ECMon: Exposing Cache Events for Monitoring

Vijay Nagarajan Rajiv Gupta
University of California at Riverside, CSE Department, Riverside, CA 92521
{vijay,gupta}@cs.ucr.edu

ABSTRACT

The advent of multicores has introduced new challenges for programmers to provide increased performance and software reliability. There has been significant interest in techniques that use software speculation to better utilize the computational power of multicores. At the same time, several recent proposals for ensuring software reliability are not applicable in a multicore setting due to their inability to handle interprocessor shared memory dependences (ISMDs). The demands for performing speculation and ensuring software reliability in a multicore setting, although seemingly different, share a common requirement: the need for monitoring program execution and collecting interprocessor dependence information at low overhead. For example, an important component of speculation is the efficient detection of miss-speculation which in turn requires dependence information. Likewise, tasks that help ensure software reliability on multicores, including *recording for replay*, require ISMD information.

In this paper, we propose *ECMon*: support for exposing cache events to the software. This enables the programmer to catch these events and react to them; in effect, efficiently exposing the ISMDs to the programmer. In the context of speculation, we show how *ECMon* optimizes the detection of miss-speculation; we use this simple support to speculate past active barriers and achieve a speedup of 12% for the set of parallel programs considered. As an application of ensuring software reliability, we show how *ECMon* can be used to record shared memory dependences on multicores using no specialized hardware support at only 2.8 fold execution time overhead.

Categories and Subject Descriptors

B.2.2 [**Memory Structures**]: Design Styles – cache memories; D.2.5 [**Software Engineering**]: Testing and Debugging – debugging aids, monitors; D.3.4 [**Programming Languages**]: Processors – compilers,optimization

General Terms

Design, Reliability, Performance, Experimentation

ISCA'09, June 20–24, 2009, Austin, Texas, USA.

Keywords

cache events, speculation past barriers, recording for replay

1. INTRODUCTION

The advent of multicores has introduced new challenges for programmers to provide increased performance. There has been significant interest on software speculation techniques [7, 37] which uncover parallelism by speculating past dependences. At the same time, there has been significant research on ensuring software reliability of programs through online monitoring of running programs for a variety of purposes including debugging [35, 23] and security [25, 26]. The demands for performing speculation and ensuring software reliability on multicores, although seemingly different, share a common requirement: the need for monitoring program execution and collecting *interprocessor shared memory dependence* (ISMD) information *correctly* and *efficiently.*

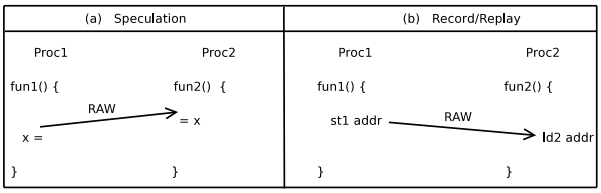


Figure 1: ISMDs in (a) speculation (b) record-replay.

- Fig. 1(a) illustrates the basic steps of software speculation, where *fun2* is executed speculatively in parallel with *fun1*, although in sequential execution *fun2* follows *fun1*. Let us assume that *fun2* is made to run in a protected memory space so that the anti and output dependences between them vanish [7, 37]. The speculation then succeeds if no true dependences are exercised between *fun1* and *fun2*; in other words, the speculation succeeds if *fun2* does not read any memory location that was written by *fun1*.
- Record-replay based debugging [12, 17, 22, 35, 40] is a technique that helps ensure software reliability by *recording* program execution, so as to enable *replay* to help in *debugging*. In a multicore setting, ISMDs need to be recorded in addition to recording other non-deterministic events. Let us consider the example shown in Fig. 1(b) which shows the parallel execution of two functions (no speculation involved) *fun1* and *fun2*. As we can see, *fun2* reads the value written by *fun1* and this RAW dependency must be recorded to ensure that *ld2* gets the correct value (from *st1*) during replay.

