Runtime Monitoring on Multicores via OASES

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 OASES: OS and Architectural Support for Efficient Shadow memory implementation for multicores that is also robust. A combination of operating system support (in the form of coupled allocation of memory pages used by the application and associated shadow memory pages) and architectural support (in the form of ISA support and exposed cache events) is proposed. Our page allocation policy enables fast translation of original addresses into corresponding shadow memory addresses; thus allowing implicit addressing of shadow memory. By exposing the cache events to the software, we ensure in software that the shadow memory instructions execute atomically with their corresponding original memory instructions. 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, Atomic Updates, Exposed cache events

1. Introduction

There has been significant research on the online monitoring of running programs using various dynamic analyses for a variety of purposes. For example, *LIFT* (16) and *Taint-Check* (15) are software tools that perform taint analysis to ensure the execution of a program is not compromised by harmful inputs; *Memcheck* (13) is a popular memory checking tool that is widely used to detect memory bugs; and *Eraser* (18) is a tool for detecting data races. A common element among these tools is that they make use of *shadow memory* (13). 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 (13). A simple *half-and-half* scheme (3; 16) 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 (13). 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 (13) 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 (13).

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 (14; 13) 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 (14). 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.

In this paper, we present OASES, a robust shadow memory implementation for multicores that addresses the above challenges of *efficient address translation* 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.

Monitoring Application	Meta Data Tracked by Shadow Memory	Code Instrumentation Required
DIFT (16; 15; 3) (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 (18) 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 (13) 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 (1) 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 1. Applications Requiring Runtime Monitoring.

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 we describe the role of monitoring in context of four popular monitoring tasks. In addition, we analyze the execution time overhead of runtime monitoring as well as degree to which various factors contribute to this overhead.

Table 1 describes four popular monitoring tasks: DIFT (15) for runtime monitoring of software attacks, Memcheck (13) a tool for runtime checking of memory errors, Eraser (18) for runtime detection of data races, and Memprofile a runtime memory profiler (1). 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 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 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*.

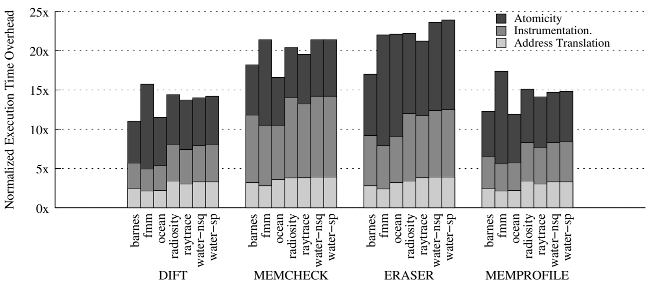

Figure 1. Overhead Imposed by Current Shadow Memory Tools.

To get an idea of the performance overhead imposed by the current shadow memory tools, we measured the overhead of performing the above monitoring tasks for the SPLASH (23) benchmarks on a 4 core processor. As shown in Fig. 1, we broke up the runtime overhead of monitoring 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. Shadow Memory Design for Multicores

We begin by providing an overview of our approach for efficiently enforcing atomicity and performing address translation. Then in subsequent sections we present our solutions in full detail. Let us first consider the problem of performing atomic updates of original memory locations and corresponding shadow memory locations.

Atomic Updates. First let us see why an OMI and its SMI(s) must be performed atomically. Consider the example shown in Fig. 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. 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. 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. (4)

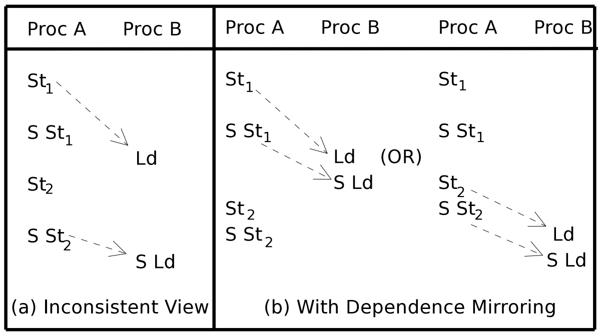

Figure 2. Atomic Updates of Shadow Memory.

