Support for Symmetric Shadow Memory in Multiprocessors

Vijay Nagarajan
vijay@cs.ucr.edu

Rajiv Gupta
gupta@cs.ucr.edu

Univ. of California at Riverside, CSE Department, Riverside, CA 92521

ABSTRACT

Runtime monitoring support serves as a foundation for the important tasks of providing security [15, 14, 2], performing debugging [13, 11, 8], and improving performance of applications [1]. Runtime monitoring, typically, requires the maintenance of meta data associated with each of the application's original memory location, which are held in corresponding *shadow memory* locations. Each original memory instruction (OMI) is then accompanied by additional shadow memory instructions (SMIs) that manipulate the meta data associated with the memory location. Often the SMIs associated with OMIs are *symmetric*, in that, original stores (loads) are accompanied by shadow stores (loads). Unfortunately, existing shadow memory implementations need thread serialization to ensure that OMIs and SMIs are executed atomically [12]. Naturally this is not an efficient approach, especially in the now ubiquitous multiprocessors.

In this paper, we present an efficient shadow memory implementation that handles symmetric shadow instructions. 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. We also couple the allocation of application memory pages with its associated shadow pages, for enabling fast translation of original addresses into corresponding shadow memory addresses. Our experiments show that the overheads of runtime monitoring tasks are significantly reduced in comparison to previous software implementations.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Run-time environments,Memory management*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Tracing*

General Terms

Experimentation, Measurement, Reliability

PADTAD'08, July 20–21, 2008, Seattle, Washington, USA.

Keywords

symmetric shadow memory, monitoring

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* [15] and *TaintCheck* [14] are software tools that perform taint analysis to ensure that the execution of a program is not compromised by harmful inputs; *Redux* [11] and Ontrac [8] are tools that compute the dynamic dependence graphs as the program executes; and *Memcheck* [12] is a popular memory checking tool that is widely used to detect memory bugs. A common element among these tools is that they make use of *shadow memory* [12] in a *symmetric* fashion. With each memory location used by the application, a *shadow* memory location is associated to store information about that memory location. Original loads (stores) in the application are accompanied by shadow loads (stores) 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.

Although the need for shadow memory support across a variety of monitoring tasks is well recognized, supporting robust shadow memory that can be efficiently accessed and correctly manipulated remains a challenge that has not been successfully addressed. The key issue at the heart of this challenge is to efficiently ensure the atomicity of OMIs and their corresponding SMIs. Since original memory operations and shadow memory operations are performed by separate instructions, maintaining atomicity incurs an additional cost. Existing software monitoring schemes [13, 12] prevent race conditions that can lead to incorrect shadow values by ensuring that a thread switch does not occur in the middle of executing a shadow instruction. Unfortunately, the problem still exists when the multithreaded program is being run on, the now ubiquitous, multicore processors. To overcome this problem of concurrent updates, existing techniques serialize the execution of threads, which essentially

Table 1: Some Uses of Shadow Memory in Monitoring.

Application	Shadow Memory For	Load Instrumentation	Store Instrumentation
DIFT [15]	Taintedness bit per byte	Access Taint bit	Update Taint bit
DDG [11]	PC and instance per byte	Access PC and instance	Update PC and instance

means that a multithreaded program is forced to run on a single processor core. Naturally, this is not an efficient way to handle multithreaded programs.

In this paper, we present support for symmetric shadow memory operations on multiprocessors that ensures *atomic updates* of OMIs and SMIs efficiently. 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 SMIs* that must be executed atomically. In the case of a SMI, during address translation, the original memory address is translated into the corresponding shadow memory address. Our solution to enforce atomicity of SMIs with their corresponding OMIs takes advantage of the symmetric nature of shadow operations. We modify the cache coherence protocol such that the *transfer of shadow values is coupled with the transfer of original values* between processors. Thus, a shadow store does not invalidate corresponding copies in other processors; nor does it *snoop* the addresses from other processors (if a snooping protocol is followed) or provide the *data reply* in case of a miss at another processor. Instead, when there is a cache miss during an original memory access in a processor, and the cache block is sent from another processor as a data reply, the shadow memory block is also sent along with it. By coupling the transfer of original and shadow values, we achieve the effect of atomically executing OMI and its corresponding SMIs.

