-SP	1.6K	512	spatial

Table 3.3: SPLASH-2 Benchmarks Description.

implemented page table structure *VAL* or using our hardware assisted implicit addressing scheme *SM*. Atomicity can be achieved using thread serialization *ser* that is currently used in Valgrind; or with the help of fine-grained locking *fgl*; or using the CSC scheme with the help of exposed cache events. The performance of implementing various monitoring tools with different ways of achieving address translation and atomicity were explored. The results of this experiment are presented in Fig. 3.13, which shows the execution time overhead of performing four different monitoring tasks: DIFT, Memcheck, Eraser and MemProfile. In each of the graphs the first bar represents the performance of using Valgrind's address translation with thread serialization *VAL:serial*. The second bar represents the performance of using Valgrind's address translation with fine-grained locking *VAL:fgl*. The third bar represents the performance of using our implicit addressing scheme with fine grained locking *SM:fgl* and finally the last bar represents the performance of using implicit addressing with CSC scheme for achieving atomicity *SM:csc*.

As we can see, the overhead of performing monitoring using *VAL:ser* can be quite high. On an average it slows down the program by a factor of 25 for performing DIFT (45x for Memcheck, 35x for Eraser, and 27x for MemProfile). Using fine-grained locking *VAL:fgl*

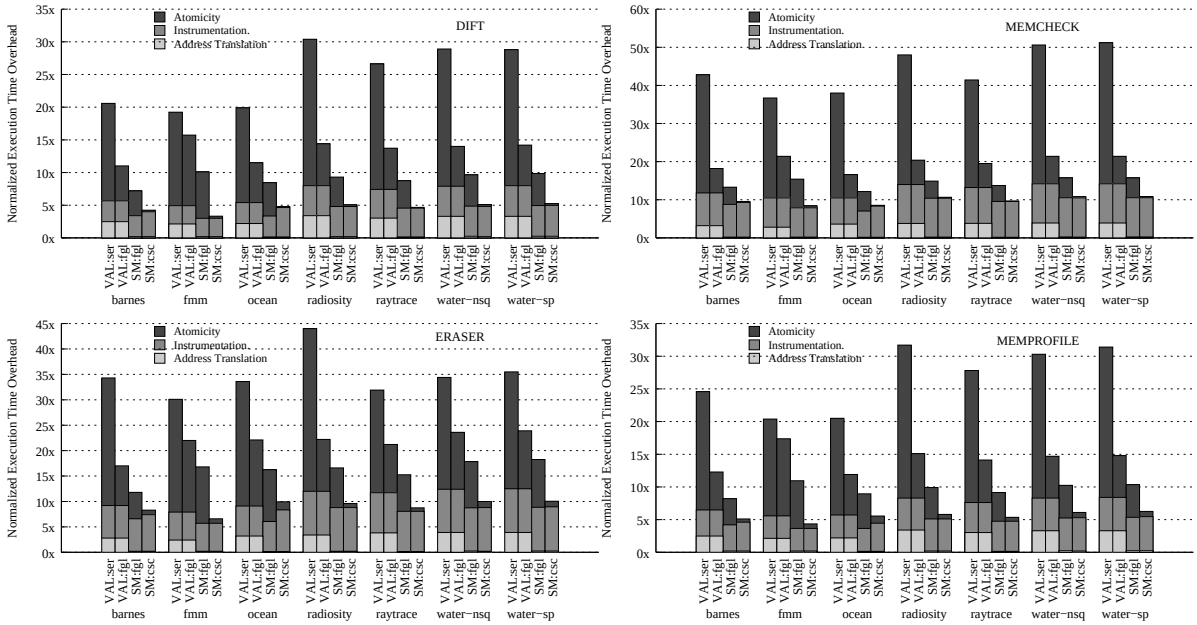


Figure 3.13: Monitoring Overhead with Various Shadow Memory Implementations.

obviates the need for thread serialization and reduces overhead to a factor of 13 slowdown for DIFT (20x for Memcheck, 21x for Eraser, 15x for MemProfile). Using implicit addressing of shadow memory proposed in this paper along with fine-grained-locking *SM:fgl* obviates the need for performing address translation in software and further reduces the overhead to a factor of 9 for DIFT (14.4x for Memcheck, 16x for Eraser, 9.5x for MemProfile). Finally our CSC scheme *SM:csc* all but eliminates the cost for performing locking and reduces the overall overhead to a factor of 4.5 slowdown for performing DIFT (9.8x for Memcheck, 8.8x for Eraser, 5.5x for MemProfile).

3.4.3 Break-Up of Overheads

To make more sense of the experimental results observed, the costs of performing monitoring were broken down into three categories: *address translation cost*, *instrumenta-*

tion cost and *atomicity cost*. While address translation cost involves execution of instructions to compute the shadow memory addresses for the original memory addresses and then access the shadow memory, instrumentation cost involves the execution of instructions for performing the particular monitoring task and atomicity cost refers to the cost of ensuring that OMIs and its corresponding SMIs are executed atomically. For this section, let us limit our discussion to the results of MemProfile.

First, let us consider the *VAL:ser* implementation. As we can see from Fig. 3.13, the atomicity costs dominate VAL:ser. This is not surprising as atomicity is enforced by thread serialization and since SPLASH-2 programs scale well, serialization almost quadruples the slowdown. Fine-grained-locking offers a slightly better alternative compared to serialization as we can infer from the results for *VAL:fgl*. However, using fine grained locks to implement atomicity additionally slows down the program by a factor of 2. This is because additional instructions (including costly atomic instructions) need to be executed for implementing locking.

Next, let us compare the overheads of *SM:fgl* with *VAL:fgl*. Since implicit addressing was used in *SM:fgl*, the cost of address translation is all but eliminated. The only cost of address translation is the small cost of executing the `shadow-start` and `shadow-end` instructions for identifying SMI. However, this cost is negligible compared to overall instrumentation overhead.

As we can see in *SM:csc*, the cost of implementing atomicity is greatly reduced. This is because using our CSC scheme, there is no need to execute additional instructions to perform locking. On the contrary, our CSC scheme serializes OMI and SMI from two processors, only if they potentially race with each other. As we can see from Fig. 3.13,

the cost for performing this limited serialization is small across all benchmarks for various monitoring tools.

Finally, it is important to note that that the overhead of performing monitoring using *SM:csc* is almost equal to the instrumentation cost that is inherent to each monitoring task. Thus, it is important to note that the two forms of architectural support added in this work: implicit addressing support and exposed cache events are effective in limiting the overhead of performing a variety of monitoring tasks.

3.4.4 Variation across Monitoring Tasks.

We observe that while instrumentation costs vary across various monitoring tasks (highest for Memcheck and lowest for DIFT), the address translation cost stays almost the same across the various monitoring tasks. It is also worth noting that the cost of implementing atomicity is slightly larger for Eraser and MemProfile in comparison with DIFT and Memcheck. This is because Eraser and MemProfile involve *general* SMIs – More specifically, original memory reads in these monitoring tools are accompanied by both reads and writes to corresponding shadow memory values. Thus shared reads in the original application, which would have caused read hits will now cause misses for corresponding accesses, causing additional slowdown.

3.4.5 Memory System Performance

In this section, memory system performance of our shadow memory scheme is evaluated in more detail. In particular, the overhead introduced by our CSC scheme for maintaining atomicity and the overhead introduced by the handling of additional shadow

values is evaluated.

Overhead introduced by CSC scheme: Recall that the CSC scheme is used to serialize OMI and SMI from two processors, if they race with each other. This serialization, albeit limited, causes additional overhead and in this section we measure this overhead.

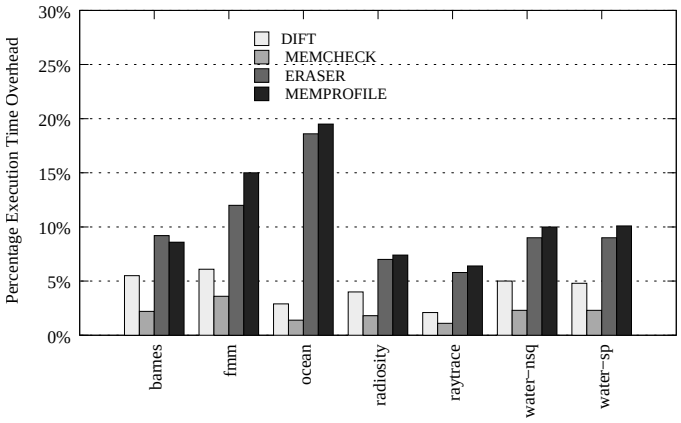


Figure 3.14: Percentage overhead due to CSC for various monitoring applications

As we can see from Fig. 3.14, the cost for performing this limited serialization is less than 18% across all the benchmark, for various monitoring applications. First, let us observe the trends across different monitoring applications. The CSC scheme introduces greater overhead for MemProfile (average 11%) and Eraser (average 10%), which consists of *general* SMIs compared to DIFT (average 4.3%) and Memcheck (average 2.1%), which use *symmetric* SMIs. This is due to the fact that dependence mirroring needs to be additonally enfoced for RAR in case of general SMIs. MemProfile and DIFT incur greater overhead for enforcing atomicity as apposed to Eraser and Memcheck owing to the fact that the latter monitoring applications are associated with heavier instrumentation, because of which the relative cost of enforcing atomicity becomes cheaper.

Second, the variation across different benchmarks is observed. It is interesting to

note that OCEAN and FMM incur high cost for monitoring applications with *general* SMI (MemProfile and Eraser) owing to the relatively larger concurrent shared memory reads in these applications. Other than this, in general, it is observed that the cost of enforcing atomicity via coupled coherence is inversely related to the instrumentation costs in each of the benchmarks.

Overhead due to extra shadow values Each of the monitoring applications are required to support additional shadow memory values, which can potentially slow down the application due to additional page faults and cache pollution.

First, the effect of additional page faults is measured. However any measurable degradation in performance due to additional page faults could not be observed. This is because of the fact that the memory footprint for these applications were small enough, so that the increased shadow pages could easily be accommodated in the main memory.

Then the effect of the additional shadow values on the miss rates of the caches was measured. Fig. 3.15 shows the L1 miss rates for various monitoring applications. First, it was observed that the L1 miss rates of the original unmonitored run in quite low (around 1%). As we can see, the miss rate increases marginally to 1.06% for DIFT; while it increases further to 1.35% for Memcheck; Eraser has a L1 miss rate of 1.7% and MemProfile has a simial miss rate of 1.8%.

Thus, only marginal increase in the L1 miss rate was observed for various monitoring applications (from 1% to 1.8%). Then, the increase in L2 miss rates across different monitoring applications was measured. However, any marked increase was not observed. Thus, the slowdown due to additional shadow values polluting the cache is very nominal.

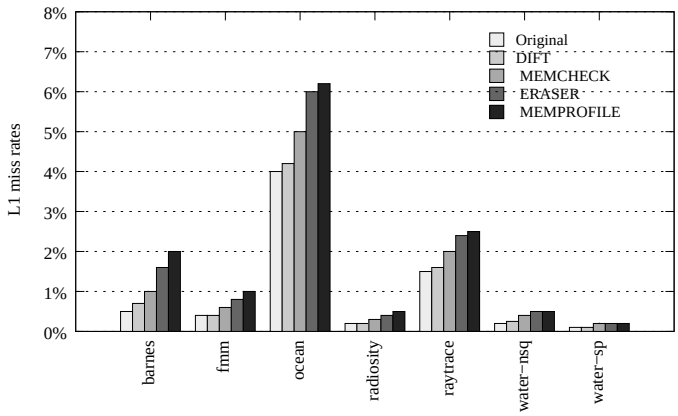


Figure 3.15: L1 Miss rates for various applications

3.5 Summary

In this chapter, a combination of architectural support (in form of ECMon: exposed cache events) and operating system support (in form of coupled allocation of memory pages used by the application and associated shadow memory pages) was used to derive a shadow memory implementation that is both efficient and robust. ECMon was used to couple the coherence of shadow memory with the coherence of the main memory, thereby ensuring that SMIs execute atomically with their corresponding OMIs. The proposed page allocation policy enables fast translation of original addresses into corresponding shadow memory addresses; thus allowing implicit and efficient addressing of shadow memory. The shadow memory support was implemented in a cycle accurate multicore simulator [80]. Four monitoring tasks DIFT, Memcheck, Eraser and MemProfile were used to evaluate the shadow memory support. Experimental evaluations showed that the shadow memory implementation was able to ensure atomicity of OMIs and SMIs efficiently. Furthermore, it was also able to significantly reduce the overhead involved in address translation.

Chapter 4

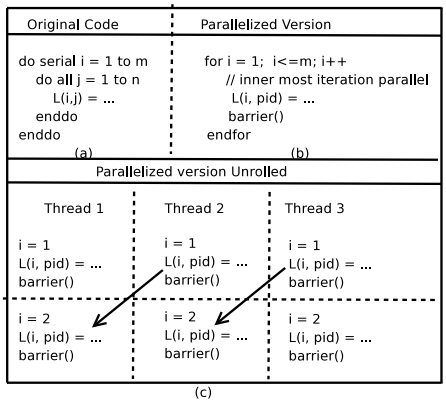
Speculative Optimizations using ECMon

In the last two chapters, ECMon support was introduced and it was used to enable two existing monitoring applications for improving reliability. The next two chapters presents two novel monitoring applications using ECMon support. In this chapter, ECMon support is used as a framework for enabling speculative optimizations in parallel programs [54, 56]. It is first observed that synchronization operations in parallel programs, although required, can greatly affect the program performance. In part, this is because they force the compiler to make conservative assumptions in generating code. It is observed that each processor spends significant portion of its execution time waiting at barrier synchronizations and immediately following synchronization, each processor executes significant number of redundant loads. In this chapter, ECMon support is first used to speculatively execute past barrier synchronizations, which is able to reduce the time spent idling at barriers, translating into a 12% reduction in execution times. ECMon support is also used to speculatively

promote variables in the presence of synchronization operations, that reduces the number of redundant loads executed, translating into an additional 2.5% reduction in execution time. This chapter then discusses an alternate implementation of misspeculation detection, in which existing hardware support for data speculation in Itanium processor in the form of ALAT (Advance load address table) is adapted to perform misspeculation detection.

4.1 Overview

The advent of multicores presents a promising opportunity for exploiting fine grained parallelism present in programs. Low latency and higher bandwidth for inter-core communication afforded by multicores allows for parallelization of codes that were previously hard to parallelize. Programs parallelized in the above fashion typically involve threads that communicate via shared memory, and synchronize with each other *frequently* to ensure that shared memory dependences between different threads are correctly enforced. In such programs threads often execute only hundreds of instructions independently before they have to synchronize with other threads running on other cores.



Program.	Time at Barrier	Redundant Loads (%)	Enforced Dep. (%)
Jacobi	25	6.25	25
Cholesky	61	24.8	6
Recurrence	42	88.5	12.5
Equake	32	2.1	3.2
Swim	24	0	3.1
Bisort	18	3.2	9
MST	28	8.4	34

Figure 4.1: Dependences enforced by synchronization and its characteristics

Fig. 4.1(a) illustrates a simple example that shows a sequential program with a doubly nested loop. While the iterations of the innermost loop are independent (*do all*) of each other, each iteration of the outermost loop (*do serial*) is dependent on values computed in the previous iteration. This naturally gives rise to a parallel implementation as shown in Fig. 4.1(b), where iterations of the innermost loop are distributed and computed by parallel threads, after which the threads synchronize at the barrier. Fig. 4.1(c) shows the loop-unrolled version of the parallel code, in which the arrows show the inter-thread dependences that are enforced by the barrier synchronization. As we can see from the above example, the purpose of adding the synchronization is to *enforce shared memory dependences across threads.*

(Effect of frequent synchronization) Such frequent synchronization operations, although required, can greatly affect program performance. In addition to causing the program execution to spend significant time at synchronization points waiting for other threads, synchronization operations also force the compiler to make conservative assumptions in generating code; for instance, variables cannot be allocated to registers across synchronization operations without precise analyses [81, 86] that guarantees that those variables that were allocated to registers cannot be modified by other threads. In other words, variables that are potentially *shared* cannot be allocated to registers across synchronization operations. To evaluate the importance of these factors, executions of a set of parallel programs that exploit fine grained parallelism were analyzed, as shown in Table 4.1. As we can see from column 1, each program spends a significant portion of its execution time (as high as 61%) waiting at barrier synchronizations. Furthermore, immediately following synchronization, each processor executes significant number of redundant loads (as high as 88%), owing to

the fact that shared variables could not be allocated to registers because of synchronization.