Our 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 locations are illustrated by two scenarios shown in Fig. 3. Let us consider the first scenario, Fig. 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. 3b shows the reverse situation, i.e. the execution of SLd must be stalled till the shadow location has been updated.

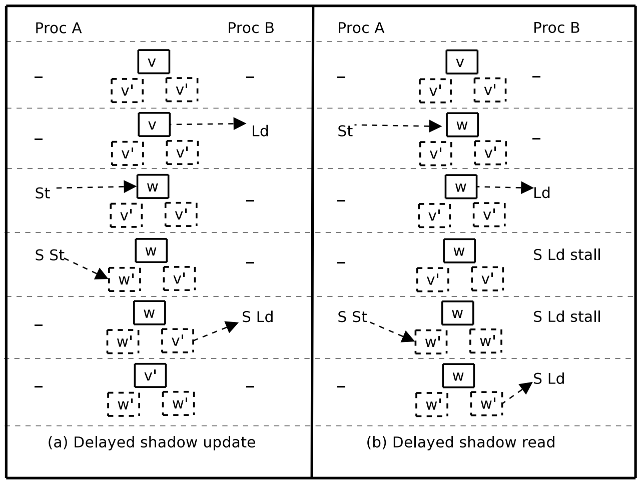

Figure 3. Timing of Shadow Value Updates.

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 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. 4 shows how cache events trigger corresponding CSC actions. To implement the CSC protocol we provide architectural support in the form of *exposing the cache events* to the software which then performs the corresponding actions of the CSC protocol. 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.

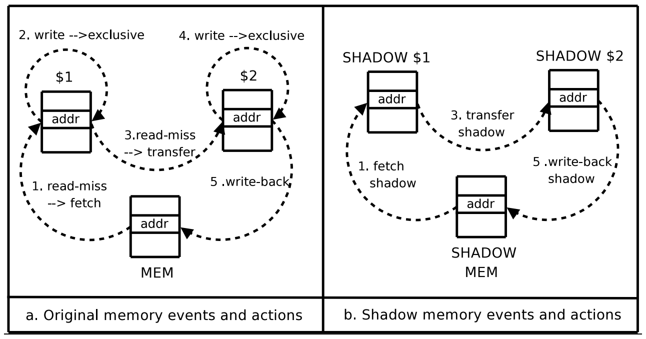

Figure 4. Coupled Shadow Coherence.

Efficient Address Translation. The process of addressing shadow memory needs to be both robust and efficient. We employ a design that meets these goals. 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 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 we take the following approach. 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, we ensure 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 pages.

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

3.1 Instruction Set Support

We need instruction set support for two purposes. First, since each OMI and all of its SMIs must execute atomically, we need a mechanism for identifying them as an *atomic block*. 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. We propose two new instructions that simultaneously meet the above requirements. 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 for various shadow values.

```
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 we assume that multiple shadow reads (writes) correspond to different shadow values. We are able to do this 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.

```
Example 1.
// init-SVC = 0
shadow-start 0x0
// Access Original memory
ld r1, addr
// Access shadow cache
ld r2, addr  // 1st shadow value
// Access shadow cache
ld r3, addr  // 2nd shadow value
shadow-end
```

```
Example 2
// init-SVC =1, pid =1
shadow-start 0x1,  0x1
// Access shadow cache
ld r1, addr  // 1st shadow value
ld r2, addr  // 2nd shadow value
shadow-end
```