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.

Our experiments show that the overheads of ensuring atomicity for performing runtime monitoring tasks (*DIFT and DDG:Dynamic Dependence graph Computation*) in multithreaded programs, running on multiprocessors is nominal and thus offers a much better alternative compared to serializing threads.

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 our shadow memory implementation. In section 4 we discuss related work and finally we conclude is section 5.

2. SHADOW MEMORY: DESIGN AND IMPLEMENTATION

In this section we describe the design and implementation of our shadow memory. 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** [15, 14, 2] (Dynamic Information Flow Tracking) is used to track whether contents of memory locations are data dependent upon potentially malicious 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).
- **DDG** (Dynamic Dependence graph) *Redux* [11] and *Ontrac* [8] are tools to compute the dynamic dependence graphs as programs execute. To enable this, two values are associated with each memory location, the PC (program counter/instruction address) and the execution instance of the recent most instruction that wrote to it. This information enables us to compute the dependence edge, when the value is later loaded. Thus, each store should be accompanied by shadow stores that store the instruction address and the instance; while a load is accompanied by shadow loads that load the shadow values.

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

- **Atomic Shadow Memory Operations.** We observe that every OMI is associated with SMIs. Moreover, an OMI and its associated SMI 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.
- **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 value, for every memory location, *DDG* associates two values per memory location. Thus, in general, capability of associating **multiple shadow values** for every memory location is needed.
- **Symmetric SMIs** We observe that the SMIs are **symmetric**, i.e. for every original *load* there is an associated *shadow load* and for every original *store*, there is

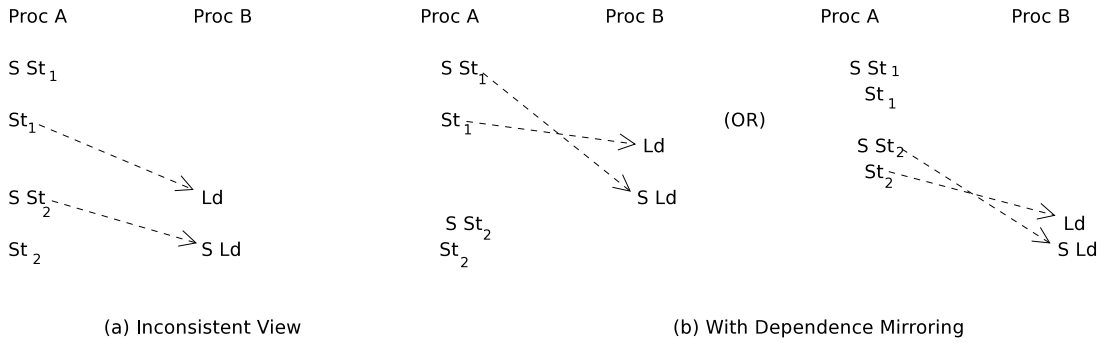

Figure 1: Requirements for Symmetric Shadow Operations.

a *shadow store*. We shall later take advantage of this symmetric property to help enforce atomicity.

2.1 Atomic Updates of Shadow Memory

As we have already discussed, in a multithreaded application executing in a multiprocessor, an OMI and its corresponding SMI(s) must be performed atomically. But first, 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. 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.

Having described how we identify an atomic block and distinguish SMIs from OMI, we now describe how we ensure atomicity. In existing software based monitoring schemes [15, 13, 12], this issue of atomicity is addressed by executing a multithreaded application on a single processor and ensuring that a thread is not descheduled in the middle of executing an atomic block. The solution we propose allows multithreaded applications to be run on multiple processors as it uses a modified cache coherence protocol to guarantee atomicity by ensuring that a consistent view of corresponding original and shadow memory locations is maintained. In particular, to achieve atomicity, the cache coherence protocol we develop ensures the following property:

Dependence Mirroring. Dependences exercised among SMIs **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 upon M_1 during an execution, then SM_2 must be similarly dependent upon SM_1.

In this section we will present our modified 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 *write-back*, *write allocate* cache which are kept coherent with an *invalidation* based *snooping* coherence protocol.

