Architectural Support for Shadow Memory in Multiprocessors

Vijay Nagarajan and Rajiv Gupta

University of California, CSE Department, Riverside, CA 92521

{vijay,gupta}@cs.ucr.edu

Abstract

Runtime monitoring support serves as a foundation for the important tasks of providing security, performing debugging, and improving performance of applications. Often runtime monitoring requires the maintenance of information associated with each of the application's original memory location, which is held in corresponding *shadow memory* locations. Unfortunately, existing robust shadow memory implementations are inefficient. In this paper, we present a shadow memory implementation that is both efficient and robust. A combination of architectural support (in the form of ISA support and augmentations to the cache coherency protocol) and operating system support (in the form of coupled allocation of memory pages used by the application and associated shadow memory pages) is proposed. By coupling the coherency of shadow memory with the coherency of the main memory, we ensure that the shadow memory instructions execute atomically with their corresponding original memory instructions. Our page allocation policy enables fast translation of original addresses into corresponding shadow memory addresses; thus allowing implicit addressing of shadow memory. This approach obviates the need for page table entries for shadow pages. Our experiments show that the overheads of runtime monitoring tasks are significantly reduced in comparison to previous software implementations.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging – debugging aids, monitors

General Terms Design, Reliability, Performance, Experimentation

Keywords Shadow Memory, Coupled coherence

1. Introduction

There has been significant research on the online monitoring of running programs using software techniques for a variety of purposes. For example, *LIFT* (Qin et al. 2006) and *Taint-Check* (Newsome and Song 2005) are software tools that perform taint analysis to ensure the execution of a program is not compromised by harmful inputs; *Memcheck* (Nethercote and Seward 2007a) is a popular memory checking tool that is widely used to detect memory bugs; and *Eraser* (Savage et al. 1997) is a tool for detecting data races. A common element among these tools is that they make use of *shadow memory* (Nethercote and Seward 2007a). 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:

- **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 (Nethercote and Seward 2007a). A simple *half-and-half* scheme (Cheng et al. 2006; Qin et al. 2006) 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 (Nethercote and Seward 2007a). 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 (Nethercote and Seward 2007a) 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 is due to shadow memory accesses (Nethercote and Seward 2007a).
- **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 (Nethercote and Seward 2007b,a) 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 the multithreaded program is being run on, the now ubiquitous, multiprocessors. To overcome this problem of concurrent updates,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VEE'09, March 11–13, 2009, Washington, DC, USA.

Copyright © 2009 ACM 978-1-60558-375-4/09/03…$5.00

existing techniques serialize the execution of threads, which essentially means that a multithreaded program is forced to run on one processor. Naturally, this is not an efficient way to handle multithreaded programs.

In this paper, we present architectural support for handling robust shadow memory on multiprocessors that addresses the above challenges of *efficient shadow memory management* and *atomic updates*. Our design couples shadow memory management (i.e., its allocation, addressing, and coherence) with the management of original memory in a manner that enables the required goals to be met. The key elements of the proposed solution are as follows:

- **Instruction Set Support.** All memory instructions (loads and stores) are generated such that they explicitly refer to original memory addresses. However, instruction set support is provided to identify an OMI and its accompanying instrumentation code that includes SMIs that must execute atomically. In the case of a shadow memory instruction, during address translation, the original memory address is translated into the corresponding shadow memory address.
- **Efficient Address Translation.** To enable efficient translation of original memory addresses into shadow memory addresses we take the following approach. A page of memory belonging to the application and the corresponding shadow memory page are allocated such that they are adjacent to each other – a shadow page follows the original memory page. Thus, from the address of a original memory location, the address of corresponding shadow memory location can be efficiently computed. Furthermore, we ensure that at any point in time if an original memory page resides in main memory then the corresponding shadow memory page also resides in main memory. Thus, while page table entries are created for original memory pages, no page table entries are required for the corresponding shadow memory pages.
- **Atomic Updates.** To enforce atomicity of SMIs with their corresponding OMIs, we explore two approaches. In the *locking* approach, we associate locks with various memory regions. 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. We leverage shadow memory support to efficiently manage the numerous locks. However, this approach suffers from overhead of executing additional instructions (including the expensive atomic instructions). To avoid the execution of additional instructions associated with locking, we also propose a modified *cache coherence protocol* which *couples* the coherence of shadow values along with original values, to achieve the effect of atomicity.

Our experiments show that the execution overheads involved in ensuring atomicity and performing address translation are significant for a variety of runtime monitoring tasks (*DIFT, Eraser, Memcheck and MemProfile*). Using the architectural support proposed in this paper, viz. implicit addressing and coupled cache coherency protocol, enabled us to largely eliminate the above overheads.

This paper is organized as follows. Section 2 describes our approach in detail including the instruction set support, OS support, and the actions performed as part of the cache coherence protocol to guarantee atomicity. In section 3, we evaluate the performance of shadow memory implementation. Related work and conclusions are given in sections 4 and 5.

2. Shadow Memory: Design

In this section we describe the design of our shadow memory in detail including the OS support, ISA support and our modified cache coherence protocol. But first we look at some common monitoring tasks that use shadow memory, to infer its properties and requirements to help us in our design.

Let us consider three popular monitoring tasks that require maintenance of meta information for each memory location used by an application:

- **DIFT** (Qin et al. 2006; Newsome and Song 2005; Cheng et al. 2006) (Dynamic Information Flow Tracking) is used to track whether contents of memory locations are data dependent upon insecure inputs. With each memory location a *taint* bit is associated, which indicates whether that memory location is data dependent on an input. Consequently, the taint bit has to be manipulated for every memory instruction. For every load (store), the taint bit corresponding to the loaded (stored) memory location has to be read (updated).
- **Eraser** (Savage et al. 1997) is used to track information to enable data race detection. With every memory word we associate 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. Thus, each memory access, either by a load or a store, must be accompanied with reading and writing of both *status* and *lock-set*.
- **Memcheck** (Nethercote and Seward 2007a) 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. The *A bit* and the *V bits* have to be read on every load and updated on every store.