Monitoring Application	Purpose of tracking ISMDs
DIFT [25, 26] (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.	Whenever there is an ISMD between two memory locations, ensure that there is a corresponding dependence between *taint* bits associated with the memory locations.
Eraser [32] 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.	Whenever there is an ISMD between two memory locations, ensure that there is a corresponding dependence between *status* and *lockset* associated with the memory locations.
Memcheck [23] 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.	Whenever there is an ISMD between two memory locations, ensure that there is a corresponding dependence between *A bit* and *V bits* associated with the memory locations.

Table 1: Other Monitoring applications requiring ISMD information.

- In addition to speculation and record-replay based debugging, the detection of ISMDs is crucial to a variety of other monitoring applications, as illustrated in Table 1. Each of the monitoring applications listed in Table 1 associates *meta data* with original memory locations in corresponding *shadow memory* locations [23]. When run on multicores, races between original memory accesses and meta data accesses can result in inconsistent meta data values [3, 19, 20, 23]. Knowledge of ISMDs between original memory operations enables us to enforce the corresponding meta data dependences and thus maintain consistent meta data [20].

Current software based monitoring tools are thwarted by these ISMDs, which makes them inapplicable for multicores [25, 26, 35]. On the contrary, hardware based monitoring tools [5, 12, 17, 21, 22, 36, 40], which are applicable, use *specialized* hardware support that is geared toward the particular monitoring application. Thus this paper is motivated by the *lack of general purpose* hardware support for detecting these ISMDs. In *ECMon*, we expose cache events to the software; in effect, efficiently exposing the ISMDs to the programmer. Whenever the cache controller of a processor receives a cache event (eg. invalidate, data value reply), it interrupts the processor, and calls a predefined *handler* function. The handler function is programmable and is defined based upon the monitoring application. In this work, we show how the handlers can be programmed for two different monitoring applications: speculation past barrier synchronizations for performance and record-replay based debugging; we use the handler associated with *invalidates* to detect miss-speculation and to record WAR dependences. Handlers associated with *data value replies* are used to record RAW and WAR dependences. By suitably programming the handler functions, other monitoring applications (including those mentioned in Table. 1) can be supported on multicores [20].

Our work is related to prior works [11, 15, 18] which observe that exposing memory operations is crucial for a variety of monitoring related tasks. One approach in [15] integrates monitoring code into the coherence protocol where the memory operations are visible. A flexible solution proposed in [11] exposes the memory operations *directly to the programmer via user mode exceptions* so that a variety of monitoring tasks can be implemented without modifying cache coherence implementation. We also follow the approach proposed in [11]. However, our work differs almost entirely as it exposes different set of cache events so as to enable contemporary monitoring applications. While the *cache-miss* event is exposed for implementing cache coherence in software [11] and implementing software controlled multithreading [18], this event is not adequate for exposing ISMDs needed for detecting miss-speculation during software speculation and enabling replay for debugging.

We implemented *ECMon* in a cycle accurate multicore simulator [29]. As an instance of software speculation, we speculatively execute threads past active barriers [8, 14] and use *ECMon* to detect miss-speculation. Using this simple support, we find that the overall program execution time is reduced by an average of 12%, as opposed to 10% slowdown without this support, for the set of 7 parallel applications considered. As an instance of reliability, we show how *ECMon* can be used to capture and record the ISMDs at only 2.8 fold overhead for SPLASH-2 programs, a monitoring task which is prohibitively expensive to perform without this simple support. Thus the main contributions of this paper are as follows:

- The observation that exposing ISMDs to the programmer is crucial to performing a variety of monitoring tasks on multicores and *ECMon*, a lightweight, general purpose mechanism for achieving the above.
- ECMon is *programmable*, since the programmer can interact with ECMon via rich ISA support. We demonstrate the efficacy of *ECMon* with two different applications: *Speculation past barriers* and *Recording ISMDs*, each of which had *no prior software solutions* and required *specialized* HW support for implementing them.

2. ECMon: ARCHITECTURAL SUPPORT

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

Table 2: Exposed Cache Events.