Let us consider the example in Fig. 1. 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. 1a, 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. Dependence mirroring must guarantee that Ld and SLd see either values produced by (St_1,SSt_1) or (St_2,SSt_2) as shown in Fig. 1b.

(Coupled Coherence) We enforce dependence mirroring by *coupling the coherency of the shadow memory with the that of original memory.* The key elements of this coupled policy are as follows:

- (**Co-transfer**) Whenever there is a read/write miss generated by a processor (A), and another processor (B) has the above block in an exclusive state, the latter transfers the block to the former. Here, we ensure that such block transfers between processors carries the original data block and *piggybacked* to it the shadow cache block(s).

- (**Co-existence**) An original block and its corresponding shadow block are brought into the cache together from the memory. Similarly they are also replaced together, when they are written back. (*Co-transfer* guarantees that they are brought *together* via coherence protocol). Thus, the original blocks and its corresponding shadow blocks *co-exist* in the cache and the memory.

- (**No Explicit Shadow Coherence Messages**) Finally, shadow store and load instructions *do not trigger explicit coherence actions* as their coherence is achieved by actions triggered by their corresponding original

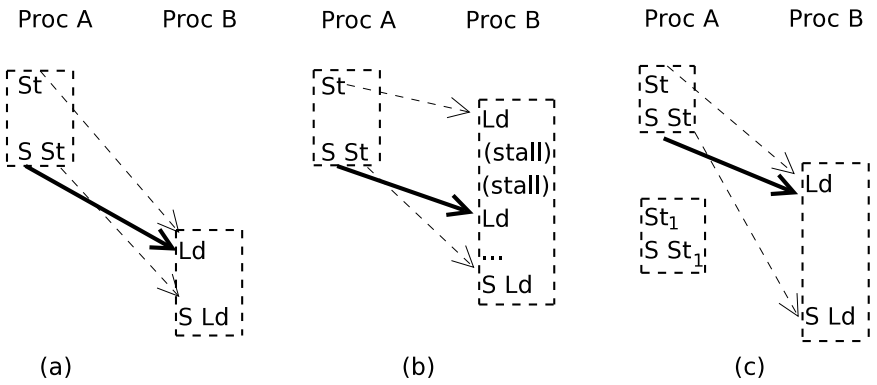

Figure 2: Dependence Mirroring: (a) RAW and WAW dependencies (b) Deferred co-transfer (c) WAR dependency

store and load instructions. Thus, shadow loads do not cause misses, since the shadow blocks must have already brought into the cache by the original load. Similarly, shadow stores do not generate invalidate messages.

Given the above modifications to the cache coherence protocol, we next show how these modifications ensure *dependence mirroring* for RAW (read after write), WAW (write after write), and WAR (write after read) dependences.

RAW and WAW Dependency:.

As shown in Fig. 2a, there is a RAW dependency $St \rightarrow Ld$ between load Ld from processor B and store St from processor A. There are two ways in which this dependency may be exercised: through cache coherence or via shared memory. Under the first scenario, 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. 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.

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. Fig. 2 (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.

Under the second scenario, the original RAW dependency is satisfied through the memory. Accordingly, some time after the blocks (original and shadow) had been updated in processor A, they will have been written back and are eventually read again, when the load Ld is encountered in processor B. Through *co-existence*, we ensure that when either of the blocks (shadow or original) is written back, the other is also simultaneously written. Similarly, when the load instruction Ld causes a read miss in processor B, both the original and shadow blocks are brought into the local cache, and this ensures dependency mirroring.

Thus, coupled coherence semantics ensures dependence mirroring for RAW dependencies. Dependence mirroring for WAW dependencies are satisfied in a similar fashion.

WAR Dependency:.

Let us now consider the scenario from Fig. 2(c) 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.

Symmetric vs General SMIs.

Our coupled coherence scheme ensures atomicity for only

symmetric SMIs. Handling *general* SMIs imposes the following complexity. Recall that in general an OMI can be accompanied by both shadow reads and writes. Thus two original load operations from two processors can be potentially accompanied by shadow reads and updates. Unfortunately, the cache coherence protocol does not expose the ordering between the two reads, for us to piggyback on the coherence and transfer the shadow values. In other words, the inability of the coherence protocol to expose RAR ordering, proves to be a stumbling block for handling general SMIs.