Table. 1 summarizes the needs of each of the above monitoring tasks. Considering these monitoring tasks we identify the following important characteristics of the monitoring tasks:

- **Association of Atomic Shadow Operations.** We observe that every OMI is associated with SMIs. Moreover, an OMI and its associated SMIs 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.
- **Symmetric vs. General Shadow Operations.** Note that the SMIs in *DIFT* are **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*.
- **Single vs. Multiple Shadow Values.** 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.

In the remainder of this section we describe our design of shadow memory which satisfies the above demands of all the above monitoring tasks. Our design consists of three components: (i) OS support for devising a robust shadow memory in which translation of original memory addresses to shadow memory addresses is quite simple and thus efficient; (ii) Instruction set support for identifying memory instructions that must be executed atomically as well as distinguishing OMI from SMIs; and (iii) Fine grained locking and the coupled cache coherence protocol that ensures atomic updates of original memory locations and corresponding shadow memory locations.

Table 1. Some Uses of Shadow Memory in Monitoring.

Application	Shadow Memory For	Load Instrumentation	Store Instrumentation
DIFT (Qin et al. 2006; Newsome and Song 2005; Cheng et al. 2006)	Taintedness bit per byte	Access Taint bit	Update Taint bit
Eraser (Savage et al. 1997)	'Lock-Set' and 'status' per word	Update 'status' and 'Lock-Set'	Update 'status' and 'Lock-Set'
Memcheck (Nethercote and Seward 2007a)	'A' bit per byte, 'V' bit per bit	Update 'A' bit, Access 'V' bits	Update 'A' bit and 'V' bits

2.1 Shadow Memory: Addressing and OS Support

The process of addressing shadow memory needs to be both robust and efficient. We simultaneously meet these goals through the following design:

- **Robustness.** We use the same virtual address to reference 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 is a corresponding shadow memory page and during page translation virtual page is translated to different physical pages – original vs shadow. This approach is robust as unlike the *half-and-half* strategy it does not require an application to reserve a significant part of its virtual address space for shadow memory.
- **Efficiency.** With OS support, every original memory page and its corresponding shadow page are allocated physical pages that are *adjacent* to each other. In fact, these pages are treated as a single entity – when the OS decides to swap out an original page into the disk, it also swaps out the associated shadow page. Similarly, both original page and its associated shadow page are swapped in together. Thus after the virtual page has been translated into a physical page by consulting the TLB, we need to simply add one to it to obtain the physical page number of the shadow memory. Thus, the translation process is highly efficient. A consequence of this scheme is that shadow memory does not require any additional TLB entries. If there is a requirement for associating multiple shadow values with each memory location, the OS allocates multiple shadow pages one for each shadow value. All shadow pages are allocated adjacent to the original memory page and by adding an offset of i to the physical page number of the original memory page, the physical page number corresponding to the ith shadow value is determined. Fig. 1 summarizes the above translation process. The determination of the *Shadow Value Count* which determines which shadow page is to be accessed will be described in the next section.

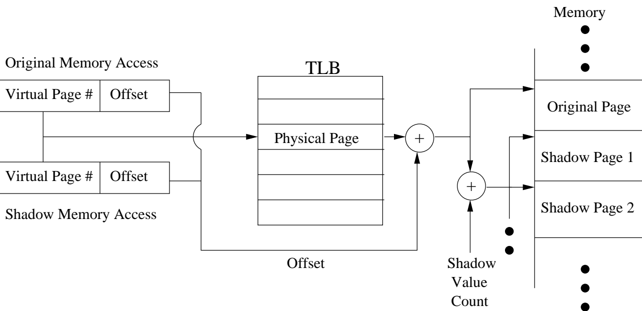

Figure 1. Address Translation

Finally, since an application may not require monitoring, we add an extra flag to the *process descriptor*, which indicates whether that particular process requires shadow memory support; and if so, the number of shadow pages to be allocated for every original page. 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.

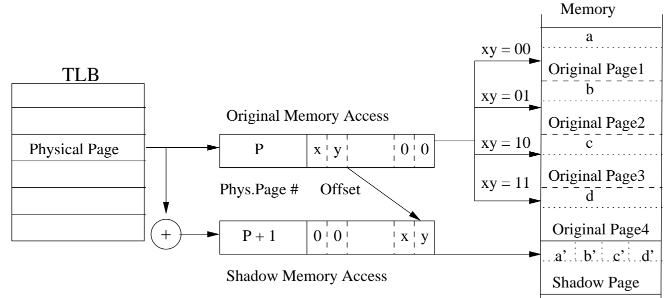

Figure 2. Handling Small-sized Shadow Values

Handling Small-sized Shadow values For some monitoring tasks, each word of original application space 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. In our scheme, we provide partial support of small-sized shadow values. More specifically we allow each original word to be associated with *half-word* and *byte* sized shadow values. Fig. 2 illustrates how we handle byte sized shadow values for every original memory word (4 bytes). The key idea is to use *superpages*; In this example we use superpages (Original pages 1 through 4 form a superpage) which are four times the size of the original page, since each shadow value is one-fourth the size of original value. Since we use superpages two of the bits (bits x and y) originally part of virtual page, now become a part of the offset. Along with an original superpage, we allocate a shadow page (of normal size). The shadow page stores the shadow values for original values present in the superpage. As illustrated, the first byte in the shadow page contains the shadow values for the first word in the *original page1* and so on. The address translation for a shadow access proceeds as follows. As before, an offset (one) is added to the fetched physical page number from the TLB. The first two bits of the offset are made $zero$ for the shadow access, since the shadow page is one-fourth the size of the superpage. To make it access the correct byte, the last two bits of the offset are set to x and y respectively.