```
Example 3
// init-SVC =1, pid = 3
shadow-start 0x1, 0x3
// Write to shadow memory
st r1, addr  // 1st shadow value
st r2, addr  // 2nd shadow value
shadow-end
```

Figure 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. 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.

3.2 Address Translation for Shadow Memory Accesses

Since the same virtual address is used by the OMI and the corresponding SMIs, we must implement an address translation scheme that efficiently translates the virtual address used by SMIs into appropriate physical addresses of shadow locations. To ensure that the translation can be performed efficiently, we make use of a page layout scheme that fixes the relative location of an original physical page and its corresponding shadow physical pages. 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. 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.

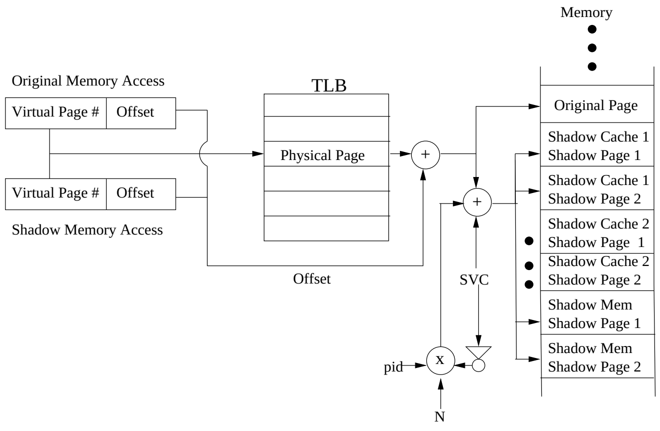

Figure 6. Address Translation.

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, we add an extra flag 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, we next show how this organization can be used to generate the *Shadow Value Count* (SVC) needed for address translation in Fig. 6. The state machine in Fig. 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.

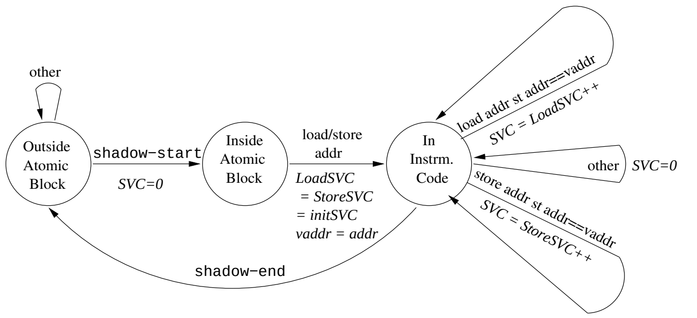

Figure 7. Generating Shadow Value Count.

Small Sized Shadow Values. In the above discussion we assumed that each memory 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 our scheme to support small-sized shadow values as discussed in (10).

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 Atomic Updates of Shadow Memory

As we have already discussed, in a multithreaded application, we need to ensure that an OMI and its corresponding SMI(s) are executed atomically. In this section, we present the CSC protocol to ensure atomicity and describe how we implement CSC in software with the help of *exposed cache events*. To implement our CSC scheme in software, the only requirement is that cache controller expose the specified events to the software. Consequently, our implementation does not rely on any particular coherence protocol used or the memory consistency model enforced by the system.

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 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. To enforce dependence mirroring, we ensure 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, we propose exposing cache events to the software. 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, we can enforce dependence mirroring in software.

Events. To implement CSC scheme in software we expose the following specific cache events:

- 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 dependencies 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). The handler is made to reside in the user space and 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 event related information to the handler as *function call parameters*: for example, the *block address* associated with the event and the remote *processor id* associated with the event, if applicable. The hardware pushes these two values (*proc* and *addr*) into the stack, before calling the handler. The programmer notifies to the hardware 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 instruction address.

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 8. State Maintained to Implement CSC.

State Maintained for Implementing CSC. Since we are implementing CSC in software, we need to maintain shadow coherence state in software as shown in Fig. 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, we maintain a flag for each processor pair (i, j) which is referred to as $ready(i, j)$. Finally when dependences are enforced through the main memory, we need to ensure 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, we maintain a count referred to as *shadow-event-cnt*, 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. 9. For this discussion, we assume macros for reading and writing into the shadow memory (steps 48 through 55) and shadow cache (steps 31 through 45).