2.1.1 Implementation of Coupled Coherence

In this section, we discuss the implementation of the our coupled coherence protocol that was discussed. Recall that the modified coherence should now implement co-existence co-transfer and withdraw coherence events for shadow instructions, all of which are necessary to ensure dependency mirroring.

Coherence Events for Requests from Processor

```
1.  switch (instruction)
    // Let us assume that there is a cache miss
2.      case original ld
            // Ensures co-transfer, co-existence
3.          Place read-miss for original block, shadow block(s)
4.          Write-back dirty blocks
5.      case original st
            // Ensures co-transfer, co-existence
6.          Place write-miss for original block, shadow block(s)
            // Ensures deferred co-transfer
7.          Set shadow block(s) as not ready
8.          Write-back dirty blocks
9.      case shadow ld
            // A cache miss cannot happen
10.         Assert(0)
11.     case shadow st
            // No Coherence events for shadow instructions
            // Set ready bit to enable deferred co-transfer
12.         Set shadow block(s) as ready
13. end switch
```

Figure 3: Coherence Algorithm

The key idea of the algorithm, which is illustrated in Fig. 3, is to treat *OMIs* and *SMIs* differently and accordingly respond by placing appropriate coherence events on the bus. 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. Let us consider the coherence events for a cache miss on a original load instruction (step 2). Co-existence is enforced by placing a read miss for not only the original cache block; but also the shadow blocks (step 3). Moreover, by placing a read miss for the original and the shadow blocks we also ensure co-transfer. This is because, if another processor has exclusive access of the original block (and the shadow blocks), it will respond with both the original block and the shadow blocks. Since we bring in both the original and shadow block(s) for an original memory load, a shadow memory load which is generated after an original memory load cannot incur a cache miss (step 9). Now let us consider the coherence events for a cache miss on an original store. Like before, a write miss is placed for both the original and the shadow blocks to ensure co-transfer and co-existence (step 6). Additionally the shadow block(s) are made *not ready* to implement deferred co-transfer (step 7). They are made *ready* only after the completion of the shadow store (step 12). When the cache receives a request for a shadow block from the bus, the request is not serviced until it is in *ready* state.

2.2 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 half 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 i^{th} shadow value is determined. Fig. 4 summarizes the above translation process. The determination of the *Shadow Value Count* (SVC) which determines which shadow page is very simple. For every OMI, the SVC is 0 so that original pages are addressed. For every SMI encountered within an atomic block, SVC is incremented so that the correct shadow value in the appropriate shadow page is addressed.

Finally, since an application may not always require monitoring, we add an extra flag to the *process descriptor*, which indicates whether that particular process requires shadow memory support. If required, the OS sets this flag at the time of process creation. 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.

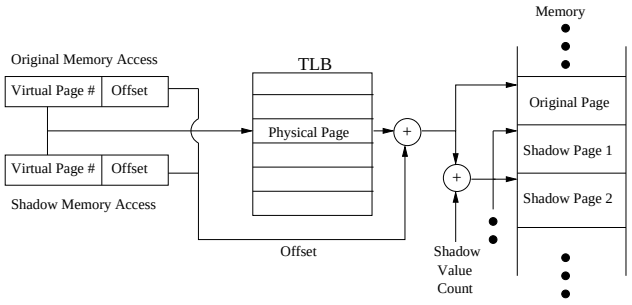

Figure 4: Address Translation

2.3 An example

Fig. 5 illustrates how the instrumented code looks like in an actual monitoring application using the example of *DDG*. Here, an original load instruction is instrumented to perform two shadow instructions that load the *PC* and the *instance* of the instruction that wrote to it the last. The `shadow` instruction indicates that the original load is instrumented with shadow loads that are to follow. The first of the shadow loads reads the value of *PC* into register reg_2. The store instruction that follows it, is not a shadow instruction, since its virtual address is different from that of the original load. The purpose of this instruction is to save the value of the PC that has been loaded. This is followed by another shadow load, which reads the second shadow value, the *instance*. Recall that during the execution of the instructions, the virtual addresses for the shadow instructions, are translated such that they access the shadow locations and not the original locations.