2.2 Instruction Set Support

We saw in the previous section that the OMI and SMI have the same virtual address to enable fast translation. Hence, there is a need to provide a means for *distinguishing* regular memory instructions from SMIs. Moreover, since each OMI can be associated with several SMIs, all of which have to be executed atomically, we need a mechanism for *identify an atomic block* of instructions for the processor.

Explicitly identifying an atomic block. Two new instructions, `shadow-start` and `shadow-end`, are introduced to enclose each OMI and its following instrumentation code to form an atomic block. While the `shadow-end` instruction, does not have any operands, the `shadow-start` instruction has an n-bit operand consisting of *n shadow write bits*, one for each shadow value associated with the original memory location. If the instrumentation code contains a write to the ith shadow value, then the ith shadow write bit

must be set. The presence of these instructions guides the actions of the cache coherence protocol to ensure atomicity – these actions will be discussed in the next section.

Implicitly distinguishing shadow instructions. The OMIs are *implicitly* distinguished from SMIs by having the compiler generate instructions within an atomic block in the following stylized fashion. The first instruction in the atomic block is the OMI being instrumented and all other instructions represent the instrumentation code. All memory instructions in the instrumentation code that access the same virtual address as the OMI are recognized as shadow instructions. Furthermore, we assume that multiple shadow reads (writes) correspond to different shadow values – the first read (write) refers to the shadow value located in the first shadow page, the second read (write) refers to the shadow value located in the second shadow page and so on. We are able to do this since each shadow memory location is read and written once in an atomic block. Note that 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. Finally, it is worth noting that our design decision for SMIs to follow the OMI is only a matter of convenience for the ease of explanation; we might as well have OMI follow the SMIs if the application demands it.

Fig. 3 illustrates the above using the example of *Eraser*. Here an original load instruction is instrumented to perform a read and write each to update each of the shadow values – *status* and *lockset*. As we can see, all load and store instructions in the atomic block refer to the same virtual address *vaddr*. When the code is executed, the reference to *vaddr* in the first load instruction results in reading of a value from original memory location. During the execution of the instrumentation code, in the first and second loads (stores), *vaddr* is translated to refer to *status* and *lockset* respectively.

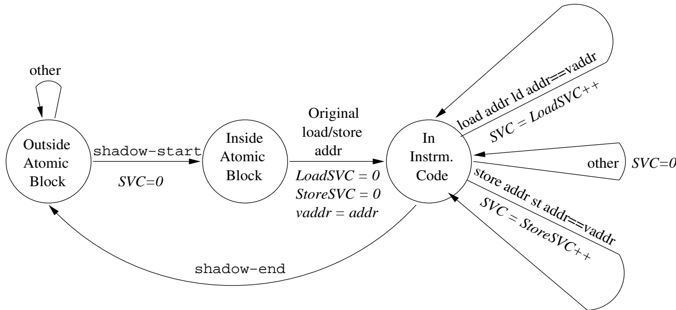

Figure 4. Generating Shadow Value Count for Address Translation.

Having described the manner in which code in atomic blocks is organized, we now show how this organization can be used to generate the *Shadow Value Count* (SVC) needed for address translation in Fig. 1. The state machine in Fig. 4 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 0. When the OMI (load or store) is encountered – the virtual address is remembered in *vaddr*; counts *LoadSVC* and *StoreSVC* are set to 0; 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.

2.3 Atomic Updates of Shadow Memory

As we have already discussed, in a multithreaded application, an OMI and its SMI(s) must be performed atomically. To see why, let

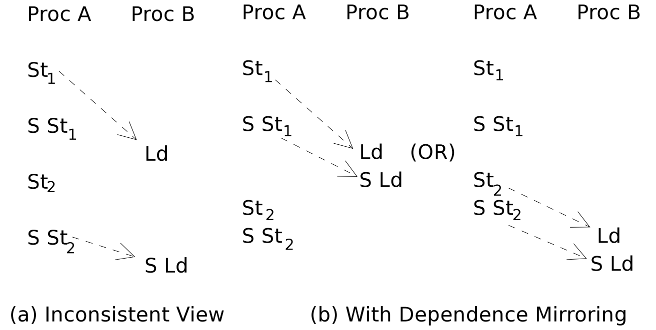

Figure 5. Atomic Updates of shadow memory.

us consider the example in Fig. 5. 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. 5a, 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. 5b. In this section we explore two solutions that guarantees atomicity without compromising on concurrency: a software solution that implements *locking* and a hardware solution using a modified *Coupled Coherence* protocol.

2.3.1 Locking

In existing software based monitoring schemes (Nethercote and Seward 2007b,a), atomicity is enforced by the serialization of threads and ensuring that a thread is not descheduled in the middle of executing an atomic block. In other words, a thread obtains a *coarse-grained lock* which enabled it to execute the OMI and its corresponding SMI(s) atomically. Instead of using one (coarse-grained) lock for implementing atomicity, we can associate a lock with different memory regions of the application memory space. Each thread that wants to execute OMI and its corresponding SMIs atomically, grabs the lock associated with the memory region and releases the lock at the end of the atomic block. Thus, this approach only serializes original and shadow instructions that operate on the same region; the size of the region is a trade-off between space and time. However, the disadvantage of using a locking scheme is that we need to potentially execute several instructions (including expensive atomic instructions), to implement locking for every memory instruction. Furthermore, using a fine-grained locking scheme requires us to manage the potentially numerous locks associated with memory regions. Fortunately, we can treat the locks as yet another set of shadow values, and can be efficiently addressed using our implicit translation scheme.