(Infrequent Dependences) One interesting property that was observed in our study is that barrier synchronizations used by these programs enforce interprocessor dependences that arise relatively infrequently. For each load executed by a processor following barrier synchronization, it was determined if the value that it read was generated by another processor prior to barrier synchronization. It was found that only 6% to 35% of loads executed at each processor involved interprocessor dependences (column 3 in the table in Fig. 4.1). Motivated by this observation, our approach consists of creating two versions of the section of code between consecutive synchronization operations. One version is a highly optimized version created under the optimistic assumption that none of the interprocessor dependences that are enforced by the synchronization operation will actually arise. The other version is unoptimized code created under the pessimistic assumption that interprocessor dependences will arise. At runtime, the optimistic code is first speculatively executed and if misspeculation occurs, the results of this version will be discarded and the non-speculative version will be executed. Clearly, the efficacy of this approach hinges on the misspeculation rate. Since interprocessor dependences arise infrequently, as we saw in our study, misspeculation rate remains low. This results in the execution of the optimized code most of the time, leading to performance savings.

(Efficient misspeculation detection) Another important parameter that affects the performance is the efficiency with which misspeculation is detected. It is worth noting that there is a misspeculation when there is an dependence between threads across barriers as shown in Fig. 4.1 – in other words, when there is an ISMD. To detect misspeculation, ECMon support detailed in chapter 2 is utilized. Recall that the ECMon support exposes ISMDs

to the software and thus can signal a misspeculation. More specifically, the `track-range` instruction is first used to specify the range of addresses that are speculatively read. Then the invalidate event is exposed to the software for the purpose of misspeculation detection. Consequently, when there is a remote write to any of the addresses specified by the `track-range` instruction, the handler is called. The call to the handler denotes a misspeculation and hence within the handler, control is transferred to the pessimistic non-speculative version.

This scheme is utilized to perform two speculative optimizations to improve parallel program performance. First, by speculatively executing past barrier synchronizations, execution time spent idling on barriers is reduced, translating into a 12% increase in performance. Second, by promoting shared variables to registers in the presence of synchronization operations, a significant amount of redundant loads is eliminated, translating into an additional performance increase of 2.5%.

This chapter is organized is as follows. First, the use of ECMon support in speculating past barriers is discussed. Then the use of ECMon support in performing speculative register promotion is discussed. This is followed by discussion of experimental evaluation. Then, an alternate technique for misspeculation detection – using support for data speculation in Itanium processors is discussed.

4.2 Speculation Past Barriers

Barrier synchronization is commonly used in programs that exploit fine grained parallelism – threads often execute only hundreds of instructions before they have to wait for other threads to arrive at the barrier. A thread that arrives at a barrier first, does no

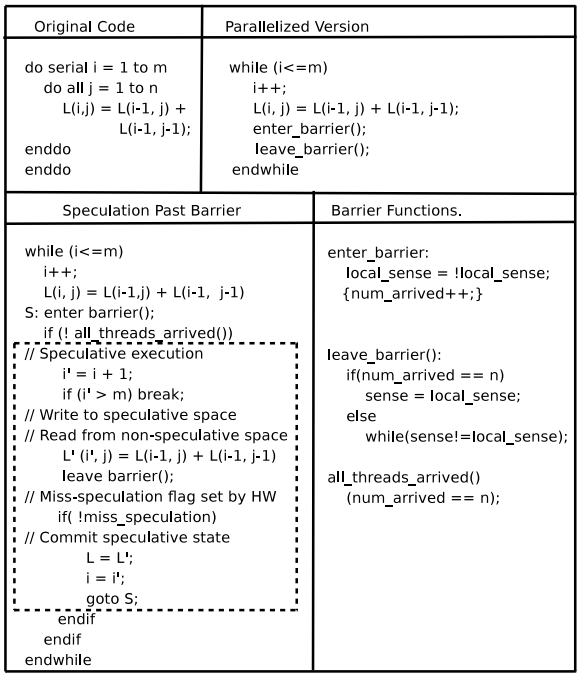


Figure 4.2: Speculative execution past barrier

useful work until other threads arrive at the barrier and this amounts to the time lost due to the barrier synchronization. In order to reduce the time lost due to the barrier synchronization, compilers typically try to distribute equal amounts of work to the different threads. However, threads often do not execute the same code and this in turn causes a variation in the arrival times. Moreover, even if each thread executes the same code, they can each take different paths leading to a variation in number of instructions executed. Experiments showed that the time spent on barrier synchronization can be as high as 61% of the total execution time for the set of programs considered. To reduce the time spent idling on synchronization, a compiler-assisted technique for speculating past barrier synchronizations is proposed. It is based on the observation that inter-thread shared memory dependences that the barriers strive to enforce can be infrequent. By speculatively executing past a bar-

rier, we in turn are speculating that the inter-thread dependences do not manifest during speculation. When the inter-thread dependences are infrequent, more often than not our speculation succeeds and a significant performance improvement in achieved.

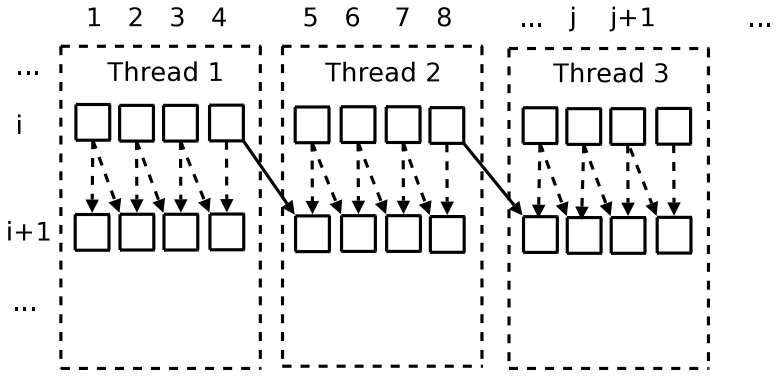


Figure 4.3: Dependences exercised

The above approach is illustrated using an example (Fig. 4.2) which shows the original sequential code, the unoptimized parallelized version and our optimized version which shows the compiler transformation for speculatively executing past barriers. The sequential code shows a doubly nested loop: each iteration of the inner loop can be done in parallel (*do all*), while the outer loop has to be performed sequentially since there is a loop-carried dependence. While each iteration of the inner loop could be given to different thread, this is not done typically [91]. To increase the computation-to-communication ratio and to preserve locality, each thread is given a part of the vector as illustrated in Fig. 4.3, where each thread is given a chunk consisting of four elements. Consequently in every iteration, each thread computes the values in its chunk, reading values computed from the previous iteration, after which it synchronizes with other threads by entering the barrier. Speculation past the barrier is enabled by generating code as illustrated in Fig. 4.2. Once

a thread arrives at a barrier, it is announced that the current thread has arrived at the barrier as shown in the function *enter-barrier*. Then it is checked if all threads have reached the barrier; if so, then there is no need to speculate and the next iteration is executed. However, if not all threads have reached the barrier, speculative execution is performed past the barrier.

4.2.1 Thread Isolation

A safe address partition is created for the speculative thread to work on. The primary benefit of this isolation is that *name* dependences that manifest between the speculative and the non-speculative threads can be safely ignored, and do not cause a misspeculation. Moreover, the need for heavy-weight rollback is obviated in case of a misspeculation; the newly created address space is merely discarded as in prior work [103]. If the speculation is successful, then the speculative state is merged with the non-speculative state. It is important to note that the above tasks viz. thread isolation, recovery from misspeculation and committing the results of a successful speculation are performed in software by the compiler. The compiler ensures thread isolation by writing to a separate address space during speculation. In other words, stores are transformed to store into the separate speculative address space. However, this creates a potential problem for reads; reads need to be able to read from original or new address space, depending on whether the read address has already been written into. To deal with this, each word of the new address space is associated with meta data which is initialized to 0. Whenever there is a store to a word, the meta-data for the corresponding word is set to 1 as shown in Fig. 4.4. Depending on the value of the meta-data, loads then read from the speculative (new) or non-speculative

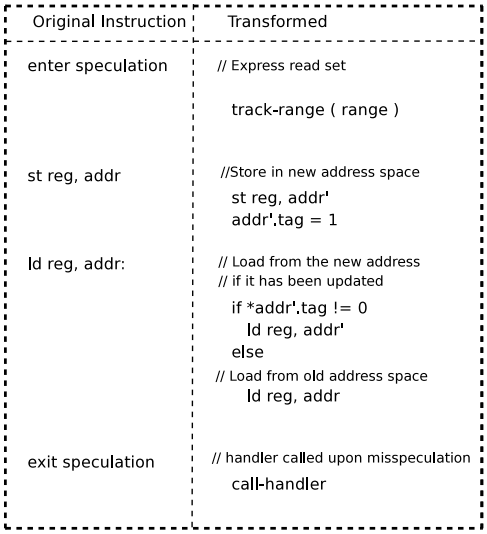


Figure 4.4: Code transformation

address space. However, for the programs considered, which essentially deals with loops working on vectors, the compiler is able to statically determine whether the reads have to read from the original or new address space, and this obviates the need for maintaining meta data for most loads and stores.

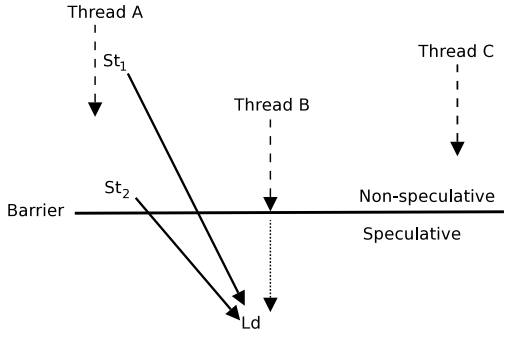
4.2.2 Misspeculation Detection

ECMon support is used to detect misspeculation. Upon entering speculative execution, the range of addresses that are read is set using the `track-range` instruction. Then the *invalidate* cache event is exposed to the software using ECMon. Whenever there is a remote write to any of the addresses specified in the `track-range` instruction , it would then cause the handler to be called. Recall that a remote write to any of the addresses read speculatively, signals an interprocessor dependency which the barrier was attempting to enforce. However, it is also the dependency that was not enforced due to the speculation and

hence such a dependency flags a misspeculation. Consequently, when the end of speculation is reached, it is checked if there was any invalidates to the read set. This is performed by executing the `call-handler` instruction. Consequently, if there had been any invalidates during speculation, the handler will be called. Within the handler, the speculative state is discarded, and control is transferred to non-speculative version. If there had been no invalidates during speculation, then the handler will not be called and the speculative state is committed. Committing the state involves copying the contents of the newly allocated space into the original non-speculative address space.

4.2.3 Reducing Misspeculation rate

Although the dependences enforced by the barriers are infrequent, they can still cause misspeculation if they manifest after the speculative code starts executing. As we can see from Fig. 4.5(a), thread B has reached the barrier and has started executing past it in speculative mode.



ORIGINAL CODE:

```
do all j = (start) to (start + size)
   L(i,j) = ...
enddo
```

PROFILING:

```
do j = (start) to (start + size)
   if L(i,j) is shared
      SHARED = SHARED U {j}
enddo
```

TRANSFORMED:

```
do j = (start) to (start+size)
   if in SHARED
      L(i,j) =...
enddo

do j = (start) to (start+size)
   if j not in SHARED
      Li,j) =...
enddo
```

Figure 4.5: (a) Reducing Misspeculation rate (b) Code transformation

When it encounters the load instruction, St_2 (thread A) has not yet been executed. In other words, the dependence between the St_2 and Ld has not yet been enforced. Thus,

when St_2 eventually executes in thread A, a misspeculation will be flagged. On the contrary, let us assume St_2 does not exist (or writes a different address) and so the only interprocessor dependence is between St_1 and Ld. In this case, note that by the time Ld instruction is executed in the (speculative) thread B, St_1 from thread A has already executed. In other words, the dependence between St_1 and Ld has already been enforced, and so this interprocessor dependency will not cause a misspeculation. Thus, to reduce the chance of misspeculation, it would be beneficial to advance writes to shared data (like St_1 and St_2), which is the focus of this optimization. To perform this optimization, the iterations that write to shared data are first identified as shown in the profiling step of Fig. 4.5(b). These iterations are then earlier than others. It is worth noting that this reordering can be performed only if the iterations in the inner loop can be performed in any order (do all).

4.3 Speculative Register Promotion

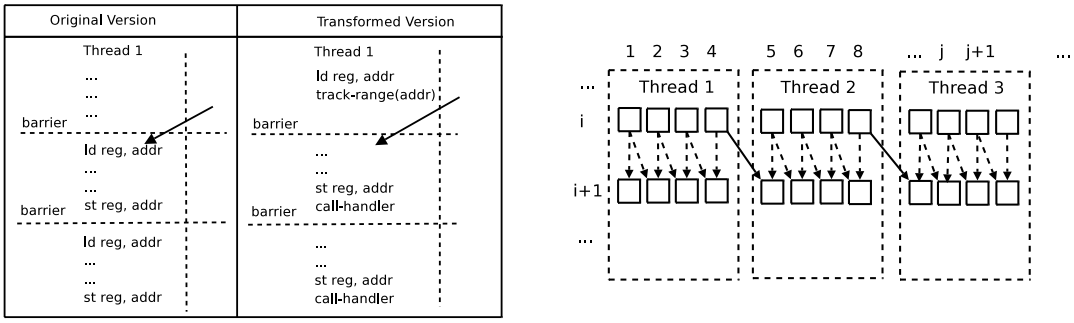


Figure 4.6: (a) Redundant loads due to barriers (b) Data partitioning

Recall that the purpose of synchronization operations are to enforce shared memory dependences across threads. However, lack of precise information about the dependences can lead to the execution of significant number of redundant loads. In our study it was

found that as high as 88% of loads executed around synchronization operations were redundant loads. Fig. 4.6(a) illustrates the reason for these redundant loads. When a barrier is reached, there is a need to dump all the variables that have been allocated registers to memory. Likewise, when a thread leaves a barrier, all the dumped variables have to be reloaded into registers. This is because, without information that guarantees that a variable is local to the thread, the compiler can not allocate the variable to a register across synchronization operations. Let us consider the same example of the doubly nested loop, whose inner loop can be parallelized. Recall that each thread is given a part of the vector to work on, to increase locality. Since each thread accesses and updates parts of the vector, the vector as a whole is *shared*. As shown in Fig. 4.6(b), thread 2, for example, reads in $L[4]$ through $L[8]$ and writes $L[5]$ through $L[8]$, every iteration. Since the vector is shared, the compiler cannot allocate individual elements across synchronizations. Thus, it cannot allocate $L[4]$ to a register in thread 2, because each iteration it is written by thread1. However, note that elements $L[5]$ through $L[7]$ are actually exclusive to thread 2 and could be allocated to registers across synchronizations. Without this fine grained partitioning information, it is hard for the compiler [81, 86] to figure out which of the variables are local to threads. On the other hand, it is relatively easier to estimate which of the variables are shared and local, by using profiling. Such probably local variables can be speculatively promoted to registers, provided there is a way to detect the case when this speculation is incorrect. In this optimization, the `track-range` instruction is used to speculatively allocate the variable to a register, at the same time, remembering the address. Whenever there is a remote write to the such addresses, the handler is called, which helps us to detect the situation when the speculation is incorrect. As we can see, in the transformed version in Fig. 4.6, by

promoting the variable to the register, the redundant loads can be removed every iteration. It is worth noting that while the redundant load can be removed, we still have to store the value to the memory every iteration before the barrier. This would serve as a means to detect misspeculation in case the same variable had been promoted to a register in some other thread.