3. EXPERIMENTAL EVALUATION

In this section, we evaluate our shadow memory support. We conducted experiments with several goals in mind. First and foremost, we wanted to study the performance efficiency of our proposed shadow memory support. Secondly, we wanted to measure the additional overhead caused by our modified coherence algorithm. 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 [16] 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 an unused opcode of the MIPS instruction set to implement the `shadow` instruction. We then modified the decoder of the simulator to decode the new instruction 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, modeling the stall time for extra traffic on the bus. The architectural parameters for our implementation are presented in Table. 2.

We evaluated our shadow memory support with two monitoring applications DIFT[15] and DDG[11, 8]. Recall that DIFT performs dynamic information flow tracking as the program executes, to ensure that the application is not compromised. DDG is another monitoring application that computes the dynamic dependence graph as the program executes. We used the SPLASH-2 [18], a standard multithreaded suite benchmarks for our evaluation.

Table 2: Architectural Parameters

Processor	4 processor, out of order
L1 Cache	32 KB 4 way 1 cycle latency
L2 Cache	512 KB 8 way 10 cycle latency
Memory	4 GB, 200 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 [6, 13]. 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.

3.2 Efficiency of Proposed Shadow Memory Support

In this experiment, we wanted to study the performance efficiency of our shadow memory support *SM*, in comparison with the state-of-the-art baseline shadow memory tool *VAL:serial* [13]. Recall that the latter requires the serialization of threads for ensuring atomicity. On the contrary, in our scheme (*SM*), atomicity is guaranteed due to our coupled coherence implementation and serialization is not necessary. An important component of our work is the ISA and the addressing support proposed owing to which the address translation in our scheme is a lot cheaper. To evaluate the address translation efficiency of our scheme, we use another version of Valgrind's shadow memory support *VAL:lb* (lb = lower bound) – one without thread serialization. However it is important to note that *VAL:lb* is not a real shadow memory implementation – the meta data values maintained by *VAL:lb* will most likely be corrupted since atomicity is not satisfied. It is merely a version we use in our experiments to evaluate the address translation efficiency of *SM*.

The results of this experiment are presented in Fig. 6, which shows execution time overheads for the various shadow memory schemes normalized to the uninstrumented execution time. Let us first concentrate on the results for *VAL:lb* (the second bar). As we can see from Fig. 6 the performance overhead of performing DIFT with Valgrind's shadow memory (non-serialized) support causes a 12 fold slowdown on an average. This is in line with other DIFT implementation that cause similar slowdowns. However, a serialized (and correct) implementation degrades the performance greatly and causes a 41 fold slowdown on an average. This is

Instruction Type	Assembly Instruction	Comment
Shadow Instruction	`shadow-start`	Instrumentation for Load
Original Load	**ld** reg_1, vaddr	*vaddr* is the virtual address
Shadow Load	**ld** reg_2, vaddr	Load PC into reg_2
	st reg_2, $addr_1$	Not a shadow access, since addr $\neq$ vaddr
Shadow Load	**ld** reg_2, vaddr	Load *instance* into reg_2
Shadow Instruction	`shadow-end`	

Figure 5: Instrumentation Example.