2.3.2 Coupled Coherence Protocol

To obviate the need to execute additional instructions for locking and unlocking, we propose a hardware solution that uses a modified cache coherence protocol to guarantee atomicity. In particular, to achieve atomicity, the cache coherence protocol we develop 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. If we revisit the example in Fig. 5b, dependence mirroring must guarantee that Ld and SLd see

Atomic Block Begins	`shadow-start` (11)	
Original Load Operation	$reg1 \leftarrow [vaddr]$	– *vaddr* is the virtual address
Instrumentation Code Begins	$reg2 \leftarrow [vaddr]$	– load *status* into reg2
	$reg2 \leftarrow newstatus(reg2)$	– compute new *status*
	$[vaddr] \leftarrow reg2$	– store updated *status*
	$reg2 \leftarrow [vaddr]$	– load *lockset* into reg2
	$reg2 \leftarrow newlockset(reg2)$	– compute new *lockset*
Instrumentation Code Ends	$[vaddr] \leftarrow reg2$	– store updated *lockset*
Atomic Block Ends	`shadow-end`	

Figure 3. Instrumentation Example.

either values produced by (St_1,SSt_1) or (St_2,SSt_2). In this section we will present our modified *coupled cache coherence protocol* that supports the above property. This protocol will be presented in context of a bus based shared memory multiprocessor where each processor has a local *writeback*, *write allocate* cache that are kept coherent with an *invalidation* based coherence protocol. It is worth noting that while the coupled coherence protocol will be presented in the above context, it is equally applicable in the context of other coherence protocols, including the directory based protocol (Patterson and Hennessy 1990).

(Coupled Coherence) We enforce dependence mirroring by *coupling the coherency of the shadow memory with the that of original memory*. The OMIs are made to generate coherence events for not only themselves but also for all the shadow memory instructions belonging to the same atomic block. Thus, the shadow store and load instructions *do not trigger explicit coherence actions* as their coherence is achieved by actions triggered by corresponding original store and load instructions.

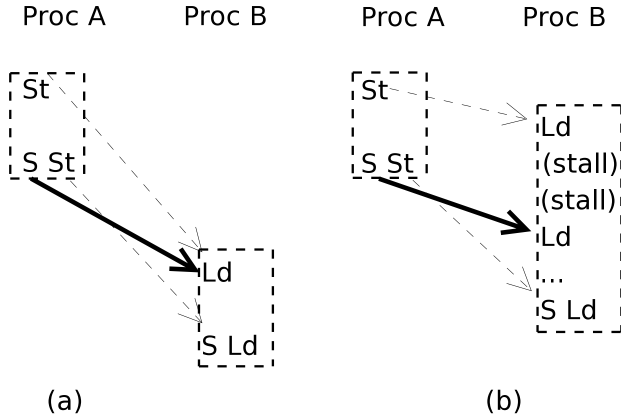

Figure 6. Dependence Mirroring for RAW.

To see how *coupled coherence* works, let us consider the example shown in Fig. 6(a). Note that the dashed box shown in the figure represents an atomic block. As we can see, there is a RAW dependency $St \rightarrow Ld$ between the OMIs. Let us assume that before the first store St is executed, each of the processors have shared copies of both original and shadow memory blocks. In our coupled coherence protocol, the original store St, will trigger invalidates for not only the original memory block, *but also the shadow memory block*. This is because the shadow store SSt is a part of the atomic block that includes the original store St. Hence, both original memory block and its corresponding shadow block in processor B will be *invalidated*. Consequently, the load Ld from processor B will result in a cache miss. In our coupled coherence scheme, this will trigger a read miss for not only the original block, *but also the shadow block*. Thus, both the original and the shadow memory blocks are brought into processor B, when the load Ld is executed. However it is important to observe that processor A should provide the shadow block only after the shadow store SSt has finished executing; otherwise it may end up providing a stale value. This ensures that the shadow block is guaranteed to be in the cache, when the shadow load $S\ Ld$ is executed and the RAW dependency between the SMIs, $S\ St \rightarrow SLd$ is enforced.

Fig. 6 (b) shows a variation of the earlier scenario, where Ld from processor B is issued before SSt finishes execution. Here the load Ld from processor B generates a *read miss* for both the original and shadow memory blocks as before. However the coherence reply for the shadow block is *deferred* until the shadow store SSt has finished executing. This results in load Ld stalling until SSt is executed in processor A, and in effect *serializes* the execution of atomic blocks from the two processors. We implement deferred coherence replies by associating a *ready* bit with every cache block; the ready bit is reset at the time of entering the atomic block and only set (meaning the block is ready) when exiting the atomic block. Thus, coupled coherence semantics ensures dependence mirroring for RAW dependencies. Dependence mirroring for WAW dependencies are satisfied in a similar fashion.

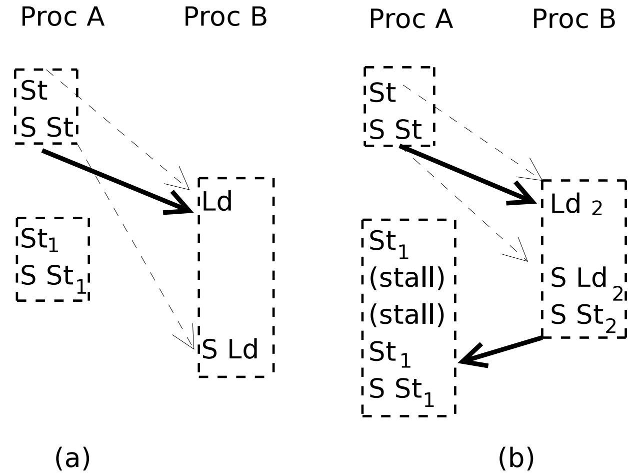

Figure 7. Dependence Mirroring for WAR.