However, speculatively promoting registers is not as simple as removing the loads during speculation (past the barrier) and remembering the addresses in the ALAT. To see why, let us consider the Fig. 4.7. As we can see by executing, `track-range` the addresses of variables that have been speculatively promoted to registers are being remembered. This will mean that, if there is a store in thread A (St_2), this will call the handler and notify us of our misspeculation. However, let us consider the case of St_1, from thread A, which is assumed to write to the same address. Since St_1 has already been executed in thread A, before the thread B has arrived at the barrier, there is no way of detecting this dependency; in other words, misspeculation can not be detected in this scenario. To handle this situation, the variables are speculatively loaded into registers once, initially (outside the loop) as shown in Fig. 4.8. This enables us to remember the addresses via the `track-range` instruction outside the loop. Whenever a thread reaches a barrier, before speculatively executing past it, we check if there has been a misspeculation by executing the `call-handler` instruction. If the handler is not called, this means that there has been no stores to the speculatively promoted addresses. This in turn means that the promoted registers can be continued to used without reloading. However, if the handler is called, then there has been a store to one of the speculatively loaded registers. This, in turn, means that the variables have to be reloaded into registers and this is precisely what is done within the handler. While

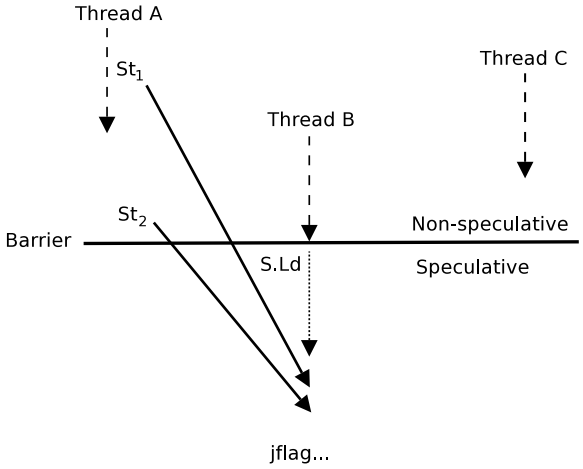


Figure 4.7: Promoting registers during speculation

Event	Code Generated

intital:	// Speculatively load the registers
	spec_load_reg();

enter speculation:	// Speculatively load the registers
	// only if there is misspeculation
	call-handler
	// set the read set
	track-range (range)

exit speculation:	// Speculatively load the registers
	// only if there is misspeculation
	call-handler
	// if handler is not called,
	// commit speculative state

handler function:	spec_load_reg():
	ld reg$_1$, addr$_1$;
	ld reg$_2$, addr$_2$;
	...
	ld reg$_n$, addr$_n$;
	track-range (addr$_1$- addr$_n$)

Figure 4.8: Code transformation

reloading the variables to registers, `track-range` instruction is again used to remember that these variables are again speculatively loaded. After taking care of registers, speculative execution past the barrier is performed. As before, before exiting the speculation, the call-handler instruction is again executed. It is important to note that this could mean one of two things: either the values that have been speculatively loaded have been written into, or the values that have been speculatively promoted have been written into. To take care of the latter, the variables are reloaded into registers, at the same time, remembering the loaded addresses using the *track-range* instruction. If the misspeculation flag has not been set at this point, the handler is not called and the speculative state is committed.

4.4 Alternate Support for Misspeculation Detection

In the previous chapter ECMon support was utilized for detecting misspeculation. However, existing support for data speculation in Itanium processors can be alternately modified and adapted for detecting misspeculation. More specifically, support for *data speculation* in the form of *Advanced Load Address Table* (ALAT) already present in Itanium processors [1, 41] is used. This architectural support is exposed to the compiler via two new instructions: the speculative read `S.Rd` and the jump on misspeculation `jflag` instruction. The `S.Rd` instruction enables us to specify the range of memory addresses that are speculatively read, which are efficiently remembered in the ALAT. Once the speculative read `S.Rd` is executed, the hardware ensures that remote writes to any of the addresses, from other processors invalidate the corresponding entry in the ALAT and sets the *misspeculation flag*. The compiler is given the ability to read this flag via the `jflag` instruction and hence can react to misspeculation by jumping to the pessimistic non-speculative version.

4.4.1 Support for misspeculation detection

(ALAT for data speculation) The support for misspeculation detection is based on data speculation support already present in Itanium processors [1, 41]. This hardware support is in the form of a hardware structure known as the *Advanced load address table* (ALAT) and the special instructions associated with it. When a data speculative load is issued with a special load instruction known as the *advanced load* `ld.a` instruction, an entry is allocated in the ALAT, storing the target register and the loaded address. Every store instruction then automatically compares its address against all entries of the ALAT. If there is a match, the corresponding entry in the ALAT is invalidated. Using the `chk.a` instruction, a check is performed before the speculatively loaded value is used; while a valid entry means that load speculation was successful, an invalid entry means that the data has to be reloaded. Our observation that the above HW support can be used for misspeculation detection in our scheme stems from the fact that entries in the ALAT are also invalidated by *remote writes to the same address from other processors* [1]. Thus, in our optimistic speculative version, values are loaded using advance load `ld.a` instructions, each load will create an entry in the ALAT. A subsequent remote write in another processor to any of the addresses loaded will precisely indicate an interprocessor dependency that was not correctly enforced through our speculation. Such remote writes invalidate the ALAT entries and thus serve as a mechanism for misspeculation detection.

(Modified ALAT for misspeculation detection) However, there are some significant differences between data speculation, which is primarily meant for scalars in sequential code, and speculative optimization for parallel programs. First, speculatively loading all values in the optimized optimistic version, via the `ld.a` instruction would very likely exhaust all

possible entries in the ALAT. To deal with this size limitation, the speculative read `S.Rd` instruction is proposed, through which the compiler can specify ranges of addresses that are to be read speculatively.The purpose of `S.Rd` instruction is to merely inform the processor about the (range of) addresses that are read, and no actual loading from memory happens. In our design, each processor's ALAT can hold 4 such address ranges. We provide this capability so that vectors that are read speculatively can be specified succinctly by the compiler. If the compiler is unable to determine the range, it then can generate code with the `S.Rd` accompanying the loads in program without specifying the range. The hardware then takes the responsibility of inserting the address into any of the ranges maintained. The hardware does this conservatively – at any time the range of addresses maintained by the hardware is guaranteed to contain all the addresses read speculatively.

Another important semantic difference is that, while data speculation requires that local stores invalidate ALAT entries, speculation for parallel programs does not require this. Accordingly, the ALAT entries created by `S.Rd` instruction are not invalidated by local stores. It is worth noting that in the modified ALAT, there is still support for conventional data speculation. Thus local stores are made to invalidate ALAT entries for the advanced load instruction `ld.a` instruction, like before. Finally, a *flag* is added to the ALAT which is used to indicate misspeculation. This flag can be reset by the compiler using the `rflag` instruction. Whenever a remote write to one of the ALAT (ranges) is detected, the *flag* is set. The compiler can access this flag via the jump on misspeculation instruction, `jflag`, using which the compiler can jump to the pessimistic non-speculative version on detecting a misspeculation.

Fig. 4.9 shows how the instructions interact with the ALAT. Following a syn-

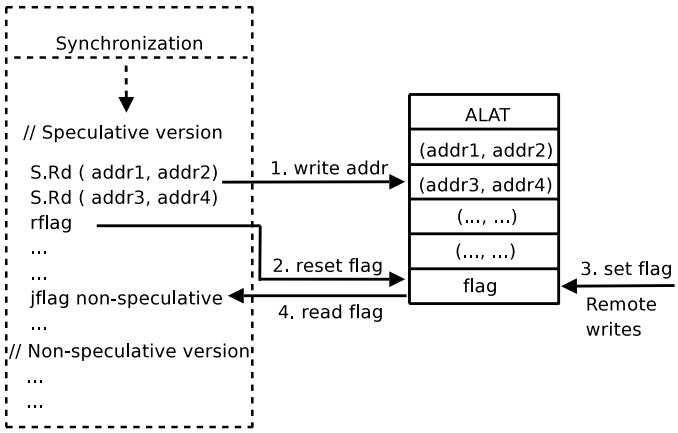


Figure 4.9: Interaction of instructions on ALAT

chronization operation, the optimistic version which assumes the absence of interprocessor dependences is first executed. The compiler specifies the range of addresses that the speculative code reads, via the `S.Rd` instruction. Accordingly, the ranges of addresses are remembered in the ALAT (step 1). The `rflag` instruction is then used to clear the misspeculation *flag* (step 2). While the optimistic version is executing, if there are any remote writes to any of the address ranges in the ALAT, the misspeculation flag is set (step 3). Finally at the end of speculative version, the compiler checks for misspeculation via the *jflag* instruction, and jumps to the non-speculative version, if there is a misspeculation.

4.4.2 Microarchitecture support

(Misspeculation detection) Fig. 4.10 illustrates the microarchitectural support needed for misspeculation detection. Remote writes that write to the addresses in the ALAT are detected using the *invalidate* cache-coherence message like in the Itanium [1]. Additional logic is required in the form of comparators for comparing the invalidated block address with

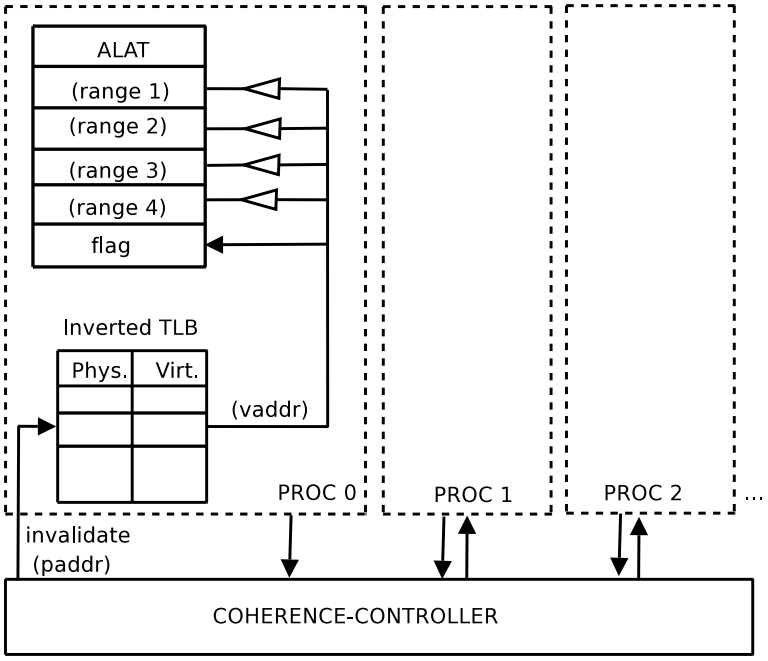


Figure 4.10: Microarchitecture support

the ranges specified in the ALAT; if there is a match, the misspeculation *flag* is set. One complication stems from the fact that the address ranges specified by the `S.Rd` instruction are virtual addresses, while the addresses associated with coherence messages are physical addresses. Accordingly, the physical addresses from the coherence messages are converted to virtual addresses before they are compared with the entries in the ALAT. To enable this conversion, an additional inverted TLB that stores the physical to virtual page mappings is maintained.

(Representing Ranges) Each ALAT entry stores an n bit address value ($a_n a_{n-1} ... a_0$) and a movable bit marker. The position of the bit marker implicitly determines the range represented by the ALAT entry. If the bit marker resides in *ith* position, it represents a range of address values between ($a_n a_{n-1} ... a_i 00...0$) to ($a_n a_{n-1} ... a_i 11...1$). For example, in Fig. 4.11, the first entry in the ALAT represents a range of addresses between 8 and 15.

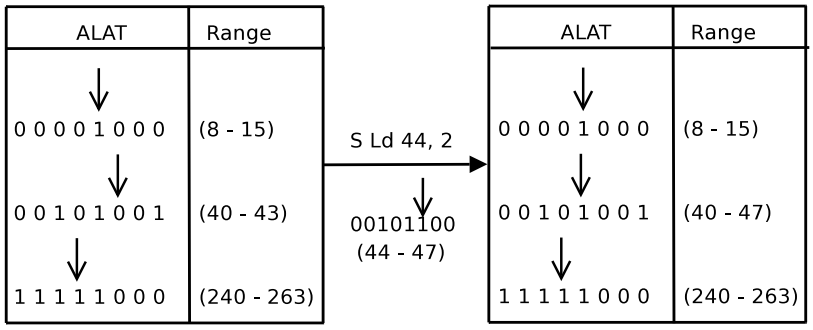


Figure 4.11: Range Representation

This is because the bit marker resides in the 3rd bit. With this representation, determining if an address value lies within the range becomes as simple as comparing the first $n - i$ higher order bits. Since the ranges are represented in the above fashion, the operands of the `S.Rd` instruction are really an *address value* and a *bit position* value to represent the range. If the compiler is not aware of the range, then it specifies a bit position of 0 [1].

(Extending the ranges) When a new `S.Rd` instruction is encountered, a new ALAT entry is created to represent this range. However, if the ALAT entries are exhausted, the new range is merged into one of the pre-existing ranges represented in the ALAT. Let us suppose that the new instruction specifies an address *addr* and each of the the ALAT entries have the address $addr_k$ with current bit position b_k. Then the first b_k bits of *addr* are compared with each entry $addr_k$. If there is match, then it means the current address *addr* has already been accounted for, in the ALAT. If there is no complete match, the ALAT entry with the closest match is chosen and its range is extended to include *addr*. The ALAT entry with the maximum number of higher order bits matching *addr* is thus selected and the bit marker is moved left until the range represented by the entry includes

[1] In our implementation, the bit marker resides initially at kth bit where 2^k is the block size of the cache, since addresses are stored at cache block granularity

addr. Fig. 4.11 shows an ALAT with 8 bit addresses that has the capacity for storing three ranges. For the first address, the bit marker is pointing at the 3rd bit, representing the range of addresses between 8 and 15. Similarly the other two entries store ranges *40 - 43* and *240 - 263* respectively. The above example also shows the transformation in the ALAT when there is a new `S.Rd` instruction with the range *44 - 47*. Since there are no free entries in the ALAT, the new range is accommodated by extending the second range. The second range is chosen since it is closest to the new range – the first 5 higher order bits match. Consequently, the range extension for the second entry is reflected by moving the bit marker towards the left.

4.4.3 Speculation using modified ALAT

This section discusses how the two speculative optimization: speculation past barriers and speculative register promotion are performed using the modified ALAT. The general algorithms for speculation using ALAT is very similar to performing speculation with ECMon. However, there is one important difference. The `S.Rd` instruction is used to maintain the addresses of the read set instead of the `track-range` instruction. This means that there is no necessity for the compiler to figure out the read set. It merely suffices that the compiler instrument every speculative load with an `S.Rd` instruction; the hardware automatically maintains the speculative read values (the read set) efficiently, with the range representation support.

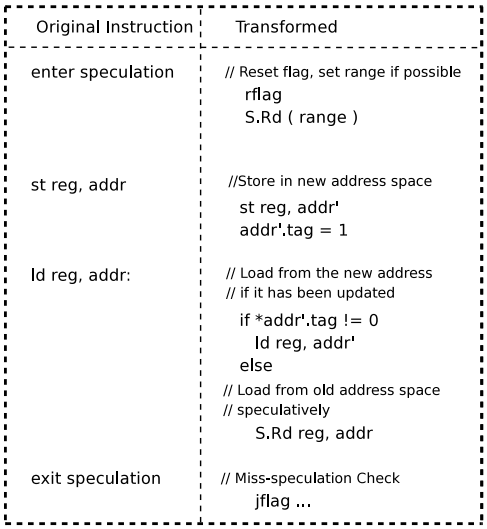


Figure 4.12: Code transformation

Speculation past barriers using ALAT

Fig. 4.12 details the code transformation for speculating past barriers using the modified ALAT. The basic idea is same as the code transformation with ECMon support with some minor differences. Upon entering speculative execution, the `rflag` instruction is used to reset the misspeculation flag. Then the range of addresses that are read using the `S.Rd` instruction. However, this step is not a necessity; the compiler can set set the range of addresses if it it has this information. If the compiler is not able to statically determine the range of addresses read, then `S.Rd` instructions are made to accompany the loads as shown in Fig. 4.12 – the hardware will ensure that all the addresses that are read from, are remembered in the ALAT. Whenever there is a remote write to any of the addresses remembered in the ALAT, it would then invalidate the ALAT entry and set the misspeculation flag. Recall that a remote write to any of the addresses read speculatively,

signals an interprocessor dependency which the barrier was attempting to enforce. However, it is also the dependency that was not enforced due to the speculation and hence such a dependency flags a misspeculation. Consequently, at the end of the speculation, the value of the misspeculation flag is checked and if it is not set, the speculative state is committed. Committing the state involves copying the contents of the newly allocated space into the original non-speculative address space, as usual.