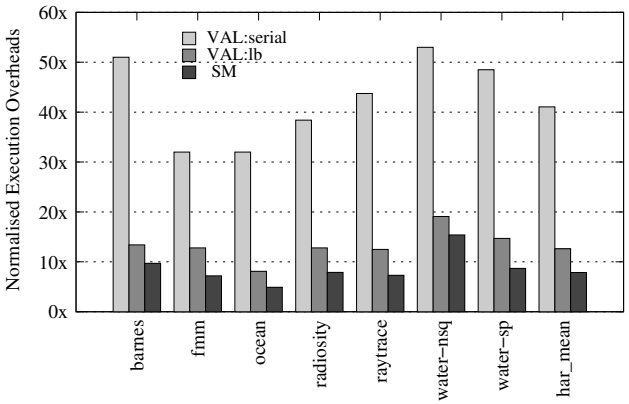

Figure 6: Overhead of performing DIFT with various shadow memory implementations

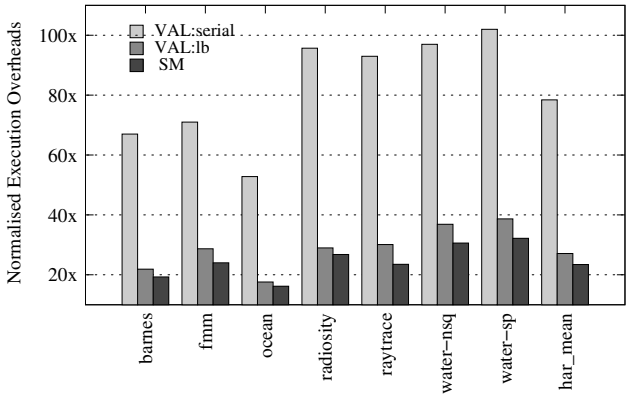

Figure 7: Overhead of computing DDG with various shadow memory implementations

also not surprising, considering the fact that the SPLASH-2 programs are highly parallel and a serialized version is expected to perform poorly compared to the parallel version. We observe that using our shadow memory support *SM* reduces this instrumentation overhead consistently across all the benchmarks. On an average, performing DIFT using *SM* causes only a 7 fold execution time overhead as opposed to the 41 fold slowdown of *VAL:serial*. More importantly, we also observe that *SM* performs significantly better compared to even *VAL:lb*, almost halving the performance overhead. The main reason for the reduction in the overhead is due to the fact that we did not need to execute additional instructions to perform the address translation.

Fig. 7 shows the corresponding performance results for the DDG application, which computes the dynamic dependence graph as the program executes. As we can see *VAL:lb* slows down the program by a factor of 27. This is much higher than DIFT because of the heavy weight instrumentation involved in computing and outputting the dependence information. Not surprisingly, the serialized version *VAL:serial* increases the slowdown to a factor of 68. The corresponding execution slowdown for *SM* is a factor of 23 slowdown. As in the case of DIFT, we observe that *SM* performs better than even *VAL:lb*, although the reduction in the execution time is not as dramatic as in DIFT because of the heavy weight instrumentation involved in outputting the dependence graph, which is common to both the versions.

From the above experiment, it is clear that our shadow memory support enables heavy weight instrumentation like DDG and DIFT in an efficient fashion without the need for serialization. The ISA and the addressing support proposed is also able to reduce the address translation overhead significantly.

3.3 Effect of Coupled Coherence

In this section, we study the overhead of providing atomicity in *SM*. Recall, that we had to modify the cache coherence algorithm to provide dependency mirroring, which was in turn used to guarantee atomicity of original and shadow memory accesses. In this experiment, we wanted to study the additional overhead (if any) of implementing our modified cache coherence protocol. To measure this overhead, we considered two versions of our shadow memory implementation: one version that implements the coupled coherence semantics and hence guarantees atomicity; and another version, the baseline, which just assumes the old coherence protocol. We performed this experiment for different applications: DIFT, DDG and a hypothetical application which manipulates three shadow values for every memory instruction. As we can see from Fig. 8, the performance overhead incurred for implementing coupled coherence is less than 0.6% on an average for DIFT and DDG. The overhead for the hypothetical application that uses three shadow values is also less than 1% on an average. The reason behind the low overhead can be explained by the fact that the total amount of traffic in the memory bus is equal for both coherence implementations. The coupled coherence implementation sees more traffic bursts in the memory bus compared to the baseline; This manifests itself as slight increases in the overhead, as the number of shadow values associated with a memory location is increased. From this experiment, it is clear that atomicity in *SM* comes almost for free, in comparison to the

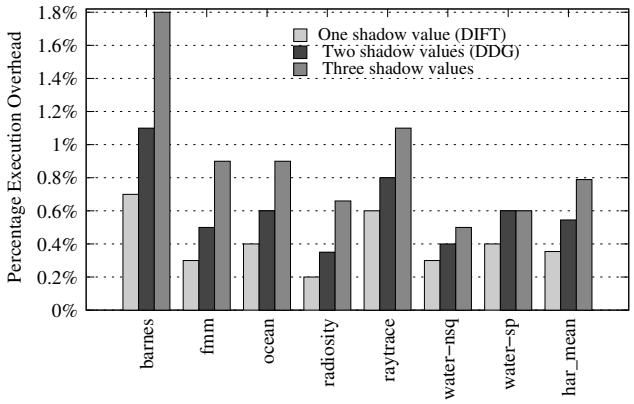

Figure 8: Percentage overhead due to coupled coherence implementation

thread serialization required in the state-of-the-art shadow memory tool.