In *ECMon*, we propose exposing cache events to the software; in effect, efficiently exposing the ISMDs to the programmer. Whenever the cache controller receives a cache event for a processor's local cache, it can be programmed to interrupt the processor, and call a predefined *handler* function, before responding to the event as shown in Fig. 2(a).

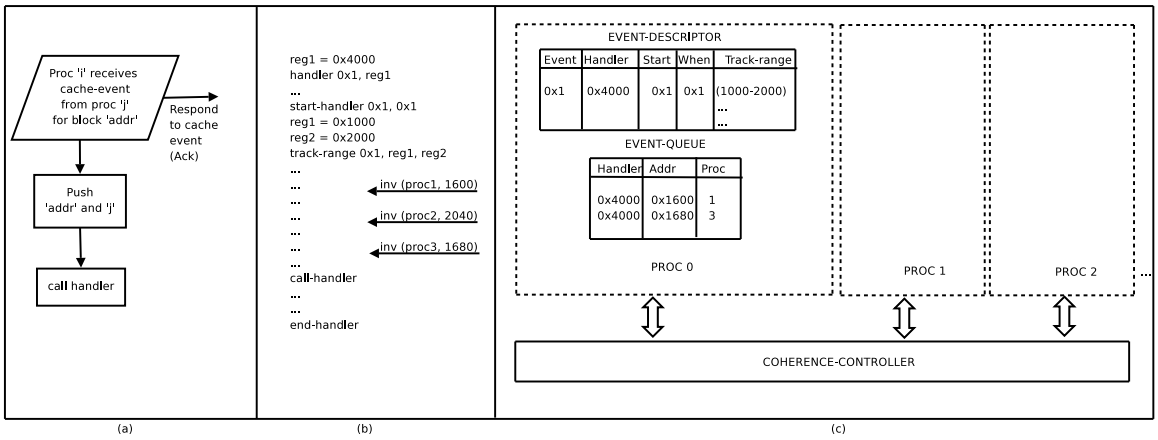


Figure 2: (a) ECMon semantics (b) New instructions added (c) Hardware structures added

For this discussion and our implementation, we consider a MOSI *directory based coherence protocol* with directory stored in the shared L2 cache, executing in an *inorder* multicore processor with local L1 caches.

(**Events and Handlers**) The cache events exposed for the applications in this work are the following: *(i) a processor sending/receiving an invalidate message*, (ii) *a processor sending/receiving a data value reply*, (iii) *a processor experiencing a read miss for a block uncached in any processor* and (iv) *a processor about to write back a block* as illustrated in Table. 2. While the first four events expose ISMPs exercised via the cache coherence network, the last two events expose ISMPs exercised via memory. The handler function is programmable and is defined based upon the requirement. Since the handler resides in user space, the semantics of the call to the handler is similar to a function call; the programmer is responsible for saving and restoring the values of registers that are used in the handler. However, the hardware is responsible for providing values to the handler as *function call parameters* as mentioned in Table 2. In general, the *block address*[1] and the *remote proc id* are the parameters. However, for the *send invalidate* event, since there are potentially multiple remote processors, the parameter for this event is the number of remote processors holding the invalidated block. Finally, it is important to note that while the handler function is called, the coherence controller *independently* responds to the cache event, as usual. In other words, the original coherence protocol itself is *unaltered* in ECMon.

(**ISA Support**) Through our proposed ISA interface, the programmer interacts with the processor and is able to effectively utilize ECMon support. The programmer notifies 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 address. We then give the programmer the ability to mark regions of code, where the *ECMon support is active*. For this purpose, we have the `start-handler` and `end-handler` instructions; the handler is actually called upon reception of the event only for execution within these two points. Furthermore, the programmer is given flexibility as to *when* the handler will be called, upon reception of an event. For this purpose, the `start-handler` instruction takes a *when* bit as one of its operands; a 0 indicates that the handler will be called as soon as the event is received; whereas a 1 indicates that the handler should be called only on *specific points*. If a 1 is specified as the operand, the programmer inserts the `call-handler` instruction (within *start-handler* and *end-handler*). When the processor receives an event, it does not call the handler immediately and only calls the handler when the `call-handler` is encountered. It is very useful for the programmer to control when the handler will be called. For example, in speculative execution we may need to call the handler after the speculation to verify its correctness. Likewise, this feature can be used to handle the *atomicity* problem associated with software monitoring [3, 23], as we shall see later. Finally, through the `track-range` instruction, we give the programmer the ability to specify the *range of block addresses* for which the handler will be called. The rationale for supporting this option is to limit the number of times the handler is called.