Speculative register promotion using ALAT

The general idea of speculatively promoting registers using ALAT support is again similar. The variables are speculatively loaded into registers once, initially (outside the loop) as shown in Fig. 4.13. This enables us to remember the addresses of the loaded variables in the ALAT. The misspeculation flag is then reset via the `rflag` instruction. Whenever a thread reaches a barrier, before speculatively executing past it, the value of the flag is checked. If flag value is not set, this means that there has been no stores to the speculatively promoted addresses. This in turn means that the promoted registers can be safely used without requiring reloading. However, flag that has been set at this point means that there has been a store to one of the ALAT entries. This, in turn, means that the values of the registers have to be reloaded. This is precisely what is done While reloading the variables to registers, the `S.Rd` is again used to remember the loaded values in the ALAT. Having taken care of registers, speculative execution is now performed. Likewise, before exiting the speculation, the value of the misspeculation flag is checked. It is important to note that this could mean one of two things: either the values that have been speculatively loaded have been written into, or the values that have been speculatively promoted have been written

Event	Code Generated

intital:	// Speculatively load the registers
	spec_load_reg();

enter speculation:	// Speculatively load the registers
	// only if flag is set.
	if(flag)
	{
	spec_load_reg();
	}
	// Reset flag, set range if possible
	rflag
	S.Rd (range)

exit speculation:	// Speculatively load the registers
	// only if flag is set.
	if(flag)
	{
	spec_load_reg();
	}
	else
	{
	// commit speculative state
	}

helper function:	spec_load_reg():
	rflag
	ld reg_1, $addr_1$; S.Rd $addr_1$
	ld reg_2, $addr_2$; S.Rd $addr_2$
	...
	ld reg_n, $addr_n$; S.Rd $addr_n$

Figure 4.13: Code transformation

into. To take care of the latter, the values of the variables are reloaded into the registers, at the same time, remembering the loaded addresses in the ALAT. If the misspeculation flag has not been set at this point, the speculative state is committed as usual.

4.5 Experimental Evaluation

Processor	8 processor, inorder
L1 Cache	32 KB 4 way
L1 hit latency	1 cycle
L2 Cache	512 KB 8 way
L2 hit latency	9 cycle
Memory latency	200 cycle
Coherence	MOSI bus based

Program.	Source
Jacobi	Iterative solver
Cholesky	Cholesky gradient
Recurrence	Linear recurrence
Equake	Earthquake simulation
Swim	Weather prediction
Bisort	Bitonic sort
MST	Minimum spanning tree

Figure 4.14: (a) Architectural parameters used for simulation (b) Programs used

In this section, the experimental results of our ECMon/modified ALAT assisted speculative optimization framework is presented. First and foremost, the performance increase obtained via speculatively executing past barrier synchronizations is presented. Since key to a good performance is low misspeculation rate, the misspeculation rate was evaluated; the effect of our compiler transformation of reordering the loops were also presented. Next, the performance increase obtained by speculatively promoting shared variables into registers was studied. But before the experimental results are presented, the implementation and the benchmarks used are described. The architectural support in the SESC [80] simulator, which is a cycle accurate CMP simulator targeting the MIPS architecture. Both the ECMon support and the modified ALAT support was implemented in the simulator. This is because the proposed architectural support of ECMon as well as the modified ALAT

is not available in current processors and hence had to be simulated. For the simulation, the architectural parameters listed in Table. 4.14 are used. The benchmarks used are a set of seven parallel programs listed in Fig. 4.14. *Cholesky* (kernel 2) and *Recurrence* (kernel 6) are parallelized versions of Livermore loops, whose implementations are described in [87]. *Equake* and *Swim* are from the SPEC *Openmp* benchmark suite, while *Bitonic sort* and *MST* are from the *Olden* benchmarks suite. Finally, the parallelized version of the *Jacobi* iteration was also used. Each program was rewritten to make use of synchronization constructs associated with the simulator and compiled each program to run on the simulator using the simulator's cross compiler. It is important to note that since the above programs synchronize frequently using barriers, they are interesting subject programs for the evaluation of our technique.

4.5.1 Execution Time Reduction using ECMon

First, the execution time reduction was measured using the ECMon support. Profiling was used to identify the part of shared data that is local to each thread and such variables were promoted to registers. To keep misspeculation at a minimum the compiler technique described to reorder the iterations of loop was used.

As we can see from Fig. 4.15(a), the execution time was reduced significantly using the two techniques. The percentage reduction in execution times ranges between 6% (Bitonic sort) and 24% (Livermore loop 2). On an average, a 12% reduction in execution time was achieved by speculatively executing past barriers. By performing speculative promotion of variables into registers, a further reduction in execution times was achieved. *Recurrence* and *Cholesky*, have a significant number of redundant loads around synchroniza-

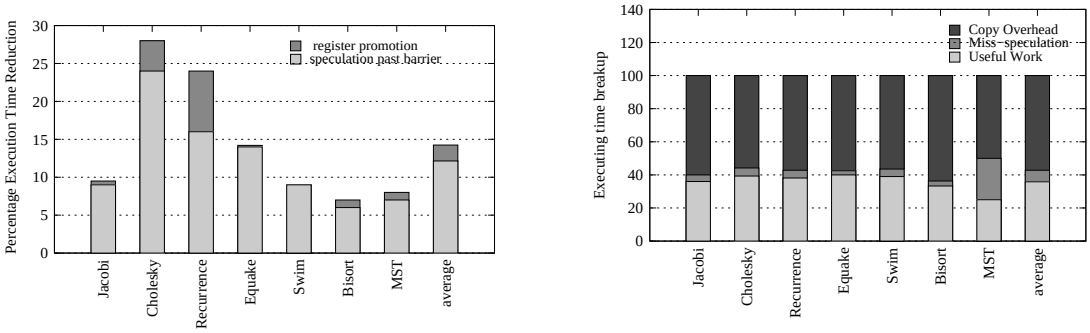


Figure 4.15: (a) Execution time reduction and (b) break up

tions and for these programs, the execution times significantly – 8% and 4% respectively. On an average, execution time was reduced by a further 2.5% across all benchmarks, due to speculative register promotion. To gain further insight as to why speedup was achieved, we measured how the original time spent in synchronization (without speculation) was now being spent with speculation. As we can see from Fig. 4.15(b), about 37% of the original time spent in barrier is now channeled into performing useful work. We can also see that the time spent inside the handler recovering from misspeculation is relatively low (about 5%), owing to small number of misspeculations. However, significant time (about 58%) was spent performing copies for maintaining and committing speculative state.

Efficacy of loop reordering Recall that to reduce misspeculation, loops were reordered, so that updates to shared data take place earlier. To determine the efficacy of this optimization, the misspeculation rates before and after application of this transformation was measured. As we can see from Fig. 4.16, this optimization significantly reduces the misspeculation rates for Jacobi (44% to 5%), Equake(54% to 2.3%) and Swim (51% to 6%) programs. In the above three programs, there was a shared update in the end of each

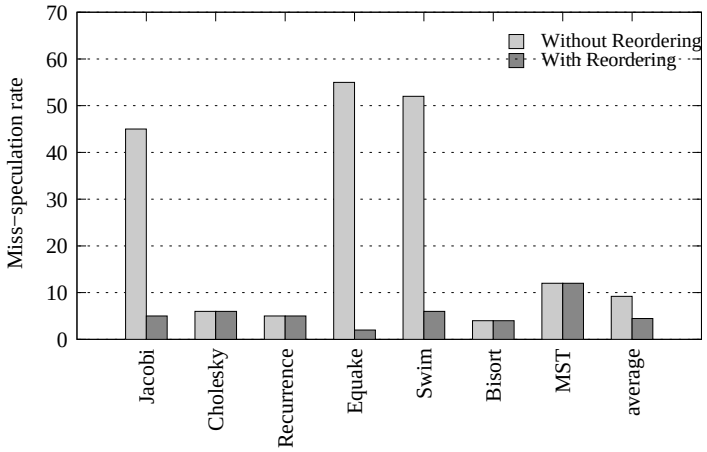


Figure 4.16: Efficacy of reordering

thread's execution which was causing misspeculation. Once this shared update was moved earlier, the misspeculation rate significantly dropped.

4.5.2 Execution time reduction using modified ALAT

The execution time reduction obtained by the application of the above speculative optimizations using a modified ALAT was also evaluated. It was found that the execution time reduction was almost identical to the above results obtained. This result is not surprising since the support involved in modified ALAT is equivalent to ECMon support for misspeculation detection. The only difference being that in ECMon a handler is called when a misspeculation is detected, while with the modified ALAT support, the code for reacting to misspeculation is inlined within the original code, using the `jflag` branching instruction. Since the misspeculation rate was quite low, the difference due to the above was not apparent in the experimental results.

4.6 Summary

In this chapter, ECMon support was used as a framework for performing speculative optimizations for parallel programs running on multicores. In particular, ECMon was used to perform two speculative optimizations to improve parallel program performance. First, by speculatively executing past barrier synchronizations, time spent idling on barriers was significantly reduced, translating into a 12% reduction in execution time. Second, by promoting shared variables to registers in the presence of synchronization, a significant amount of redundant loads were reduced, translating into a further reduction of 2.5% in execution time. Finally, an alternate technique for misspeculation detection using a modified ALAT (Advanced load address table), which is already present in Itanium processors, was discussed.

Chapter 5

Self Recovery in Server Programs

It is important that long running server programs retain *availability* amidst software failures. However, server programs do fail and one of the important causes of failures in server programs is due to *memory errors*. One safe way of recovering from these crashes is to periodically checkpoint program state and *rollback* to the most recent checkpoint on a crash. However, checkpointing program state periodically can be quite expensive. In this chapter, a detailed study is conducted to see how memory corruption propagates in server programs. Our study shows that memory locations that are corrupted during the processing of an user request, generally do not propagate *across* user requests. On the contrary, the memory locations that are corrupted are generally *cleansed* automatically, as memory (stack or the heap) gets deallocated or when memory gets overwritten with uncorrupted values. This self cleansing property in server programs led us to believe that recovering from crashes does not necessarily require the expensive roll back of state for recovery. Motivated by this observation, SRS, a technique for *self recovery in server* programs which takes advantage of *self-cleansing* to recover from crashes is proposed. Since SRS is a software

based monitoring application that utilizes meta data stored in shadow memory, it is not able to run on multicores because of races between data and meta data [64]. To enable SRS for multicores, ECMon support is utilized to enforce consistency of data and meta data accesses.

5.1 Overview

Long running server programs seek to maximize their uptime and thereby ensure that they are *available* to users. However, server programs do fail and one of the important causes of failures in server programs is due to *memory errors*. According to the *National Vulnerability Database* [4], memory errors like buffer overflows, format string errors, integer overflows etc. constitute a significant percentage (30% as of 2008) of software failures. Memory bugs in the server code, when exposed by certain user request, can lead to memory corruption which can eventually lead to crashes or even software attacks (if user input is malicious).

There has been significant research on the recovery from software failures. One safe way of recovering from such software failures and ensure availability is to periodically checkpoint program state and rollback to the most recent checkpoint, when failure is detected [30, 71, 72, 77, 101]. Having rollbacked to a safe state, all user requests starting from the safe state point until the crash point are replayed; during replay, the particular bad user request that triggered the crash is identified and dropped [72, 101] so that the same failure is not repeated. However the checkpointing/rollback scheme has its limitations. First, checkpointing program state periodically can be quite expensive. Second, since recovery can involve rolling back of considerable state information, the throughput and response time of

the server can be reduced significantly during rollback recovery. Third, since checkpointing is done only at specific program points, recovery can involve the replay of several good user requests. Finally, checkpointing/rollback system is complex and poses implementation challenges for multithreaded programs [72].

In this chapter, a detailed study of memory corruption was conducted in server programs, to see how memory corruption propagates as the server program executes, with a view to understand whether the expensive checkpointing and rollback operations are really needed. If the memory corruption does really propagate a lot through memory, then that would vindicate the expensive state rollback in the checkpointing/rollback approach; on the contrary, if memory corruption does not propagate all that much, such expensive recovery mechanisms might not be really needed. In our study of real world server programs, values written to the memory by different store instances (all store instances were comprehensively tested) are assumed to be "corrupt" and its propagation studies. This study showed that memory locations that are corrupted during the processing of an user request, generally *do not propagate across user requests*. On the contrary, the memory locations that are corrupted are often *cleansed* automatically, as memory (stack or the heap) gets deallocated or when memory gets overwritten with uncorrupted values. Thus this self cleansing property in server programs showed that recovering from crashes does not necessarily require the expensive roll back of state for recovery.

Motivated by the above study, *SRS* a safe technique for enabling Self Recovery in Server programs is proposed. In SRS, the self-cleansing property is in server programs is utilized to *isolate* the faulty user request from other succeeding benign user requests, without checkpointing or rollback. In other words, in SRS, when a crash occurs, we do not

rollback state to a previously saved safe state; on the contrary, we *suppress the crash and execute forward*. When executing forward, SRS guarantees that the user requests succeeding the faulty request do not read any values written during the processing of the faulty request, thereby achieving the effect of dropping the bad request. Despite self cleansing, there can be a small number of memory locations that remain corrupt; if there is a need to access such a corrupted location, a *demand driven* approach is used to restore the corrupted value when a read for it is encountered. Thus in SRS, instead of the expensive rollback operation to restore the memory contents to a safe state, an efficient *demand driven restoration* technique us used. Furthermore, since execution is made to proceed forward past a crash, the need to replay benign user requests is eliminated altogether.

However, executing forwards past a crash is not without its own challenges. Even though the first crash can be suppressed, similar crashes can recur (and likely will recur), when values that are dependent on the corrupted memory values are used later. To prevent such crashes from recurring a mechanism called *crash suppression* is used, in which those instructions that use values that are corrupted are not made to execute, and are *suppressed*. Owing to the self-cleansing property, fewer memory locations remain corrupted because of which fewer instructions need to be suppressed as execution moves forward.

This demand demand approach for recovery requires program monitoring, albeit minimal, and is accomplished via software instrumentation. However, as shown in chapter 3, monitoring multithreaded server programs running on multicores, introduces races between data and meta data, which can corrupt the monitoring process. To enable monitoring of multithreaded server programs on multicores, we use ECMon support.

SRS was evaluated with real world memory bugs in 4 widely used server programs

and it was found that SRS could successfully recover from the failures caused by faulty user requests. It was also found that SRS is efficient, causing negligible drop in the response time of the program during normal run and after recovery.

5.2 Study of Memory Corruption Propagation

The process of exposing a memory bug in a program via a user request that causes a crash consists of three events in program execution. In the first step, the bug in the source code is traversed by user input. In the second step, the traversal of the bug leads to the first point of memory corruption; this is the point when a memory location is mishandled in some way. The corrupted memory location then *propagates* across memory where it spreads and corrupts other memory locations. Finally, a crash occurs when there is an access to a spurious memory location. The goal of a checkpointing/rollback system is to rollback to a prior memory state, a state that is hopefully devoid of memory corruption. We conducted a detailed study of memory corruption in server programs to see how memory corruption propagates as the server program executes, with a view to understand whether the expensive checkpointing and rollback operations are really needed. Does the corrupted memory location go on to corrupt other memory locations; if so on an average how much does the corruption spread? Is it possible that the set of corrupted memory locations can shrink? These were some of the questions that the memory propagation study can potentially answer. If the memory corruption does really propagate a lot through memory, then that would vindicate the expensive state rollback in the checkpointing/rollback approach; on the contrary, if memory corruption does not propagate all that much, such expensive recovery mechanisms might not be really needed. In this section, the server programs used

in the study are first discussed. Then, the study of memory corruption propagation along with an analysis of salient observations is discussed.

4 widely used real world server programs are used in this study. *Mysqld*, *cvs*, *squid*, and *apache* are the servers considered as listed in Table 5.1. For each of these programs, the client sends separate user requests; for *mysqld* 3 separate requests that build and manipulate separate tables are sent; for cvs 3 separate requests to checkout source code are sent; for *apache* and *squid* 3 requests are sent for downloading files of various sizes.

5.2.1 Memory Propagation Study

Methodology: The purpose of this study is to understand how memory corruption propagates through memory as the program executes. One possibility is to study the propagation of the memory corruption in real bugs. However, the memory propagation clearly depends on what memory location(s) are corrupted. For example, a global variable that is marked corrupt is expected to lead to several other corrupted values, while a local temporary may only lead to fewer corrupt values. Hence, to comprehensively study the propagation of memory corruption, several executions of the server program are considered; in each execution, different memory locations are assumed to be corrupt and its propagation studied. More specifically, in each execution it is assumed that a *specific dynamic store instruction writes a corrupt value to memory and study the propagation of this corruption in the rest of the execution.* This way, all possible memory locations that can potentially be corrupted are exhaustively covered.

The steps of the memory propagation study, as illustrated in Fig. 5.1, consists of two phases. In the first phase, called *Trace Stores*, a trace of all store instructions and the

Table 5.1: Server Programs Characteristics.

Program	Description	LOC
mysqld	Database server	588K
cvs	Version Control server	93K
squid	Web Proxy cache server	283K
Apache	Web server	114K

Table 5.2: Memory Corruption Propagation.