3.4 Sensitivity Towards number of Shadow Values

In this experiment, we wanted to study the effect of the number of shadow values (associated with each memory location) towards the execution time overhead, specifically on the address translation overhead. For this experiment, we used three hypothetical applications each of which manipulates one, two and three shadow values respectively for every memory instruction; the hypothetical applications were used just to ensure that the instrumentation code retains the same flavor as the number of shadow values are varied. For each application, we had two versions *SM*, that uses our shadow memory support and *VAL:lb*, that uses Valgrind's shadow memory support without serialization.

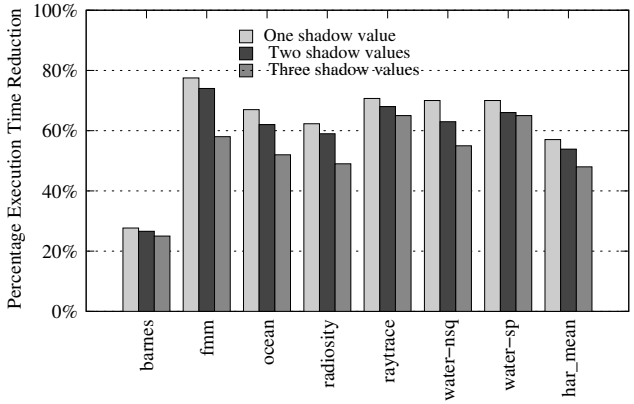

Figure 9: Sensitivity of our shadow memory support with respect to the number of shadow values.

Fig. 9 shows the percentage reduction in execution time overhead, caused by using our shadow memory support, as the number of shadow memory values associated with a memory location is varied. As we can see, the percentage reduction almost stays a constant (50% to 55% on an average) as the number of shadow values are increased, confirming the scalability of our approach with respect to the number of shadow values associated with each memory location. Thus this experiment shows that *SM* reduces the address translation overhead significantly compared to *VAL:lb* regardless of the number of shadow values associated with a memory location.

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. While hardware based monitoring [4, 17, 19, 9] 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 [4, 17] 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 application, which means each monitoring application requires a different set of hardware changes. On the contrary, software based monitoring schemes, use program instrumentation techniques [6, 13] 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 [14], 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 [15, 2, 12] to optimize the high overhead of software monitoring. In this paper, we identify shadow memory as an integral part of all software based monitoring applications 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 [13]. 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. There has been a very recent proposal [3] to use transactional memory support to execute the SMIs and the OMIs concurrently. However, as we already discussed, TM [5] or hardware atomicity proposed in [10] require support for checkpointing the state of the processor [7] to enable rollback and re-execution. On the contrary, there is no need for rollback or re-execution in our scheme. We observe that atomicity is only needed for OMI and its corresponding SMIs, and take advantage of the inherent structure to come up with a much simpler scheme.

There has been a recent proposal [3] 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 [5] or *hardware atomicity* support proposed in [10], 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 [7] or conflict detection since there is no rollback or re-execution.

5. CONCLUSIONS

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 for symmetric SMIs 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 into a cycle accurate multiprocessor simulator [16], which also models OS services. We evaluated our approach with two monitoring applications DIFT [15] and DDG [11, 8] and found that our shadow memory implementation, requiring no thread serialization, was significantly more efficient compared to the state-of-the-art software tool.

For future work, we plan to extend our shadow memory implementation to handle not only handle *symmetric* SMIs, but also handle *general* SMIs.

Acknowledgements. We would like to thank the anonymous reviewers for their valuable feedback. This research is supported by NSF grants CNS-0751961, CNS-0751949, CNS-0810906, and CCF-0753470.

6. REFERENCES

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

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

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

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

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

[6] 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.

[7] 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.

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

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

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

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

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

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.