```
1.   // Proc 'i' is about to send data reply to proc 'j'
2.   // for block address 'addr'
3.   data-reply-request (addr, j)
4.       while (ready(i,j) == true) ;
5.       ready(i,j) = true;
6.   end
7.
8.   // Proc 'j' receives  data reply from proc 'i'
9.   // for block address 'addr'
10.  data-reply (addr, i)
11.      while (ready(i,j) == false);
12.      write-shadow-cache (j, addr)= access-shadow-cache (i, addr);
13.      ready (i,j) = false;
14.  end
15.
16.  // Proc 'i' is about to write back block 'addr'
17.  write-back (addr, event-cnt)
18.      while (event-cnt != shadow-event-cnt);
19.      write-shadow-memory (addr)= access-shadow-cache (i, addr);
20.      shadow-event-cnt++;
21.  end
22.
23.  // Proc 'i' experiences a read miss for block 'addr'
24.  read-Miss (addr, event-cnt);
25.      while (event-cnt != shadow-event-cnt);
26.      write-shadow-cache (i, addr) = access-shadow-memory (addr);
27.      shadow-event-cnt++;
28.  end
29.  ---------------------------------------------------------------
30.
31.  // reads from registers and writes to shadow cache
32.  write-shadow-cache (pid, addr)
33.      shadow-start 0x1, pid   // init-SVC = 1, pid = pid
34.      st r1, addr             // 1st shadow value
35.      st r2, addr             // 2nd shadow value
36.      shadow-end
37.  end
38.
39.  // reads from shadow cache and writes to register
40.  read-shadow-cache (pid, addr)
41.      shadow-start 0x1, pid   //init-SVC = 1, pid = pid
42.      ld r1, addr             // 1st shadow value
43.      ld r2, addr             // 2nd shadow value
44.      shadow-end
45.  end
46.
47.  // reads from registers and writes to shadow memory
48.  write-shadow-memory (addr)
49.      //  pid = N+1 to access shadow memory, N = # of processors
50.      write-shadow-cache (N+1, addr)
51.
52.  // reads from shadow memory and writes to registers
53.  read-shadow-memory (addr)
54.      // pid = N+1 to access shadow memory, N = # of processors
55.      read-shadow-cache ( N+1, addr)
```

Figure 9. Handlers for Various Cache Coherence Events.

(*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, we copy the corresponding shadow block from processor j to processor i and thus implement co-transfer in software. However, we have to ensure 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. 10(b).

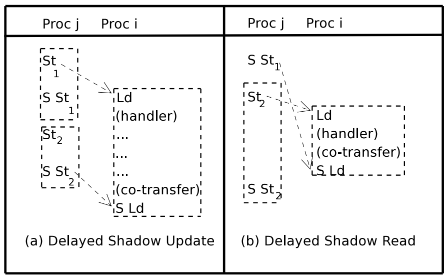

Figure 10. Co-transfer Pathological Scenarios.

Likewise, processor j may have updated the shadow block by a later SMI, by the time it is copied, as shown in Fig. 10(a). To deal with these scenarios, we make sure 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. 10(b). Within the handler, we set the *ready(i,j)* flag to 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. 11 shows the CSC actions performed in software for the delayed shadow read scenario.

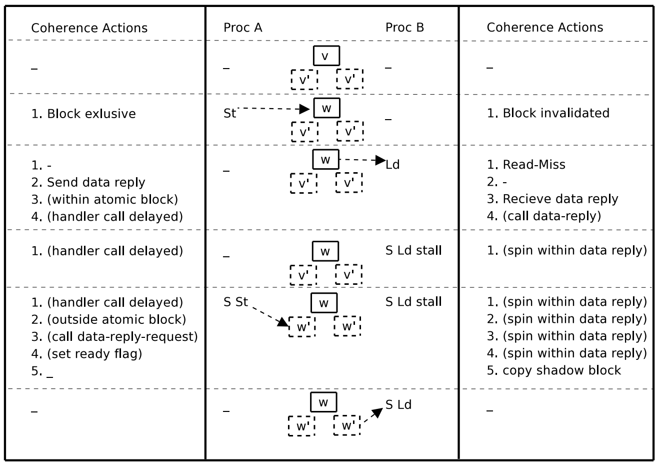

Figure 11. Cache Coherence Events and CSC Actions.

(*Co-transfer through memory.*) In the CSC scheme, we need to ensure that shadow blocks are brought in (and written back) from the memory along with original blocks. To implement this, we expose the *read miss* and *writeback* events 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 we copy the shadow cache contents of the original block 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 we copy the shadow block from the shadow memory to the shadow cache (step 26). However, we have to ensure 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. We prevent nested handler invocations 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.