Action	Target	Src1	Src2
Fault	Corr.		
Instr: Target = src1 op src2	Corr.	Uncorr.	Corr.
Instr: Target = src1 op src2	Corr.	Corr.	Uncorr.
Instr: Target = src1 op src2	Corr.	Corr.	Corr.
Instr: Target = src1 op src2	Uncorr.	Uncorr.	Uncorr.
Deallocation	Uncorr.		

corresponding instruction count when each store instruction is executed (see step 4 in 1st phase) are collected, so that each dynamic store can be uniquely identified. In the second phase, called *Memory Corruption Propagation*, we repeatedly assume each unique dynamic store to be corrupted, i.e. we *assume* that it writes a corrupt value to the memory, and study the memory corruption propagation. This is achieved by associating a *corruption* bit with every memory word and register and propagating the *corruption* bits as the program executes. using the propagation rules detailed in Table. 5.2 (see Fig. 5.1 2nd phase, steps 9,10). When an instruction experiences an exception (for example an out-of-bounds load), the target of that faulting instruction is marked corrupt. When a corrupted location is used by an instruction, then the value computed (or defined) by that instruction in turn is marked corrupt. However, when an instruction whose uses are uncorrupted, redefines a value, then the redefined value is considered to be uncorrupt, since it is computed with valid

```
Phase 1: Trace Stores

Let cnt: global instruction count

1.  switch (instruction)
2.    cnt ++                                  // Update Current instruction count
3.    case store:
4.      fprintf (trace-file, cnt)             // Write current instruction count to trace file

Phase 2: Memory Corruption Propagation

For each memory word addr, register reg,
src.corruption : Whether src (address/reg) is corrupted

1.  while ( !eof (trace-file) )
2.    fscanf (trace-file, propagate_cnt)      // Read count from trace file and start new execution
3.    switch (instruction)
4.      cnt++                                 // Update current instruction count
5.      case target = src1 op src2            // Target, src1, src2 can be memory / register
6.        if (cnt == propagate_cnt)
7.          propagation = true                // Start propagation mode
8.          target.corruption = true          // Initial Corruption
9.        if (propagation)
10.           if (src1.corruption or src2.corruption)
11.             target.corruption = true      // Propagation semantics
12.       target = src1 op src2               // Regular semantics
```

Figure 5.1: Algorithm for Memory Propagation Study.

operands. Finally, when memory (stack or heap) gets deallocated it is marked uncorrupt. Thus for a *load* instruction that moves a memory value into a register, the corruption bits corresponding to the memory word are also moved into a register. Similarly, the corruption bits are cleared when memory gets deallocated.

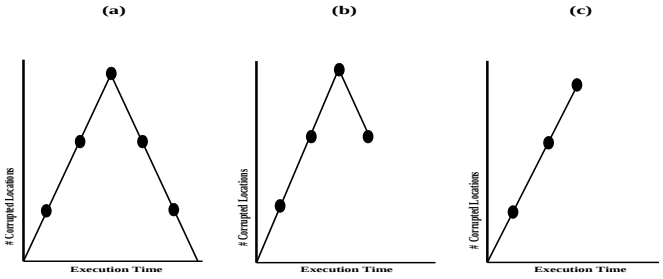


Figure 5.2: Variation of Corrupted Memory Locations with Time.

Observations and Analysis: When a memory value is corrupted, it can go on to corrupt other memory locations; at the same time, the memory locations that are marked corrupt can revert back to being uncorrupt, as memory gets deallocated. Fig. 5.2 shows

different ways in which the number of corrupted memory locations varied, after the initial memory corruption. One common pattern that was observed was that, a memory location once corrupted, *marks zero or more memory locations* corrupt, *each of which* revert back to uncorrupted state, as that respective memory locations are deallocated (see Fig. 5.2(a)). On the other hand, Fig. 5.2(b) shows the situation in which not all of the memory locations that are corrupted, revert back to uncorrupt state. Similarly, Fig. 5.2(c) shows the situation in which none of the memory locations that are corrupted, revert back to uncorrupted state. This can happen when the corrupted variables are those variables that are used *across user requests*, in which case they are not deallocated.

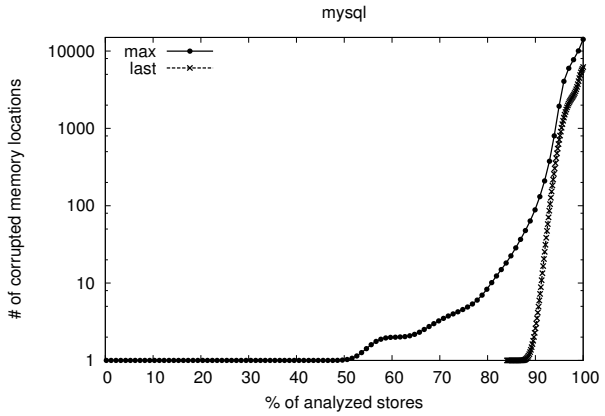


Figure 5.3: Variation in Max Corrupted and Final Corrupted across different execution instances for *mysqld*.

To get a quantitative perspective of the propagation of memory corruption, *the maximum number of corrupted memory locations* and *the final number of corrupted locations at the end of the current user request* was measured. For each of the above metrics the *min, max, mean and median* across different executions was measured, where different memory locations are marked corrupt as shown in Table. 5.3. To study how the above values are distributed across different executions, the values for each of the benchmarks were also

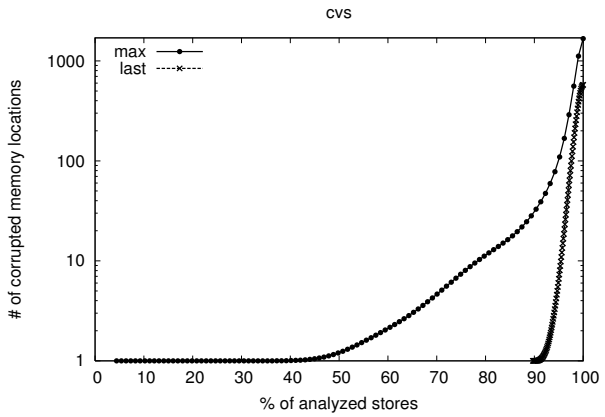


Figure 5.4: Variation in Max Corrupted and Final Corrupted across different execution instances for *cvs*.

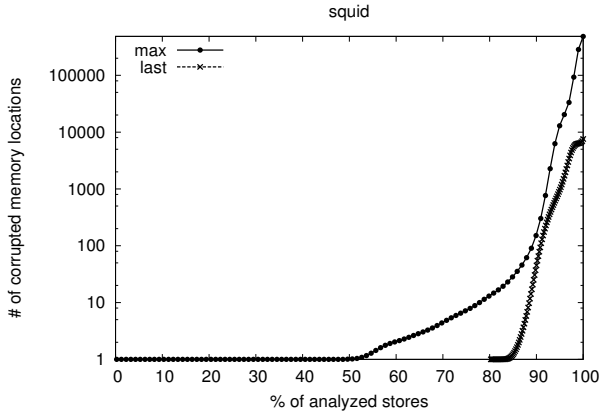


Figure 5.5: Variation in Max Corrupted and Final Corrupted across different execution instances for *squid*.

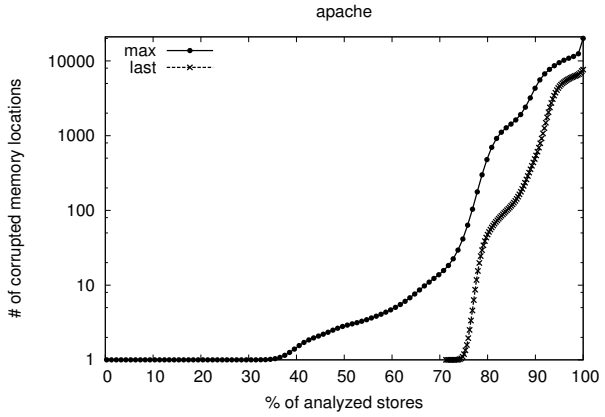


Figure 5.6: Variation in Max Corrupted and Final Corrupted across different execution instances for *apache*.

Table 5.3: Memory Propagation Study.

Program	Max Corrupted Location				Final Corrupted Locations			
	Min	Max	Mean	Med	Min	Max	Mean	Med
mysqld	1	14241	377	1	0	4995	120	0
cvs	1	1672	56	1	0	577	18	0
squid	1	475771	6997	2	0	8680	423	0
apache	1	32672	3228	5	0	6631	607	0

plotted as shown in Figs. 5.3 through 5.6. Each graph shows the variation in the max corrupted values and final number of corrupted values over different execution instances, in each of which an unique store is assumed to write a corrupt value. From Table. 5.3 and Figs. 5.3 through 5.6 the following observations are made:

- The minimum value of maximum number of corrupted locations (minmax) across all benchmarks is 1; this is the one that is initially marked corrupt and does not corrupt any more new locations. This can happen for several reasons. One possible reason is because the corrupted value can be used as a loop counter, in which case the corruption does not propagate across other memory locations. Another possible reason is because the stored value is sometimes not read at all, in which case no propagation occurs.

- We also observe that the minimum value of final number of corrupted locations (min-final) is 0 . This corresponds to the case in which each of the memory locations that were marked corrupt were reverted back to uncorrupted state. This can happen due to two reasons. First, those memory locations that have been marked corrupt could have been deallocated; second, the values that have been marked corrupt could have been overwritten with uncorrupt values.

- The maximum number of maximum number of corrupted locations (maxmax) can

be large. This can happen if some of the important variables that has several uses is marked corrupt. However, as we can see from Figs. 5.3 through 5.6 often the maximum number of corrupted locations is quite low. For 80% of the execution instances, less than 10 memory locations get corrupted.

- The maximum number of final number of corrupted locations (maxfinal) is relatively lower than maxmax. This is because of the *self cleansing* effect in server programs; this causes a large number of memory locations that are marked corrupt are reverted back to uncorrupt state. This fact is confirmed from Figs. 5.3 through 5.6 where the final number of corrupted locations is significantly lower. In fact, between 70% and 90% of the execution instances, the final number of corrupted memory locations is 0, meaning whatever memory locations that were marked corrupt were fully cleansed.

- This fact is further reinforced when the median of the maximum number of corrupted locations (medmax) and final number of corrupted locations (medfinal) is observed. While medmax is *less than 5* across all benchmarks, medfinal is 0. This means that more often than not, a corrupted memory location goes on to mark only few other additional memory locations as corrupt, each of which are reverted back to uncorrupted state.

Thus the most important insight that was inferred from the memory corruption propagation study is that a memory location that is marked corrupt goes on to corrupt only a few other memory locations, most of which are uncorrupted by the end of processing of the user request. This is what is known as the *self cleansing* property inherent in server programs.

5.2.2 What causes self cleansing?

Next, another study was conducted to find out the reasons for the above observations. In particular, the reason for this study is to find *why* self cleansing takes place in server programs. Why does memory corruption spread and then diminish rapidly as the server begin to handle the next request. One possible reason could be that user requests are already isolated, in that, there is very little data that is shared between user requests, which can cause memory locations corrupted during the processing of one user request to become invisible for other user requests. So, this study was conducted to find out if this is indeed true.

```
Memory Isolation Study

Let curr-id : Current user request id
For each memory word addr:
addr.request-id: User request that wrote to it last
shared : set of shared memory locations

1.   switch (instruction)
2.       case store:
             addr.request-id = curr-id
3.       case load:
             if (addr.request-id < curr-id)
                 shared = shared U addr
```

Figure 5.7: Algorithm for Isolation Study.

Methodology: The purpose of this study is to determine the degree of isolation among user requests already inherent in server programs. In this study, the server was connected with several user requests, the number of memory locations that are *shared* across user requests was determined. A memory location is said to be shared if it was written into by an earlier user request, and read by a later request; in other words if a memory location is used to exercise an inter user request RAW dependence, it is considered

shared. A small percentage of shared memory locations, would mean that values written during the processing of one request , is not read during the processing of other user requests and thus can explain why *self cleansing* takes place. To determine the number of shared memory locations, stores and loads are instrumented as shown in Fig. 5.7. Each memory location is tagged with a shadow memory location; each store is instrumented to store the current user request id in the shadow memory location associated with the original address. Each load is instrumented, to check if the value loaded comes from a previous user request, by checking the user request id; if that is the case, that particular memory address is added to the set of shared addresses.

Observations and Analysis: The percentage of memory locations that are shared across several user requests is measured in each of the server programs. As we can see from Table. 5.4 only a small percentage of memory locations (ranging from 6% to 35%) were shared across user requests. The effect of the complexity of user requests on the number of shared memory locations was also measured. Results are presented for two programs *mysqld* and *squid*. For *mysqld*, the number of shared memory locations was measured, as the size of the tables are varied. For *squid*, the sizes of the webpages that squid fetches are varied and the effect on the number of shared memory locations is measured. As we can see from Figs. 5.8 and 5.9: as the complexity of user requests increases, the number of non-shared memory locations increases rapidly, while the number of shared memory locations almost remains the same. This evidence points to the fact that server programs have a fixed *global state* that is shared across user requests.

Thus the most important insight that was derived from this study is that most of the values that are written during the processing of a user request are used locally; only a small

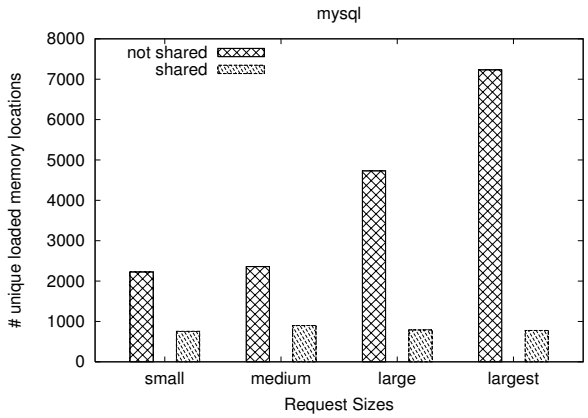


Figure 5.8: Variation in Shared/Unshared with complexity of user requests for *squid*.

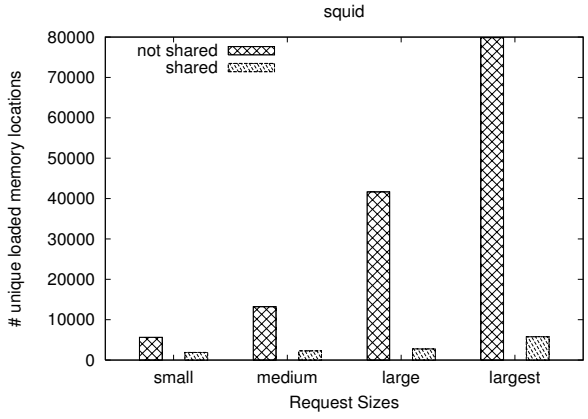


Figure 5.9: Variation in Shared/Unshared with complexity of user requests for *mysqld*.

Table 5.4: Isolation Study.

Program	# Shared	# Non-Shared	% of Shared
mysqld	758	6493	10.4
cvs	271	1457	15.6
squid	2753	41650	6.2
Apache	1471	2646	35.7

(fixed) amount of global state is shared across user requests. The small shared state is the reason why memory corruption does not propagate across user requests, and hence explains self cleansing.

5.3 Design and Implementation of SRS

In this section, the design and implementation of our technique SRS, for recovering from server failures related to memory corruption, is discussed in detail. First, the general idea of SRS is discussed at a high level. Then the steps involved in implementing SRS at a conceptual level is discussed in detail.

When a server program experiences a failure while processing a user request, ideally the server should not crash; on the contrary, the server program should continue to process future user requests. At the same time, the memory state that has been corrupted by the fault inducing memory request *should not* be visible to future user requests. In other words, the faulty user request should be *isolated* from the processing of other benign user requests. One way to perform this is to checkpoint memory and architectural state before processing every user request; upon the detection of a failure, the state can then be rollbacked to the prior benign state. The objective of SRS is to ensure semantics similar to rollback based recovery scheme, without performing checkpointing or rollback. For this SRS takes

advantage of *self cleansing* inherent in server programs. Recall that from the study it was inferred that different user requests of sever programs share very little shared state, owing to which memory corruption that happens during the processing of a user request is largely invisible to future requests. Thus the main steps in SRS are two fold:

- Execute past a failure, when a failure is detected, instead of rolling back, so as to trigger *self cleansing*. We enable execution past a failure by executing instructions under *crash suppression* mode – under suppression mode, instructions, any of whose source operands are corrupt, are *not executed* and the target is *marked corrupt*.

- Despite self cleansing there can be small number of memory locations that remain corrupted – if there is a need to access such corrupted locations later (by a later request), a *demand driven* approach is used to restore the corrupted values only when it is needed to be read. In SRS, the set of memory locations that are potentially shared across user requests are identified via profiling; these specific memory locations are specially monitored, so that in case of a failure *these specific locations are restored in a demand driven fashion*.