Let us now consider the scenario from Fig. 7(a) to illustrate how dependency mirroring is achieved for WAR dependencies. Since the store St_1 from processor A executes after the load Ld from processor B, the load reads the value from the earlier store St. Thus there is a WAR dependency $Ld \rightarrow St_1$ between the OMIs. However, note that the shadow store SSt_1 from processor A executes before the shadow load SLd from processor B. If conventional coherence semantics had been followed, the shadow load SLd would have read the value written by the shadow store SSt_1. However,

with *coupled coherence*, the shadow value for SLd would have already been transferred to processor B when the load Ld was executed. Thus the shadow load SLd reads the value from the shadow store SSt and hence the WAR dependency $S\ Ld \rightarrow SSt_1$ is enforced. Finally, Fig. 7(b) illustrates a similar scenario where the original load from processor B is also associated with a shadow store, SSt_2. Consequently, the original load Ld_2 from processor B, invalidates the shadow block in processor A. Thus when St_1 executes in processor A, it results in a cache miss for the shadow block. The shadow block is later transferred from processor B when it finishes executing the atomic block. As before, St_1 is made to stall until this time.

Coherence Events for Requests from Processor

```
1.  switch (instruction)
2.    case shadow-start(w_1, w_2...w_n)
3.      for(i = 1 to n) shadow.write_i = w_i
4.    case original ld/st
5.      Access original address in cache
6.      if ld and cache-miss Place read-miss for original block
7.      else Place write-miss for original block
8.      for(i = 1 to n)
9.        Access shadow.addr_i in cache
10.       if shadow.write_i Place write-miss for ith shadow block
11.       if !shadow.write_i and cache-miss
12.         Place read-miss for ith shadow block
13.       end if
14.       set ith shadow block not ready
15.     end for
16.   case shadow ld/st
17.     Access shadow address in cache
18.     Assert(cache-hit)
19.   case shadow-end
20.     for(i = 1 to n)
21.       set ith shadow block ready
22.     end for
23. end switch
```

Coherence Events for Requests from Bus

```
24. switch (message)
25.   case read-miss (block)
26.     if block is in exclusive state
27.       set the block to shared state
28.       while block is !ready
29.       place block on bus
30.     end if
31.   case write-miss (block)
32.     set the block to invalid state
32.     if block is in exclusive state
34.       set the block to invalid state
35.       while block is !ready
36.       place block on bus
37.     end if
38. end switch
```

Figure 8. Coherence Algorithm

The coherence semantics is now summarized in Fig. 8. This algorithm first presents the coherence actions taken while handling requests from the processor. Then the coherence actions for requests from the bus are described. The algorithm makes use of the $ready$ bit, which is is an additional bit associated with every cache block, used to implement deferred coherence replies. For the following discussion let us assume that each shadow memory location is associated with n additional shadow locations. Thus, the n write bits $w_1, w_2...w_n$ associated with the `shadow-start` instruction specify whether the corresponding shadow memory location is written inside the atomic block considered (step 3). Recall that the first memory instruction following the `shadow-start` instruction refers to the OMI (which is associated with other SMIs). Steps 5 through 15 describe the coherence events for OMIs. We first access the original address in the cache and place a read or a write miss depending on whether the OMI is a load or a store (steps 5 through 7). These steps are similar to actions performed for normal memory instructions. In our modified coherence scheme, we also access the shadow addresses for each shadow value associated with the OMI. We place a write-miss for the ith shadow block, if the ith shadow value is going to be written into. Otherwise, we place a read miss for the shadow block, if it is unavailable in the cache (steps 11-13). We then reset the $ready$ bit associated with the shadow block, to indicate that the block is not yet ready to be transferred, if there is a request from the bus. Steps 16 through 18 describe the coherence events when a shadow store or load is encountered. Shadow accesses are guaranteed to be cache hits in our scheme, since we already fetched the shadow blocks. Finally, when we encounter a `shadow-end` we set the $ready$ bit associated with the shadow blocks, to indicate that the shadow blocks have been updated and are ready to be transferred (step 21).

Steps 24 through 38 describe the coherence events for requests from the bus. In our modified coherence scheme, a cache block is placed in bus as a response to a coherence request, only when the $ready$ bit for the block has been set. For example, if a cache receives a read-miss on the bus, and the block is in exclusive state, the requested block is placed in the bus, only when it is $ready$. A write-miss on the bus is handled similarly.

Finally, we briefly discuss how we can handle events which can interrupt the processor within an atomic block, i.e., within the `shadow-start` and `shadow-end` instructions. We handle thread switches, as it was handled in a prior scheme in Valgrind (Nethercote and Seward 2007b), in which the thread scheduler is made aware of SMIs and is forced to switch threads only after the SMIs for an original instruction is completed. In our scheme, we can expose the *Inside atomic block* flag to the scheduler, so that the scheduler is aware of atomic blocks and will only switch threads only when the processor is *Outside atomic block*. As far as page faults are concerned, it is worth noting that a SMI cannot cause a page fault on its own because, the shadow pages and its corresponding original page is treated as a single entity in our scheme.

3. Experimental Evaluation

In this section, we perform experimental evaluation of our shadow memory support. But before we discuss our experimental results, we briefly discuss our implementation.