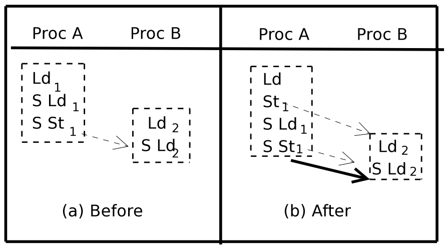

Figure 12. Transformation to Handle General SMIs.

Handling general SMIs. Let us discuss how we deal with general SMIs, where an original memory load is accompanied by both shadow loads and shadow stores. As we can see from Fig. 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, we convert the original load into a load followed by a (silent) store that writes the same loaded value back to the memory, as shown in Fig. 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.

4. 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.

4.1 Implementation

We implemented our shadow memory support including the OS support and support for exposed cache events in the SESC (17) 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, we used unused opcodes of the MIPS instruction set to implement the newly added instructions. We then modified the decoder of the simulator to decode the new instructions and implemented their semantics by adding the hardware structures to the simulator. 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 exposed cache events support for an invalidate based snooping protocol for a multicore architecture with shared L2 cache. The architectural parameters for our implementation are presented in Table. 2. We evaluated our 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 2. Architectural Parameters.

memory support with four monitoring/profiling applications viz. DIFT(16), Memcheck(13), Eraser(18) and MemProfile(1). 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.

We performed instrumentation by modifying the assembler output generated by the gcc-4.1 compiler. 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 (8; 14). 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 (23), a standard multithreaded suite (Table 4.1), 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.

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.

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 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. 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. 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* 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).

4.3 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. 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 (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.

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`

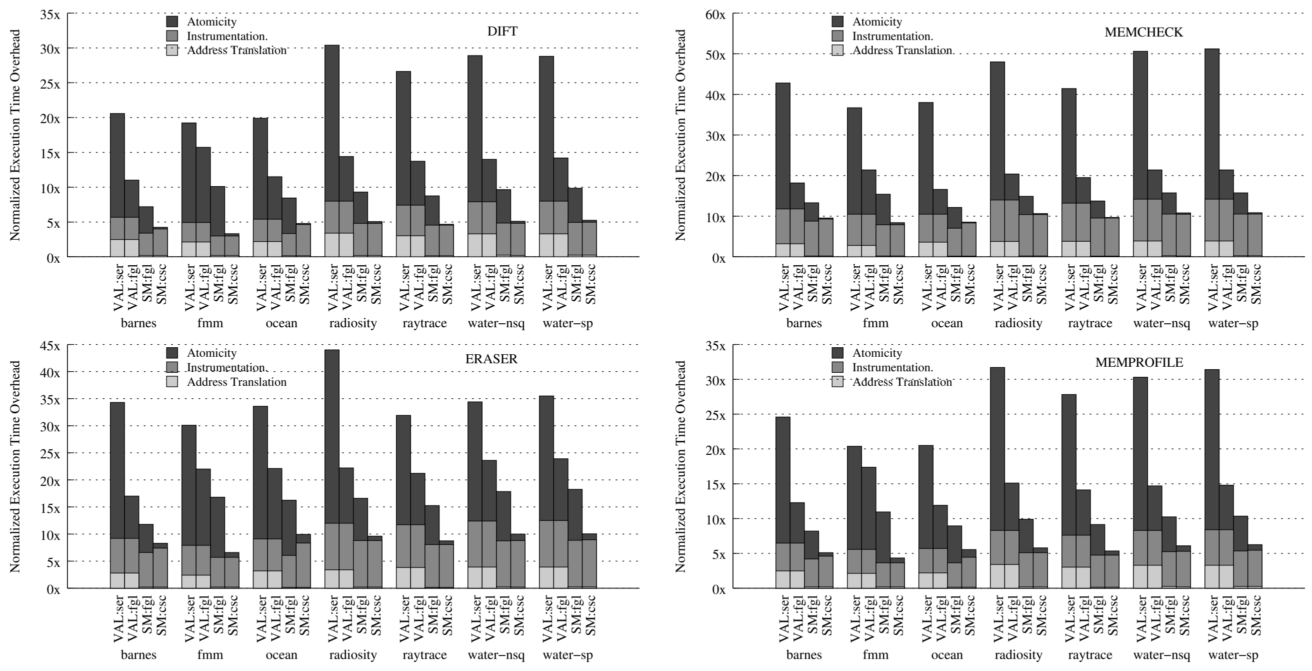

Figure 13. Monitoring Overhead with Various Shadow Memory Implementations.

instructions for identifying SMI. However, this cost is negligible compared to overall instrumentation overhead.

Finally, 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. 13, the cost for performing this limited serialization is small across all benchmarks for various monitoring tools. However, it is important to note that for enforcing atomicity, we needed to maintain per processor shadow caches; this results in slightly increased L2 miss rates. The effect of this increased miss rates causes slight increase in instrumentation costs especially for the ocean and the barnes programs. For other programs, there is no perceptible change in the memory system performance.

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 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 tasks.

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.