The rest of this section is organized in follows. The crash suppression semantics and its implementation is discussed in detail. Then it is discussed how isolation is realized with demand driven restoration of shared memory locations. Then it is briefly discuss how SRS maintains thread safety for multithreaded code. Finally, each of the above steps is integrated and present as the SRS technique.

```
For each memory word addr, register reg
   src.corruption : Whether src (address/reg) is corrupted

1.    switch (instruction)
2.       case faulting instruction:
3.          suppression = true                  // Start suppression mode
4.          target.corruption = true            // Initial Corruption

5.       case target = src1 op src2
6.          if (suppression)

7.             if (src1.corrupt or src2.corrupt)
8.                target.corrupt = true         // Suppression semantics
9.             else
10.               target = src1 op src2         // Regular semantics
```

Figure 5.10: Suppression Semantics.

5.3.1 Crash Suppression

Once a failure is detected while processing a user request, in SRS, execution is continued forwards so that we can *take advantage of the self-cleansing property*[1]. However, executing forwards past a crash is not without its own challenges. Even though the first crash can be suppressed, similar crashes can recur (and likely will recur), when values that are dependent on the corrupted memory values are used later. We enable execution past a failure using *crash suppression semantics*, in which an instruction is *suppressed* without executing, if any of its source operands are corrupted. The basic steps for realizing *suppression* semantics are outlined in Fig. 5.10. We implement suppression by associating a *corruption* bit with every memory word and register. Upon detecting a failure, the *corruption* bit for the target of the instruction that causes the failure is set. Suppression semantics entail that any instruction, any of whose source operands are marked corrupt, is suppressed and is not

[1]For this work, failure refers to an OS *out-of-bounds exception*, although others sensors [72] can be used to detect a failure

executed. However, the corruption bit is propagated, which means the target operand's corruption bit is set. If the corruption bit for a branch predicate is set, the control is made to skip the whole branch structure; For instance if the predicate for an *if-then-else* structure is marked corrupt, then both the *then* and the *else* parts of the structure are skipped. Dealing with indirect jumps (and returns) whose target address is marked corrupt is more tricky. In our current implementation, profiling is used to figure out the most frequent branch target for such indirect jumps and jump to that target. In case profile information is not available, we directly jump to the next user request.

5.3.2 Ensuring Isolation

We need to ensure that an user request, that encountered a failure, should be isolated from future benign user requests. This entails that values written to memory during the processing of the faulty request should not be visible to future requests. Those memory locations that are shared across user requests are those which can possibly be visible to future requests. Fortunately, our study shows that the number of shared memory locations are relatively small. In SRS, memory locations that are likely shared using profiling are identified. During normal run, these shared memory locations are monitored and multiple versions of these shared memory locations are maintained. Thus, when an user request experiences a failure and there is a need to access one of these shared memory locations, the corrupted value can be restored to its original uncorrupted state, on demand. However, it is worth noting that profiling only gives us an *underestimate* of the set of memory locations that are shared across user requests. Thus, a situation could be potentially encountered when there is a need to access a corrupted shared memory location, which has not been

identified and monitored. Our approach to deal with this situation is to provide capability to *detect* such a situation. When such a situation arises, the server is restarted; by doing this it is ensured that recovery is *fail safe*. Having explained our approach for ensuring isolation at a high level, now let us consider in detail the individual steps.

Monitoring Shared Locations

The memory locations that are potentially shared across user requests are found using profiling. In the profiling run we connect to the client with several user requests, and then identify the memory locations which are written to by an earlier request and read by a later request. In other words, the set of memory locations which are used to enforce *true dependences* across user requests are identified. These are the memory locations in which global state is maintained. For instance, a global variable which essentially stores the *number of user requests handled* will have this property. These set of memory locations are known as the *TrackSet*. It is worth noting that this is only an underestimate, since profiling is used to identify this set. Nothing is done regarding anti and output dependences; this is because these locations are overwritten with a new (uncorrupt) value in the later request.

Let us see how the TrackSet is monitored and how different versions are maintained for the memory locations in the TrackSet. Whenever the processing of a new user request starts, memory (the *TrackLog*) is allocated to hold the previous values of the memory locations within the TrackSet. The TrackLog is a buffer, each entry having two values: the address and the (previous) value. All stores operating on the TrackSet are instrumented so that their prior values can be maintained in the TrackLog. If the current user request does not experience a fault, then the TrackLog can simply be discarded. However, if the

user request experiences a fault, the TrackLog is used later to restore the corrupt memory locations to their prior values, on demand.

Demand Driven Restoration

The main idea of demand driven restoration, is to restore the corrupted value using the TrackLog, when it is is about to be read in a future request. To be able to do this we first need to identify that a value is corrupt, when it is about to be accessed. A value is considered corrupt, if it was written into during the processing of a fault inducing user request. To identify such memory values, every memory location is shadowed [2] with a *request-id*. This is essentially a unique number associated with every user request. All stores are then instrumented to additionally write the *current request-id* to the shadow location associated with the memory address.

When a user request experiences a failure, this is remembered by adding the current request-id to the list of *failed requests*. Once a failure is encountered, all loads in the program are instrumented to perform an additional check. By comparing the request-id associated with the loaded memory location with the list of failed user request, we are essentially checking if the value loaded comes from a value that has been stored during the processing a fail request. If so, the TrackLog is consulted using the effective address as an index into the TrackLog. An entry in the TrackLog means that this memory location has been identified during profiling and the value has been backed up in TrackLog. Accordingly, the value in then restored from the TrackLog into the actual memory location. In other words, corrupted values are restored on demand, when they are accessed.

[2] It is sufficient if the heap and global space are shadowed, since stack memory locations are not used to enforce true dependences across user requests

However, it is important to note that the TrackLog is only an underestimate of the actual set of values that is shared across user requests. Hence it is possible that the TrackLog does not have an entry if the memory location had not been identified as a part of TrackSet during profiling. If this is the case, the server is restarted to ensure fail safety.

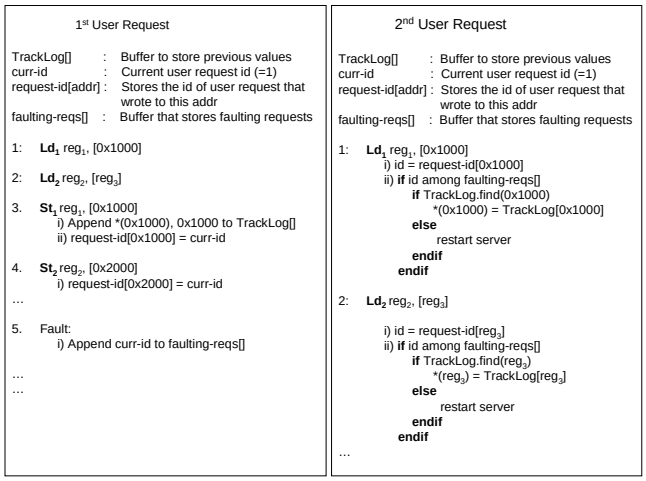


Figure 5.11: Ensuring Isolation.

An Example

In this section, the steps involved in ensuring isolation are summarized with a simple example. Let us consider two user requests the first of which experiences a fault while being processed. The first step is to identify the *TrackSet* using profiling. Let us assume that during profiling run, there is only one memory location ($0x1000$) that is shared between two user requests, and it is the one that is written into by St_1 and read by Ld_1 across user requests as shown in Fig. 5.11. Consequently, St_1 is instrumented with code that stores the prior value residing in the memory location to the TrackLog associated with

the user request (step 3). Since St_2 does not write to the TrackSet, it is not instrumented in the above fashion. However, it is instrumented to write the current request-id in the shadow memory associated the memory location (step 4). Let us assume that the program then experiences a fault while executing a subsequent instruction, while processing the same user request. Upon a fault (step 5), the current request-id is added into the list of failed user requests.

Now let us consider the processing of the next user request, and in particular the execution of the two loads Ld_1 and Ld_2. Ld_1 which obtains its value from St_1, now gets it value from a store that was executed during the processing of a fault inducing request (request 1). In other words, the value to be loaded is corrupt and the condition (step ii) evaluates to true. Since the value is corrupt, the TrackLog is searched for the value. As the value was already appended in step 3 of the 1st request, it is found there. Consequently, the value is restored from the TrackLog and the correct (uncorrupt) value is loaded. Now, let us consider the execution of Ld_2. Let us assume that during this run Ld_2 actually gets its value from St_2 since the value of reg_3 happens to be $0x2000$ during this run. This is an instance of a dependency across user requests that was not captured during profiling. Consequently, the value is not found in the TrackLog and the server is restarted.

Handling Multithreaded Code using ECMon

Since the SRS technique is associated with meta data for every memory location (for eg. request id) and includes software instrumentation associated with memory instructions, races present in the source can lead to meta data inconsistency [16]. For example, let us consider Fig. 5.12 which illustrates the atomicity issue. Recall that each store is

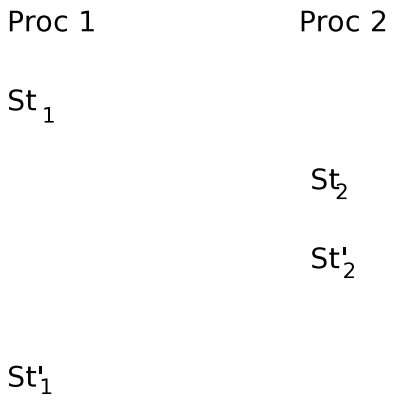


Figure 5.12: Atomicity issue.

associated with a shadow store that writes the request id to the shadow memory location. In the above example, while St_1 from processor 1 executes before St_2 from processor 2, the corresponding shadow stores are executed in the opposite order. This means that while the value written by St_2 is the value that is written last, the request id is the one corresponding to the request from processor 1. In SRS, this problem is dealt with by serializing the threads and making sure that thread switches do not occur between data and meta data updates as in Valgrind [64]. However it is worth noting that thread serialization is inefficient as it forces the code to run on a uniprocessor. To make SRS applicable for server programs running on multicores, ECMon support is utilized to enforce atomicity (as described in chapters 3).

5.3.3 SRS Summary

Each of the steps of SRS are illustrated in the concise algorithm shown in Fig. 5.13. The steps of SRS are roughly divided into four phases. In the first profiling phase, which is performed offline, the *TrackSet* is determined and all the stores that write to the TrackSet

Offline: Normal Run	Online: Fault Detection
1. Perform profiling and determine TrackSet. 2. Determine the Stores that operate on TrackSet. Online: Normal Run 1. For stores that operate on TrackSet maintain their previous values in TrackLog. 2. Instrument other stores to write current request id to shadow memory.	1. Catch OS exceptions for access violations. 2. Append current user request to set of faulting requests. 3. Enter Suppression mode until the end of current request. Online: Recovery Run 1. Instrument loads to check if they were written during faulting request 2. If so, check if the available in TrackLog. If available, restore it, otherwise restart.

Figure 5.13: Summary: SRS.

are determined. In the normal run, the TrackLog is maintained for all stores that write to the TrackSet. For all other stores, the current request id is written out to the shadow memory to assist on-demand restoration. It is worth noting that the online overhead imposed by SRS is the overhead of executing the additional instrumentation. The next phase consists of the fault detection. In this phase, the OS *out-of-bounds exception* is used to denote a fault. However it is worth noting that other sensors including security attack detection tools can be used in this step. Once the fault is detected, the current user request is added to the set of faulting requests, and execution enters suppression mode. In the final recovery phase, loads are instrumented to check if the request id corresponding to the loaded value comes from a faulty user request. If this is the case, then the TrackLog is searched and the correct value is restored on demand. If the entry is not found in the TrackLog, the server is restarted.

5.4 Experimental Evaluation

The experimental evaluation of SRS was performed with several goals in mind. First and foremost, the efficacy of SRS in recovering from real faults in server programs in evaluated. At the same time, the overheads imposed by SRS in the normal run and during recovery was also investigated. But before the results of our experiments are presented, the implementation of the prototype is discussed.

5.4.1 Implementation

A functionally working version of SRS was implemented in *Valgrind* [64] and was used for conducting the study. Valgrind's shadow memory support [63] was used for storing the various meta data information used for implementing SRS. The starting instruction address, where processing of every user request commences, was manually identified. Once this address is encountered the Tracklog is allocated and the current request id is incremented. With Valgrind, the out-of-bounds OS exception was captured, and then execution continued in suppression mode. After completing the faulty user request and encountering the start address of the next request, the execution leaves suppression mode and enters recovery mode, where every load includes the fail safety check. However, since the Valgrind infrastructure, which is built for the ease of writing complex tools, imposes higher overheads, a performance optimized version was also implemented in *dynamoRIO* [10]. This version of SRS, which was used to measure the overhead of SRS, includes all instrumentation of the prior version, but does not catch OS exceptions, so it has to be manually run in normal mode or recovery mode.

The overhead of SRS with ECMon support while monitoring programs running on

multicores, was also evaluated. For this the SESC [80] simulator was used. Recall that the ECMon support was built into the SESC simulator. However, the server programs could not be run with the simulator infrastructure. Hence, the SPLASH2 [110] benchmarks were used in its place. It is important to note that these experiments were carried out merely to estimate the overheads of performing SRS on multicores and there was no fault in the programs themselves.

5.4.2 Recovery in the presence of faults

Versions of the server programs with real memory errors were used for this experiment. The bugs in the program are described in the Table. 5.5.

For each of the programs, the server was connected with about 10 user requests, with the special user request that triggers the fault as the 5th request. Different user requests compared to the one used in profiling runs were used. In each of the cases, the bug causes an out-of-bounds OS exception which SRS catches; then, SRS enters suppression mode under which SRS was able to safely execute to the end of the faulty request, without experiencing other faults. Once the faulty request is "processed", SRS enters recovery mode in which fail safety checks are added before every load. In our set of experiments, it was observed that none of the fail safety checks failed; thus the need to restart the server never arises. We believe that typically, the need to restart will not occur, since the shared variables stay mostly the same irrespective of the variation in the user request. Thus this experiment shows that SRS can be used to survive faults safely.

Table 5.5: Bugs in Server Programs.

Program	Bug
mysqld	Uninitialized Read [3]
cvs	Double free [42]
squid	Buffer overflow [42]
Apache	Stack Overflow [42]

5.4.3 Performance of SRS: uniprocessor

In this experiment, the overhead experienced in the response time imposed by SRS was measured during normal run as well as during recovery. Recall that the overhead imposed during normal run is due to the additional instrumentation involved for maintaining the TrackLog and for storing the current user request id for every original store instruction. As we can see from Fig. 5.14, the overhead imposed is very low, on an average, 5% across all benchmarks. This overhead is low mainly for two reasons. First, since these are not computationally bound programs, the additional instrumentation could easily be tolerated. Second, the additional instrumentation during normal run, is only an additional store for every store instruction. Since the processor does not generally wait for the stores, the overhead imposed is not high.

We also measured the additional overhead imposed after recovery, which is basically the overhead for performing the fail safety checks along with every load. Even this overhead is pretty low, on an average 8% across each of the benchmarks (Fig. 5.15). The overhead is higher because now the instrumentation is performed for every load along with a safety check.

The overhead involved in executing in suppression mode was also measured. As we can see, from Fig. 5.16, this overhead can be as high as 3 times for these benchmarks.

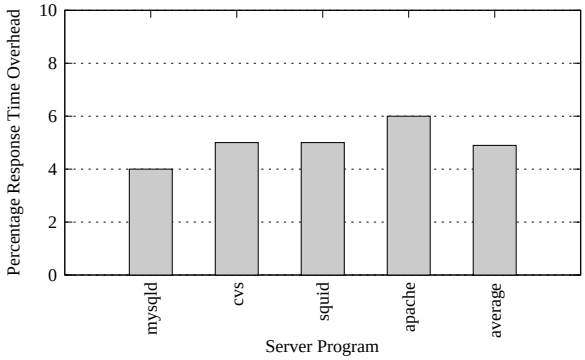


Figure 5.14: Response Time Overhead in Normal Run.

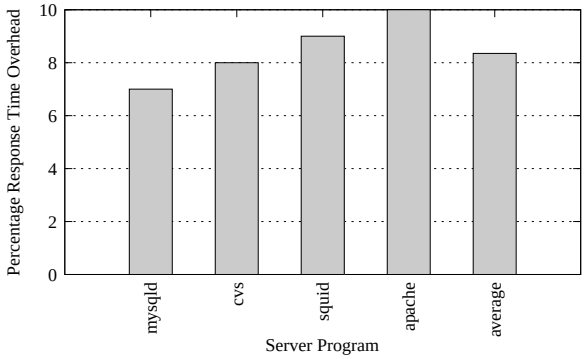


Figure 5.15: Response Time Overhead after Recovery.