3.1 Implementation

We implemented our shadow memory support including the OS and coherence algorithms in the SESC (Renau et al. 2005) simulator, targeting the MIPS architecture. The simulator is a cycle accurate multiprocessor simulator which also simulates primary functions of the OS including memory allocation and TLB handling. To implement ISA changes, we used unused opcodes of the MIPS instruction set to implement the `shadow-start` and `shadow-end` instructions. We then modified the decoder of the simulator to decode the new instructions and identify original and SMIs. We implemented our address translation support by modifying the OS page allocation algorithm to allocate additionally the shadow pages along with the original pages. We also modified the page replacement algorithm to consider the original and shadow pages as a single entity and replace them together. Finally, we implemented our coherence algorithms for an invalidate based snooping protocol for a multi-

core architecture with shared L2 cache. The architectural parameters for our implementation are presented in Table. 2. We evaluated our shadow memory support with four monitoring/profiling applications viz. DIFT(Qin et al. 2006), Memcheck(Nethercote and Seward 2007a), Eraser(Savage et al. 1997) and finally MemProfile(Angiolini et al. 2005) , a simple memory profiler that keeps a count of number of reads and writes to each memory location. This tool has potential uses in situation where we require memory locality information; for instance to allocate memory words to scratchpad memory(Angiolini et al. 2005). We very briefly describe how we performed the instrumentation for each of the monitoring tasks. For implementing DIFT, we associated a byte of shadow value for every original memory word that kept track of the taintedness of that word. We modified the system calls (that were emulated by the simulator) to initialize the taint values. Eraser is a tool for identifying data races. We implemented the first part of the algorithm which characterizes each memory word as *virgin*, *exclusive*, *shared* or *shared-modified*. We did not implement the second part of the algorithm that then uses this information to maintain the locksets. With each memory word, we associated 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. We implemented Memcheck-lite, a version of Memcheck in which the register level V-bits propagation is not implemented. We implemented a version that has been optimized for word based memory operations. For implementing MemProfile, we associated two words of data along with each original memory word, used for maintaining the number of reads and writes to that memory word.

Table 2. Architectural Parameters

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

We performed instrumentation by modifying the assembler output generated by the gcc-4.1 compiler, utilizing the `shadow-start` and `shadow-end` instructions to enable shadow memory support. One limitation of using the assembler for performing instrumentation, is that the library files are not instrumented. However, the performance results are likely to be close to our experimental results since the SPLASH-2 programs spend relatively lesser time in the libraries. It is worth noting that our shadow memory support is equally applicable to other binary translation systems (Luk et al. 2005; Nethercote and Seward 2007b). We only used the help of the assembler to perform the instrumentation, since we were not aware of publicly available dynamic translation tools that let us perform instrumentation for the MIPS architecture. We used the SPLASH-2 (Woo et al. 1995), a standard multithreaded suite (Table 3.1), benchmarks for our evaluation. We could not get the program VOLREND to compile using the compiler infrastructure that targets the simulator and hence we omitted VOLREND from our experiments.

3.2 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 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 coupled coherency protocol proposed in this paper (*coh*). In this experiment, we explore the performance of implementing various monitoring tools with different ways of achieving address translation and atomicity. The results of this experiment are presented in Fig. 9, which shows the execution time overhead of performing 4 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 coupled coherency protocol for achieving atomicity, *SM:coh*.

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. SPLASH-2 Benchmarks Description.

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 25x for performing DIFT, 45x for Memcheck, 35x for Eraser, and 27x for MemProfile. Using fine-grained locking *VAL:fgl* obviates the need for thread serialization and reduces overhead to a factor of 13x 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 9x for DIFT, 14.4x for Memcheck, 16x for Eraser, 9.5x for MemProfile. Finally using our coupled coherence protocol *SM:coh* all but eliminates the cost for performing locking and reduces the overhead to a factor of 4.3x slowdown for performing DIFT, 9.6x for Memcheck, 8.4x for Memcheck, 5x for MemProfile.

3.3 Break-Up of Overheads

To make more sense of the experimental results observed we break down the costs of performing monitoring into three categories: *address translation cost*, *instrumentation 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. 9, 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 (we used 4 processors in our simulation). Fine-grained-locking offers a slightly better alternative compared to serialization as we can infer from the results for *VAL:fgl*. However, as we can see, 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.