5. 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 (5; 11; 22; 6; 19; 24) 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 (5; 19) 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 (2) 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 (8; 14) 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 (15), 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 (16; 3; 13) to optimize the high overhead of software monitoring. In this paper, we 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 be serialized to maintain correctness (14). 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 implementing our CSC scheme in software.

Half-and-half memory schemes: Several runtime monitoring approaches that use shadow memory (3; 16; 6; 22) 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 (13) 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 (13). Moreover, it does not scale when we need to associate more than one shadow value per memory location. To improve robustness, Valgrind's *Memcheck* (13) 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 (4) 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 (7) or *hardware atomicity* support proposed in (12), if available, could also be used in conjunction with our efficient addressing scheme to enforce atomicity. However, our CSC scheme, in comparison with TM, does not require support for checkpointing (9) or conflict detection since there is no rollback or re-execution.

Other work: The address translation and OS support proposed in this work is related to support provided for handling superpages (21; 20). 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.

6. Conclusion

In this paper, a combination of architectural support (in form of ISA support and 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. By exposing cache events to the software, we were able to couple the cohere of shadow memory with the coherence of the main memory, thereby ensuring that 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 into a cycle accurate multicore simulator (17), 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 also thank the reviewers for providing useful comments to improve the paper.

References

[1] 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.

[2] 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*, pages 377–388, 2008.

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

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

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

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

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

[8] 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.

[9] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: checkpointed early resource recycling in out-of-order microprocessors. In *MICRO 35*, pages 3–14, 2002.

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

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

[12] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. In *ISCA*, pages 174–185, 2007.

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

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] M. R. Swanson, L. Stoller, and J. B. Carter. Increasing tlb reach using superpages backed by shadow memory. In *ISCA*, pages 204–213, 1998.

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

[22] 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.

[23] 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.

[24] 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.