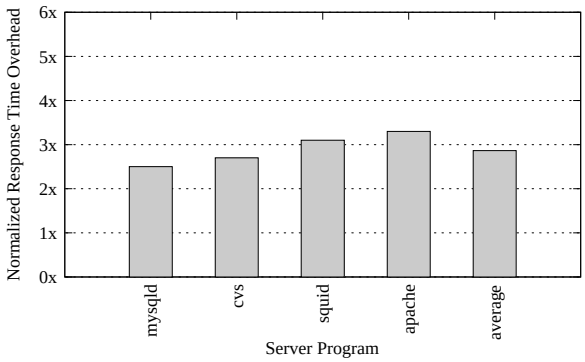


Figure 5.16: Response Time Overhead during Recovery in Suppression Mode.

However it is important to note that the overhead for performing suppression is only for the duration of processing the *faulty request*, hence the dip in performance is only applicable for a very short time. Once the faulty request is handled, then *SRS stops executing in suppression mode*. Furthermore, there has been significant research on performing dynamic information flow tracking efficiently using hardware support [97, 21, 23]. DIFT hardware can be used to optimize the performance of the suppression mode, since the instrumentation operations performed during suppression mode resemble those that are performed during dynamic information flow tracking.

5.4.4 Performance of SRS: multicore

The overhead of SRS while executing on a multicore using ECMon support was also measured. This experiment was carried out using our simulator infrastructure. Since the server programs could not be compiled for this infrastructure, we used the popular SPLASH2 benchmarks for this study. In this experiment, the overhead of SRS during normal run was measured.

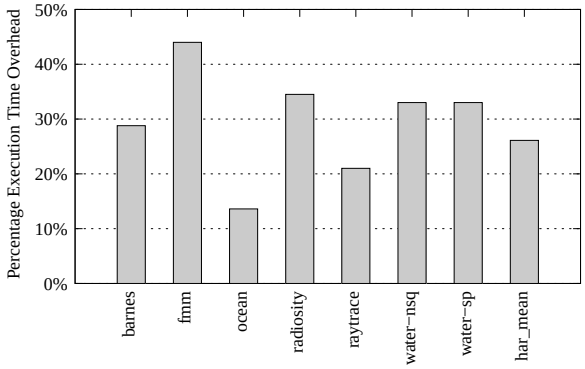


Figure 5.17: Response Time Overhead in Normal Run.

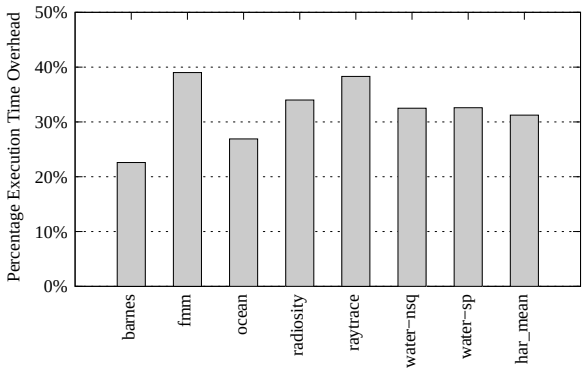


Figure 5.18: Response Time Overhead after Recovery.

As we can see from Fig. 5.17, the overhead of SRS, when run on multicores is 27% on an average, while the overhead during recovery in 31% on an average. The overheads are higher than those of server programs, since these programs are cpu intensive and it is not easy to accommodate additional instrumentation without incurring overheads. It is important to note that the atomicity is required only during normal run and not during recovery. This is because during recovery, the instrumentation consists of loads for every load original load instruction and hence there is no possibility of races between data and meta data accesses. However, the overhead during recovery is higher due to the fact that there are greater loads than stores in these programs and thus, there is need for greater number of instrumentation instructions executed during recovery.

5.4.5 Performance of Checkpointing/Rollback Schemes

One issue with Checkpointing based rollback recovery schemes is the frequency of checkpointing, which in turn results in the trade-off between the normal execution performance and recovery performance. Infrequent checkpointing can reduce the overhead of

checkpointing, but can increase the cost of recovery [72]. On the other hand, frequent checkpointing can cause greater overhead during normal execution. A highly optimized checkpointing system [95] with a checkpointing interval of 50ms resulted in a overhead of about 11% during normal execution run, with marginal overheads during recovery. The overheads incurred by SRS are thus comparable (slightly lesser) to overheads experienced in a checkpointing based rollback recovery scheme, without the need for a complex checkpointing/rollback system.

5.5 Summary

In this chapter, a technique known as SRS that was used to retain the availability of server programs amidst software failures caused due to memory errors was presented. SRS was motivated by a detailed study that was conducted to see how memory corruption propagates in server programs. The study showed that memory locations that are corrupted during the processing of an user request, generally do not propagate *across* user requests. On the contrary, the memory locations that are corrupted are generally *cleansed* automatically, as memory (stack or the heap) gets deallocated or when memory gets overwritten with uncorrupted values. This self cleansing property in server programs led us to believe that recovering from crashes does not necessarily require the expensive roll back of state for recovery. Motivated by this observation, SRS, a technique for self recovery in server programs which takes advantage of *self-cleansing* to recover from crashes was proposed. Experiments conducted on real world server programs with real bugs, showed that in each of the cases the server program could efficiently recover from the crash and the faulty user request was isolated from future benign user requests. Performance evaluations revealed that SRS could

efficiently recover from a crash, and causes nominal overhead of about 5% during normal run when considering executions on a uniprocessor. Since races between data and metadata accesses make a direct implementation of SRS unsuitable for multicores, ECMon support was used to adapt it for multicores.

Chapter 6

Related Work

6.1 Software based Monitoring

Software based monitoring techniques instrument the program with additional code for enabling monitoring. While software based monitoring requires no special hardware support and can be applied to a variety of monitoring tasks, they cause significant program slowdown. In general, software monitoring tools take advantage of a *dynamic binary translator* (DBT) [43, 64, 108] for adding the additional instrumentation code for monitoring. *Dynamic taint analysis* [66], *LIFT* [73], *TaintTrace* [15] and *Dytan* [19] are tools that perform dynamic information tracking. *Flashback* [95] and *Jockey*[85] are tools that enable deterministic replay debugging by recording and replaying program execution. *Memcheck* [63] and *eraser* [88] are debugging tools; while the former is used to debug memory errors, the latter is used to detect data races in multithreaded programs. *Redux* [62] and *OnTrac* [58] are tools that trace a program's execution as it executes. Often software monitoring tools use *shadow memory* [63] to maintain meta data for every memory location

of the original program; for this reason such tools [66, 63, 88], are known as shadow memory tools. Each location in the original program is associated with a shadow memory location, where the meta data is maintained. Allocating and maintaining shadow memory in a robust and efficient fashion is one of the important challenges facing software monitoring techniques [63].

Although there are several software based monitoring tools, they are thwarted by ISMDs, which makes them either inapplicable [66, 73, 95] or inefficient [25, 63] on multicores. For example, both *Flashback* [95] and *Jockey*[85] are only applicable for (single and) multithreaded programs running on a uniprocessor. Likewise, software based taint analysis tools [15, 66, 73] are unable to monitor multithreaded programs running on multiprocessors [16]. Software tools that perform monitoring need separate instructions to perform monitoring, giving rise to races between data and meta-data when executed on multicores [16]. Since the ISMDs can be captured using *ECMon*, the proposed support in this dissertation, these races can be dealt with in our dissertation. This dissertation shows how exposed cache events could be used to program a variety of software monitoring tools, including shadow memory tools [55, 53, 52] and tools for performing DRD [54]. Furthermore, this dissertation also shows how shadow memory, which is used in several software monitoring tools, can be managed robustly and efficiently with support for managing and addressing shadow pages [55, 53, 52].

6.2 Hardware based Monitoring

Hardware based monitoring techniques use hardware support for enabling monitoring. While hardware based monitoring tools are generally faster than software based

monitoring tools, they can not be directly used in real machines. Hardware based monitoring techniques, can be coarsely divided into two two categories: those that use *specialized* hardware support for tackling one single monitoring problem (or a small class of problems), and those that use *general purpose* hardware support that can be programmed for tackling a large class of monitoring problems.

6.2.1 Specialized hardware support

There has been several proposals that use specialized hardware support to ensure the reliability of software [23, 60, 61, 97, 111]. The hardware support involved in each of the above proposals is non-trivial and involves changes to the processor pipeline, caches, cache coherence and memory subsystem. For example, *FDR* [111],*Bugnet* [61], and *Strata* [60], which are hardware tools for recording the programs execution, involve augmentations to the cache coherence protocol to capture dependences, changes to processor pipeline to maintain instruction counts, addition of per block counters to the caches and addition of other hardware structures to optimize recording. *Rerun* [35] and *DeLorean* [48] represent recent work in the above area, that optimize the hardware requirements for performing the recording. Similarly, *DIFT* [97], *Minos* [22], *Raksha* [23] and *flexitaint* [28] use specialized hardware with changes to processor pipeline and memory subsystem to perform dynamic information flow tracking (DIFT). Recently there has been work to utilize support for data speculation in processors to perform DIFT [13]. In this dissertation, the minimal hardware support needed (ECMon: exposing cache events) is isolated, so that all other tasks required are performed in software. This approach, in addition to increasing the flexibility and programmability, makes *ECMon* applicable to a variety of monitoring tasks.

6.2.2 General purpose hardware support

Recently there has been work that strives to design general purpose hardware support for a variety of monitoring applications [14, 59, 90, 105]. Chen et al. [14] and Venkataramani et al. [105] propose general purpose hardware support for enabling a variety of monitoring techniques such as Memcheck, Addrcheck and data race detection. The hardware support used in the above works involves support for accessing shadow memory location efficiently and performing additional operations in hardware along with original instructions. Shetty et al. [90] and Nagarajan et al. [59] propose using idle cores in a multicore processor for performing monitoring tasks. To communicate between cores, they utilize hardware based communication channels between cores in multicores. However, the above techniques concentrate on monitoring that target sequential programs. For example, none of the above techniques can be directly used to record ISMDs. On the contrary, the techniques proposed in this dissertation can handle a wide variety of monitoring tasks including ones that are geared towards parallel programs running on multicores.

6.3 Transactional Memory

The problem of detecting cross-thread dependence violations at run time is known as *conflict detection* under *Transactional memory* (TM) [33] parlance. STM systems [6, 44] instrument loads and stores with *read/write barriers* to detect conflicts. On the contrary, HTM systems [32, 33, 75] like TLS systems [18, 32] rely on hardware support (modifications to caches/cache coherence) to detect conflicts. Hybrid TMs [24, 47, 92] use hardware to perform the simple and common case and rely on software support to handle the uncommon

case. Recent proposals on hybrid TM have proposed hardware support for conflict detection. While *SigTM* [47] uses hardware signatures for conflict detection, *RTM* proposed the *Alert-on-Update* [94] mechanism which triggers a software handler when specified lines are modified remotely. Whereas the hardware support involved in *ECMon* is similar to *Alert-On-Update*, it is shown how other cache events (in addition to remote update), can be used for performing a variety of monitoring applications including speculation bast barriers and recording of ISMDs. There has been a recent proposal [16] to use transactional memory support for dealing with the atomicity problem facing software monitoring tools. However, it does not discuss the efficient addressing of shadow memory which is also an important inefficiency in current software based shadow memory tools.

6.4 Speculative Techniques

6.4.1 Speculation past synchronization operations

The *Fuzzy Barrier* [31] is a compile time approach to reduce time spent idling on barriers by specifying a range of instructions over which the synchronization can take place instead of a specific point where the threads must synchronize. However, this approach relies on the compiler to find instructions that can be safely executed while a thread is waiting for others to reach the barrier. *Speculative lock elision* [74] and *Speculative synchronization*[46] are hardware techniques to speculatively execute threads past synchronization operations. While the former dynamically converts lock-based codes into lock-free codes, the latter also applies to flag synchronizations and barriers. The thread that has reached the barrier, speculatively executes past the barrier. Hardware support (addition of per block tags to

the cache, modifications to the cache coherence, support for register checkpointing) is used to monitor dependence violations between the speculate thread(s) that are executing past the barrier and other non-speculative threads that are yet to reach the barrier. If such a violation is detected, the speculative thread is rollbacked to the synchronization point. Our approach, on the contrary, detects miss-speculation using cache events.

6.4.2 Speculative parallelization

Recently there has been significant work on parallelizing sequential loops. One commonly used approach for parallelization of loops is software pipelining. This technique partitions a loop into multiple pipeline stages where each stage is executed on a different processor. Decoupled software pipelining (DSWP) [69, 76, 104] is a technique that targets multicores. The proposed DSWP techniques requires two kinds of hardware support that is not commonly supported by current processors. First, hardware support is used to achieve efficient message passing between different cores. Second, hardware support is versioned memory which is used to support speculative DSWP parallelization. Since DSWP requires the flow of data among the cores to be acyclic, in general, it is difficult to balance the workloads across the cores. Raman et al. [76] address this issue by parallelizing the workload of overloaded stages using DO-ALL techniques. This technique achieves better scalability than DSWP but it does not support speculative parallelization which limits its applicability. Other recent works on software pipelining target stream and graphic processors [36, 11, 26, 37, 102]. In this dissertation we show that *ECMon*, support for exposing cache events, is sufficient for exposing parallelism in programs. However, it remains to be seen as to how exactly will ECMon impact the performance of the above techniques.

The alternative approach to exploiting loop parallelism is DO-ALL technique [25, 39, 38, 18, 32, 106, 96, 9, 45, 114] where each iteration of a loop is executed on one processor. Among these works, a large number of them focus on thread level speculation (TLS) which essentially is a hardware-based technique for extracting parallelism from sequential codes [18, 32, 106, 96, 9, 45, 114]. In TLS, speculative threads are spawned to venture into unsafe program sections. The memory state of the speculative thread is buffered in the cache, to help create thread isolation. Hardware support is required to check for cross thread dependence violations; upon detection of these violations, the speculative thread is squashed and restarted on the fly. While the above HW support is specialized, in that, the support is useful only for parallelization, the support proposed in this dissertation *ECMon* is general purpose, in that it has other applications.

Recently there has been work on purely software based speculation techniques [17, 25, 103], where it was shown that coarse grained parallelism in sequential programs could be extracted without any hardware support. In this dissertation, we show additionally how fine grained parallelism can be extracted with support for exposing cache events. However, it remains to be seen if ECMon can be used to further optimize the above techniques.

6.5 Recovery in Server Programs

Recovering from failures has been a subject of significant research over the years. There has been significant work on coping with software failures by using various kinds of *rebooting* techniques. While whole program restart [30] works by simply restarting the failed application, a small set of partial software components may be selectively restarted [12] to reduce the cost of recovery. The problem with restarting techniques is that the server

program can be temporarily unavailable during restart. To reduce the cost of recovery, checkpointing based techniques[30, 71, 72, 77, 101], periodically checkpoint program state and rollback to the most recent checkpoint when the failure is detected. Having rollbacked to a safe state, that particular user request is then dropped [72, 101] so that the same failure is not repeated. However, the space and time overheads of checkpointing can be expensive if the checkpointing interval is small [71, 72]. On the contrary, if the checkpoint interval is large then the throughput and the response time of the server can be affected during recovery.

There has also been work on the reliability of *long running programs* using a combination of checkpointing and tracing [100, 112]. The scope of self recovery in this dissertation is to consider server programs, since there is a pressing need for them to be available. However, the techniques presented in this paper are also applicable for other long running programs, which process different user inputs continually.

There has been recent research on recovering from memory errors without the need for checkpointing or rollbacks [82, 93]. Self recovery in this dissertation is closely related to *failure oblivious computing* [82], which also observes and utilizes the *self cleansing* property for maintaining server availability amidst failures. In the above work, instead of crashing when an illegal memory access occurs, the server continues program execution by simply discarding the illegal writes and manufacturing values to return for illegal reads. The success of the above technique hinges on small error propagation distances on server programs, which is referred to as *self cleansing* in our work. However in this dissertation, instead of speculating the programmer's intentions (for example, by manufacturing values for reads), the faulty request is nullified using *crash suppression* and isolated.

Recovery oriented computing [68, 70] proposes a system in which software components of a system are designed to be isolated, so that the impact on failures can be reduced. The dissertation is related to work on recovering from failing device drivers [98, 99] in that the above works also try to build a system that tries to isolate the failing device drivers from other parts of the system. However, this dissertation shows how this isolation is already present is server programs.