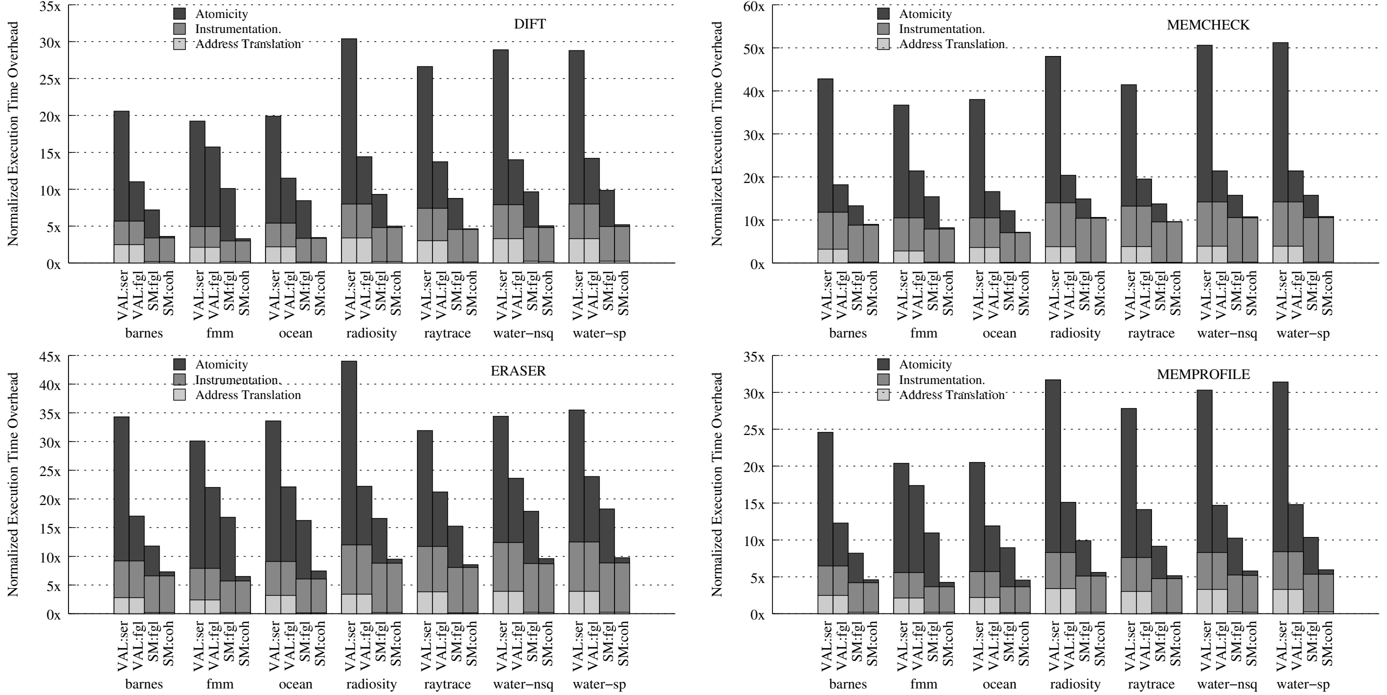

Figure 9. Monitoring Overhead with various shadow memory implementations.

Next, let us compare the overheads of *SM:fgl* with *VAL:fgl*. Since we use implicit addressing 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. Finally, as we can see in *SM:coh*, the cost of implementing atomicity is greatly reduced. This is because using our coupled coherency protocol, there is no need to execute additional instructions to perform locking. However, recall that our coupled coherence protocol automatically serializes OMI and SMI from two processors, if they race with each other. As we can see from Fig. 9, 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:coh* is almost equal to the instrumentation cost that is inherent to each monitoring task. Thus we observe that the *two forms of architectural support added in this work: implicit addressing support and cache coherence are effective in limiting the overhead of performing a variety of monitoring and profiling tasks*.

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 Memory System Performance

In this section, we evaluate the memory system performance of our shadow memory scheme in more detail. In particular, we measure the overhead introduced by our coupled coherence protocol for maintaining atomicity and the overhead introduced due to the handling of additional shadow values.

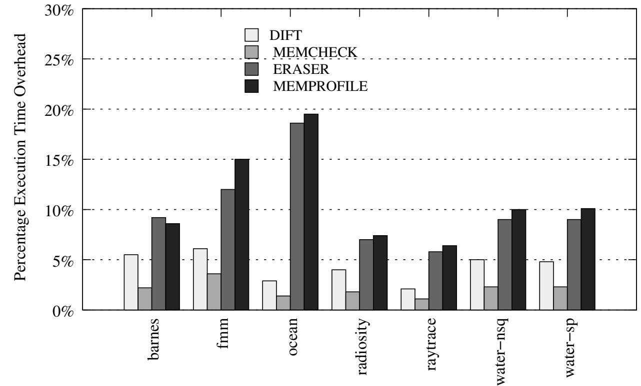

Figure 10. Percentage overhead due to coupled coherence implementation for various monitoring applications

Overhead introduced by Coupled Coherence Protocol: Recall that our coupled coherence protocol automatically serializes 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. As we can see from Fig. 10, 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. We observe that coupled coherency protocol 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 we need to additionally enforce dependence mirroring 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, let us observe the variation across different benchmarks. 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, we observe 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, we attempted to measure the effect of additional page faults. We found that we could not observe any measurable degradation in performance due to additional page faults. 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.

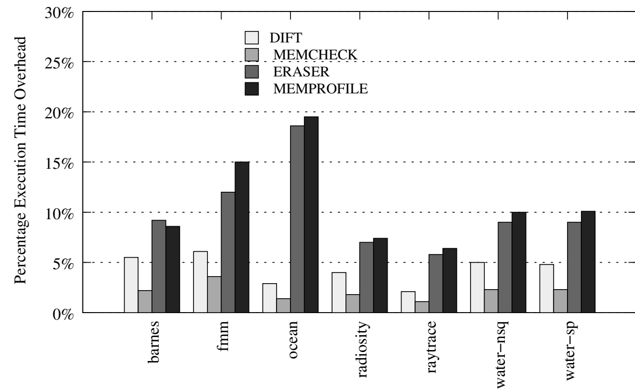

Figure 11. L1 Miss rates for various applications

Then we measured the effect of the additional shadow values on the miss rates of the caches. Fig. 11 shows the L1 miss rates for various monitoring applications. First, we observe that the 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 similar miss rate of 1.8%. Thus, we can observe only marginal increase in the cache miss rate for various monitoring applications (from 1% to 1.8%) due to additional shadow values.

4. Related Work

There has been significant research on monitoring a program as the program executes. Monitoring techniques can be broadly divided into hardware and software based approaches.

Hardware-based Monitoring Schemes: While hardware based monitoring (Dalton et al. 2007; Narayanasamy et al. 2006; Venkataramani et al. 2007, 2008; Suh et al. 2004; Xu et al. 2003) tools are fast, they require specialized hardware support in the form of wholesale changes to the processor pipeline, memory management and the caches. For example, hardware based DIFT (Dalton et al. 2007; Suh et al. 2004) requires that loads and stores in the program also load and store the respective taint values. More importantly, the hardware changes are specific to the monitoring task, which means each monitoring task requires a different set of hardware changes. However, recent work (Chen et al. 2008) proposes a flexible hardware solution that is applicable over a range of monitoring applications. The above work can be used in conjunction with the shadow memory support provided in this work, to further reduce the instrumentation cost.