There has been significant work on recovering from *Transient soft errors* [49, 79, 78, 109, 107], which are radiation induced errors that cause random bit flips in both the computational and memory hardware. Bit flips to computation circuitry and memory elements are a form of memory corruption, and the results of the study conducted in this work are equally applicable for transient errors. In particular, the observation that a corrupted memory location , most often, corrupts only a few other memory locations can be taken advantage by a system that recovers from transient errors.

Chapter 7

Conclusion

In this chapter the main contributions of the dissertation are summarized. Then the future directions of research are briefly discussed.

7.1 Dissertation Contributions

With the advent of multicores, there is a huge demand for programmers to write parallel programs to utilize the power of multicores. However, there is a huge effort involved in writing correct, efficient parallel programs. This has resulted in a huge demand for software tools that assist the programmers in writing correct and efficient parallel programs. Such tools include those that help in automatically parallelizing sequential programs, tools that help in exposing (additional) parallelism present in parallel programs, and tools that help in debugging and ensuring the secure execution of parallel programs. The above monitoring tools, since they require some form of dynamic analysis to be performed while the program is running, are known as runtime monitoring tools. This dissertation first observes that runtime monitoring on multicores require that interprocessor shared memory

dependences (ISMDs) be detected efficiently. However, current runtime monitoring tools, since they can not deal with ISMDs, are not applicable for multicores. This dissertation proposes support for enabling efficient and programmable runtime monitoring of parallel programs running on multicores. It has then been showed how this support can be used to increase the performance and enhance the reliability of parallel programs running on multicores. In particular the contributions of this dissertation are as follows.

- **ECMon:** Support for exposing cache events to software, has been proposed. By exposing cache events to the software, software is made aware of ISMDs. This enables software based monitoring of parallel programs on multicores. ECMon is light-weight, requiring minimal hardware changes. More specifically, ECMon requires no changes to the processor pipeline and the cache coherence protocol.

- **Shadow memory tools using ECMon:** ECMon is programmable and general purpose. This has been illustrated by implementing a variety of monitoring tools using ECMon support. In particular, it has been shown how a class of monitoring tools known as shadow memory tools, that include DIFT: tool for ensuring secure execution, Memcheck: tool for detecting memory errors and Eraser: tool for detecting data races, can be implemented efficiently using ECMon support. More specifically, it is shown how the above tools can be implemented for parallel programs at almost the same execution time overhead as sequential programs. It is also shown how lightweight OS support can further help in reducing the execution time overhead of accessing meta data stored in shadow memory.

- **Novel monitoring applications using ECMon** This dissertation has shown how ECMon can be used to develop novel monitoring application for parallel programs running on multicores. It is shown how ECMon can be used to record shared memory dependences as a parallel program executes, which can then be used to replay the program. Furthermore, it is shown that this can be done efficiently only resulting is 3 fold execution time reduction. It is also shown how ECMon support can be used by multithreaded server programs to recover from memory errors, without requiring checkpointing or rollback.

- **Speculation using ECMon** This dissertation has also shown that ECMon can be utilized to perform speculative optimizations in parallel programs, which can be used to increase the performance of parallel programs running on multicores. More specifically, this dissertation has shown how ECMon can serve as a framework for performing two speculative optimizations. First by speculating past barrier synchronizations, it is shown that the time spent idling at barriers can be decreased translating into a 12% increase in performance. Second, by speculatively promoting shared variables in the presence of synchronization operations, it is shown how significant redundant loads can be reduced translating into a performance increase of a further 2.5%.

7.2 Future Work

While this dissertation has already illustrated the used of ECMon with novel monitoring applications, the applications considered are by no means exhaustive. Future work can proceed by exploring other novel applications that ECMon can enable. Another line of

future work is to identify and design other general purpose HW support that can be used to further optimize the costs involved in monitoring parallel programs on multicores.

Ensuring SC using ECMon: With the advent of multicores, there is great demand to write correct parallel programs. One of the implicit assumptions that a programmer makes about the parallel program he writes, is *sequential consistency* (SC) [40]. However multicores often use weaker memory consistency models [7] for purposes of efficiency. Compilers introduce *memory fences* [27, 89] (or delays) to ensure a sequentially consistent execution of a parallel program running on a machine supporting a weaker memory consistency model. To identify the insertion points of fences, compilers need to compute shared memory dependency information. However, since static analysis is used to identify the shared memory dependences, fences are often introduced conservatively, which can significantly slowdown a program. To increase performance, one possible research direction is to utilize ECMon, to introduce fences dynamically only when it is necessary.

Transactional memory using ECMon: Recently there has been huge interest in the industry and academia on transactional memory (TM). In addition to providing lock free concurrency, it can also be used for performing speculative parallelization. TM, which mainly attempts to isolate different threads executing concurrently, consists of two major components: conflict detection and memory versioning. Conflict detection, whose purpose is to detect if two concurrently executing threads share a dependence, can potentially be implemented using ECMon support. Memory versioning, whose purpose is to maintain memory isolation between threads, can be potentially implemented using OS support proposed in chapter 3. Thus, it would be interesting to see if the support proposed in this dissertation, can be used to implement transactional memory efficiently.

Parallelizing software based monitoring tools: ECMon support can be used to enable software based monitoring of parallel programs running on multicores. However, it does not deal with the problem of parallelizing the instrumentation involved in software based monitoring. This is particularly relevant with manycore processors, which can have several cores that are idle. Nagarajan et al. [59] have proposed a hardware based communication queue between cores, to help in parallelizing DIFT on a multicore. However, the above work only considers sequential programs. It would be interesting to apply this technique with the above support in conjunction with ECMon, so that the above technique can also be applied for parallel programs. More generally, identifying the support needed to parallelize the instrumentation involved in software monitoring would constitute an interesting direction of future work.

Programming language support for monitoring: Current software monitoring techniques are contingent on system programmers writing correct code for the instrumentation associated with every instruction. With ECMon, there is added programming effort involved in writing the handlers associated with every cache event. It would be beneficial to convey the semantics of the monitoring task via programming language constructs. Then the actual code for accomplishing the monitoring could be potentially generated by the compiler. Thus the design of the necessary programming constructs to convey the Semitics of a monitoring task constitutes an important direction of future work.

Bibliography

[1] Itanium software developers manual. In *http://www.intel.com/design/itanium/manuals/iiasdmanual.htm*.

[2] Midnight commander. http://www.cert.org/stats.

[3] mysql bug. bugs.mysql.com/bug.php?id=110.

[4] National vulnerability database. http://nvd.nist.gov/statistics.cfm.

[5] National vulnerability database statistics. http://nvd.nist.gov/statistics.cfm.

[6] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI*, pages 26–37, 2006.

[7] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[8] F. Angiolini, L. Benini, and A. Caprara. An efficient profile-based algorithm for scratchpad memory partitioning. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(11):1660–1676, 2005.

[9] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *SPAA*, pages 99–108, 2002.

[10] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO*, pages 265–275. IEEE Computer Society, 2003.

[11] I. Buck. *Stream computing on graphics hardware*. PhD thesis, Stanford, CA, USA, 2005. Adviser-Pat Hanrahan.

[12] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. In *OSDI*, pages 31–44, 2004.

[13] H. Chen, X. Wu, L. Yuan, B. Zang, P.-C. Yew, and F. T. Chong. From speculation to security: Practical and efficient information flow tracking using speculative hardware. In *ISCA*, pages 401–412, 2008.

[14] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *ISCA*, 2008.

[15] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. *ISCC*, pages 749–754, 2006.

[16] J. Chung, M. Dalton, H. Kannan, and C. Kozyrakis. Thread-safe binary translation using transactional memory. In *HPCA*, 2008.

[17] M. H. Cintra and D. R. L. Ferraris. Design space exploration of a software speculative parallelization scheme. *IEEE Trans. Parallel Distrib. Syst.*, 16(6):562–576, 2005.

[18] M. H. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *ISCA*, pages 13–24, 2000.

[19] J. A. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA*, pages 196–206, 2007.

[20] K. D. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *POPL*, pages 49–59, 1989.

[21] J. R. Crandall, S. F. Wu, and F. T. Chong. Minos: Architectural support for protecting control data. *ACM Trans. Archit. Code Optim.*, 3(4):359–389, 2006.

[22] J. R. Crandall, S. F. Wu, and F. T. Chong. Minos: Architectural support for protecting control data. *TACO*, 3(4):359–389, 2006.

[23] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *ISCA*, pages 482–493, 2007.

[24] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII*, pages 336–346, 2006.

[25] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI*, pages 223–234, 2007.

[26] K. Fan, H. hul Park, M. Kudlur, and S. ott Mahlke. Modulo scheduling for highly customized datapaths to increase hardware reusability. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 124–133, New York, NY, USA, 2008. ACM.

[27] X. Fang, J. Lee, and S. P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *ICS*, pages 285–294, 2003.

[28] Y. S. G. Venkataramani, I. Doudalis and M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *HPCA*, 2008.

[29] J. Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[30] J. Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[31] R. Gupta. The fuzzy barrier: A mechanism for high speed synchronization of processors. In *ASPLOS*, pages 54–63, 1989.

[32] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS*, pages 58–69, 1998.

[33] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[34] M. Hind, M. G. Burke, P. R. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, 1999.

[35] D. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *ISCA*, pages 265–276, 2008.

[36] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *Computer*, 36(8):54–62, 2003.

[37] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *PLDI*, 2008.

[38] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. In *ASPLOS*, pages 233–243, 2008.

[39] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222, 2007.

[40] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[41] J. Lin, T. Chen, W.-C. Hsu, and P.-C. Yew. Speculative register promotion using advanced load address table (alat). In *CGO*, pages 125–134, 2003.

[42] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: A benchmark for evaluating bug detection tools. In *Bugs*, 2005.

[43] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[44] V. J. Marathe, W. N. S. III, and M. L. Scott. Adaptive software transactional memory. In *DISC*, pages 354–368, 2005.

[45] P. Marcuello and A. González. Clustered speculative multithreaded processors. In *ICS*, pages 365–372, 1999.

[46] J. F. Martínez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *ASPLOS*, pages 18–29, 2002.

[47] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA*, pages 69–80, 2007.

[48] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution effciently. In *ISCA*, pages 289–300, 2008.

[49] S. S. Mukherjee, J. S. Emer, and S. K. Reinhardt. The soft error problem: An architectural perspective. In *HPCA*, pages 243–247, 2005.

[50] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.

[51] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. Sigrace: signature-based data race detection. In *ISCA*, pages 337–348, 2009.

[52] V. Nagarajan and R. Gupta. Support for symmetric shadow memory in multiprocessors. In *PADTAD*, page 5, 2008.

[53] V. Nagarajan and R. Gupta. Architectural support for shadow memory in multiprocessors. In *VEE*, pages 1–10, 2009.

[54] V. Nagarajan and R. Gupta. Ecmon: Exposing cache events for monitoring. In *ISCA*, 2009.

[55] V. Nagarajan and R. Gupta. Runtime monitoring on multicores via oases. *SIGOPS Oper. Syst. Rev.*, 43(2):15–24, 2009.

[56] V. Nagarajan and R. Gupta. Speculative optimizations for parallel programs on multicores. In *LCPC*, 2009.

[57] V. Nagarajan, D. Jeffrey, and R. Gupta. Self-recovery in server programs. In *ISMM*, pages 49–58, 2009.

[58] V. Nagarajan, D. Jeffrey, R. Gupta, and N. Gupta. Ontrac: A system for efficient online tracing for debugging. In *ICSM*, pages 445–454, 2007.

[59] V. Nagarajan, H.-S. Kim, Y. Wu, and R. Gupta. Dynamic information flow tracking on multicores. 2008.

[60] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *ASPLOS-XII*, pages 229–240, 2006.

[61] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Recording application-level execution for deterministic replay debugging. *IEEE Micro*, 26(1):100–109, 2006.

[62] N. Nethercote and A. Mycroft. Redux: A dynamic dataflow tracer. *Electronic Notes in Theoretical Computer Science 89 No. 2*, 2003.

[63] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *VEE*, pages 65–74, 2007.

[64] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007.

[65] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Workshop on Parallel and Distributed Debugging*, pages 1–11, 1993.

[66] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[67] C. O'Hanlon. A conversation with john hennessy and david patterson. *ACM Queue*, 4(10):14–22, 2006.

[68] D. L. Oppenheimer, A. B. Brown, J. Beck, D. Hettena, J. Kuroda, N. Treuhaft, D. A. Patterson, and K. A. Yelick. Roc-1: Hardware support for recovery-oriented computing. *IEEE Trans. Computers*, 51(2):100–107, 2002.

[69] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society.

[70] D. A. Patterson. Recovery oriented computing: A new research agenda for a new century. In *HPCA*, page 247, 2002.

[71] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Trans. Parallel Distrib. Syst.*, 9(10):972–986, 1998.

[72] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies - a safe method to survive software failures. In *SOSP*, pages 235–248, 2005.

[73] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO 39*, pages 135–148, 2006.

[74] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO*, pages 294–305, 2001.

[75] R. Rajwar, M. Herlihy, and K. K. Lai. Virtualizing transactional memory. In *ISCA*, pages 494–505, 2005.

[76] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 114–123, New York, NY, USA, 2008. ACM.

[77] B. Randell, P. A. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Comput. Surv.*, 10(2):123–165, 1978.

[78] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA*, pages 25–36, 2000.

[79] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: Software implemented fault tolerance. In *CGO*, pages 243–254, 2005.

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

[81] M. C. Rinard. Analysis of multithreaded programs. In *SAS*, pages 1–19, 2001.

[82] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, pages 303–316, 2004.

[83] M. Ronsse and K. D. Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[84] M. Ronsse and K. D. Bosschere. Non-intrusive on-the-fly data race detection using execution replay. In *AADEBUG*, 2000.

[85] Y. Saito. Jockey: a user-space library for record-replay debugging. In *AADEBUG*, pages 69–76, 2005.

[86] A. Salcianu and M. C. Rinard. Pointer and escape analysis for multithreaded programs. In *PPOPP*, pages 12–23, 2001.

[87] J. Sampson, R. González, J.-F. Collard, N. P. Jouppi, M. Schlansker, and B. Calder. Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers. In *MICRO*, pages 235–246, 2006.

[88] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[89] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

[90] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. Heapmon: A helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM Journal of Research and Development*, 50(2-3):261–276, 2006.

[91] J. Shirako, J. M. Zhao, V. K. Nandivada, and V. Sarkar. Chunking parallel loops in the presence of synchronization. In *ICS*, pages 181–192, 2009.

[92] A. Shriraman, M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas, and M. L. Scott. An integrated hardware-software approach to flexible transactional memory. In *ISCA*, pages 104–115, 2007.

[93] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *USENIX Annual Technical Conference, General Track*, pages 149–161, 2005.

[94] M. F. Spear, A. Shriraman, H. Hossain, S. Dwarkadas, and M. L. Scott. Alert-on-update: a communication aid for shared memory multiprocessors. In *PPOPP*, pages 132–133, 2007.

[95] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *ATEC*, pages 3–3, 2004.

[96] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *ISCA*, pages 1–12, 2000.

[97] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, pages 85–96, 2004.

[98] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers (awarded best paper!). In *OSDI*, pages 1–16, 2004.

[99] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *SOSP*, pages 207–222, 2003.

[100] S. Tallam, C. Tian, R. Gupta, and X. Zhang. Enabling tracing of long-running multithreaded programs via dynamic execution reduction. In *ISSTA*, pages 207–218, 2007.

[101] S. Tallam, C. Tian, R. Gupta, and X. Zhang. Avoiding program failures through safe execution perturbations. In *COMPSAC*, pages 152–159, 2008.

[102] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[103] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *MICRO*, pages 330–341, 2008.

[104] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.

[105] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *HPCA*, pages 273–284, 2007.

[106] T. N. Vijaykumar, S. Gopal, J. E. Smith, and G. S. Sohi. Speculative versioning cache. *IEEE Trans. Parallel Distrib. Syst.*, 12(12):1305–1317, 2001.

[107] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *ISCA*, pages 87–98, 2002.

[108] C. Wang, S. Hu, H.-S. Kim, S. R. Nair, M. B. Jr., Z. Ying, and Y. Wu. Stardbt: An efficient multi-platform dynamic binary translation system. In *Asia-Pacific Computer Systems Architecture Conference*, pages 4–15, 2007.

[109] C. Wang, H.-S. Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *CGO*, pages 244–258, 2007.

[110] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.

[111] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, pages 122–133, 2003.

[112] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *SIGSOFT '06/FSE-14*, pages 81–91, 2006.

[113] P. Zhou, R. Teodorescu, and Y. Zhou. Hard: Hardware-assisted lockset-based race detection. In *HPCA*, pages 121–132, 2007.

[114] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *MICRO*, pages 85–96, 2002.