Software-based monitoring schemes: On the contrary, software based monitoring schemes, use program instrumentation techniques (Luk et al. 2005; Nethercote and Seward 2007b) to instrument the original application with additional code that is able to perform the monitoring. Unfortunately, the main issue with software monitoring has been the speed. For example Dynamic taint checking (Newsome and Song 2005), which is one of the first schemes for software based monitoring causes very high overhead, in the order of 40 fold for SPEC programs. There has been several efforts (Qin et al. 2006; Cheng et al. 2006; Nethercote and Seward 2007a) to optimize the high overhead of software monitoring. In this paper, we identify shadow memory as an integral part of all software based monitoring tasks and provide ISA and OS support to efficiently support shadow memory. Thus, the support provided is able to be used efficiently in a variety software based applications. Another important limitation of software based monitoring schemes is its inefficiency in dealing with multithreaded programs. Currently, multithreaded programs have to be serialized to maintain correctness (Nethercote and Seward 2007b). This is because of the need to execute the OMIs and the SMIs atomically. In this paper, we deal with this problem without the serialization of the threads, by making small changes to the coherence protocol.

Half-and-half memory schemes: Several runtime monitoring approaches that use shadow memory (Cheng et al. 2006; Qin et al. 2006; Venkataramani et al. 2008, 2007) split virtual address space *apriori* so that the translation between original address to the shadow address can be achieved very efficiently. However, it was found by (Nethercote and Seward 2007a) that these class of approaches (known as half-and-half scheme) due its less flexible layout means that it fails for some programs in linux and is incompatible with operating systems with restrictive layouts (Nethercote and Seward 2007a). Moreover, it does not scale when we need to associate more than one shadow value per memory location. To improve robustness, Valgrind's *Memcheck* (Nethercote and Seward 2007a) implements a two-level page table in software. In this paper, we propose simple support to achieve the efficiency of the former, without sacrificing on the robustness.

TM for atomicity: There has been a recent proposal (Chung et al. 2008) to use transactional memory support to execute the SMIs and the OMIs concurrently, but is does not discuss the efficient addressing of SMIs which is also an important inefficiency in current software based shadow memory tools. *TM* support (Herlihy and Moss 1993) or *hardware atomicity* support proposed in (Neelakantam et al. 2007), if available, could also be used in conjunction with our efficient addressing scheme to enforce atomicity. However, our coupled coherence scheme, in comparison with TM, does not require support for checkpointing (Martínez et al. 2002) or conflict detection since there is no rollback or re-execution.

Other work: The idea of dependence mirroring was proposed earlier in our preliminary work (Nagarajan and Gupta 2008). However, we could handle only symmetric SMIs in our prior work. On the contrary, that limitation is eliminated in the current work which can handle *general* SMIs, which enables us to handle monitoring applications such as Memcheck and Eraser. Finally, the address translation and OS support proposed in this work is related

to support provided for handling superpages (Talluri and Hill 1994; Swanson et al. 1998). However, while the above work focuses on mainly increasing the performance and the *reach* of the TLB, we use the extra shadow pages for the purpose of monitoring.

5. Conclusion

In this paper, a combination of architectural support (in form of ISA support and augmentations to the cache coherency protocol) 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. By coupling the coherency of shadow memory with the coherency of the main memory, we ensure that the SMIs execute atomically with their corresponding OMIs. Our page allocation policy enables fast translation of original addresses into corresponding shadow memory addresses; thus allowing implicit addressing of shadow memory.

We implemented our shadow memory support in a cycle accurate multiprocessor simulator (Renau et al. 2005), which also models OS services. We evaluated our approach with four monitoring tasks DIFT, Memcheck, Eraser and MemProfile and found that our 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.

Acknowledgments

This work is supported by NSF grants CNS-0810906, CNS-0751961, CCF-0753470, and CNS-0751949 to the University of California, Riverside. We would like to thank the anonymous reviewers for providing useful comments to improve the paper.

References

Federico Angiolini, Luca Benini, and Alberto Caprara. An efficient profile-based algorithm for scratchpad memory partitioning. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(11):1660–1676, 2005.

Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *ISCA*, pages 377–388, 2008.

Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. *ISCC*, pages 749–754, 2006.

JaeWoong Chung, Michael Dalton, Hari Kannan, and Christos Kozyrakis. Thread-safe binary translation using transactional memory. In *HPCA*, 2008.

Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. In *ISCA*, pages 482–493, 2007.

Maurice Herlihy and J. Eliot. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

José F. Martínez, Jose Renau, Michael C. Huang, Milos Prvulovic, and Josep Torrellas. Cherry: checkpointed early resource recycling in out-of-order microprocessors. In *MICRO 35*, pages 3–14, 2002.

Vijay Nagarajan and Rajiv Gupta. Support for symmetric shadow memory in multiprocessors. In *PADTAD*, page 5, 2008.

Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Recording application-level execution for deterministic replay debugging. *IEEE Micro*, 26(1):100–109, 2006.

Naveen Neelakantam, Ravi Rajwar, Suresh Srinivas, Uma Srinivasan, and Craig Zilles. Hardware atomicity for reliable software speculation. In *ISCA*, pages 174–185, 2007.

Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *VEE*, pages 65–74, 2007a.

Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007b.

James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

David A. Patterson and John L. Hennessy. *Computer architecture: a quantitative approach*. 1990.

Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO 39*, pages 135–148, 2006.

Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, pages 85–96, 2004.

Mark R. Swanson, Leigh Stoller, and John B. Carter. Increasing tlb reach using superpages backed by shadow memory. In *ISCA*, pages 204–213, 1998.

Madhusudhan Talluri and Mark D. Hill. Surpassing the tlb performance of superpages with less operating system support. In *ASPLOS*, pages 171–182, 1994.

Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *HPCA*, 2008.

Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *HPCA*, pages 273–284, 2007.

Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.

Min Xu, Rastislav Bodik, and Mark D. Hill. A ”flight data recorder” for enabling full-system multiprocessor deterministic replay. In *ISCA*, pages 122–133, 2003.