Having explained the purpose of each of the new instructions added at a high level, we now describe in detail with an example as shown in Fig. 2(b), the semantics of the instructions and the hardware structures that need to be added to implement the semantics, as illustrated in Fig. 2(c).

(**Handler instruction**) The programmer notifies the hardware through the `handler` instruction, what handler to call for what event. The handler instruction has two operands. The first operand is used to specify the *event code*, which is a predefined code for each cache event. For example, the *event code* $0x1$ may refer to the event when the processor receives an *invalidate* message. Through the second operand which is a register, the programmer specifies the *instruction address* of the handler to the hardware. In the above example, the handler resides in the instruction address *0x4000*. To maintain this information, each core of the multicore maintains an *event-descriptor* table. When the `handler` instruction is encountered it adds the *event code* and the handler's instruction address to the *event-descriptor* table as shown in Fig. 2.

(**Start-handler and end-handler instructions**) Through the `start-handler` and `end-handler` instructions, the programmer marks the region of code where the *ECMon support is active*. The handler is actually called upon reception of an event only for execution within these two points. The *start-*

[1]The address refers to the virtual address. Since the addresses seen in the coherence messages are physical, they are converted to virtual addresses and then passed as a parameter.

handler takes two operands. While the first operand specifies the *event-code*, the second operand is a one bit operand (called the *when* bit) which controls *when the handler should be invoked*. When the *start-handler* instruction is encountered, it sets the *start* bit and the *when* bit in the *event-descriptor*; on the other hand, the *end-handler* clears the *start-bit*.

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

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

2.1 Completeness

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

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

RAW, WAW dependences: *Proc 1* holding a block in exclusive state (due to a prior write to that block) can later write it back to the memory. A read (or write) to the same block by a different processor, *proc 2*, results in a RAW (or WAW) dependency between *proc 2* and *proc 1*, which is exercised through the memory. We expose two additional events to detect the above dependence. When a processor is about to *write-back an exclusively held block* to the directory, we expose this *write-back* event to the processor causing the *write-back*. Later when a different processor requests the block, it sends a read-miss to the directory. If such a *read miss* request is received by the directory and is *uncached* in any of the processors, we expose the above *read-miss* event to the processor causing the read miss. To summarize, we detect RAW (WAR) dependences by exposing the appropriate *write-back* and *read-miss* events.

2.2 Correctness

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

3. SPECULATION PAST BARRIERS

Barrier synchronization is commonly used in parallel programs when situations arise in a program when a given thread has to wait for other threads to arrive at a point before it can proceed. Upon reaching a barrier, a thread has to stall until other threads reach the barrier. Thus a thread that arrives at a barrier first, does no useful work until other threads arrive at the barrier and this amounts to the time lost due to the barrier synchronization. In order to reduce the time lost due to the barrier synchronization, compilers typically try to distribute equal amounts of work to the different threads. Even if this is the case, threads do not always execute the same code which in turn causes a variation in the arrival times. Moreover, even if each thread executes the same code, they can each take different paths leading to a variation in number of instructions executed. From our experiments, we found that the time spent on barrier synchronization can be as high as 25% of the total execution time for the set of programs considered. Furthermore, we also observed that the time spent on synchronization increased as the number of processors were increased.

To reduce the time spent during synchronization, we execute speculatively past barriers. We then use *ECMon* to detect miss-speculation and if there is one, we restore the state of the program back to the state at the time of barrier synchronization. The idea of executing past barriers while synchronization is pending has been proposed before [8, 14, 27]. While [8] is a compiler based approach to identify the code

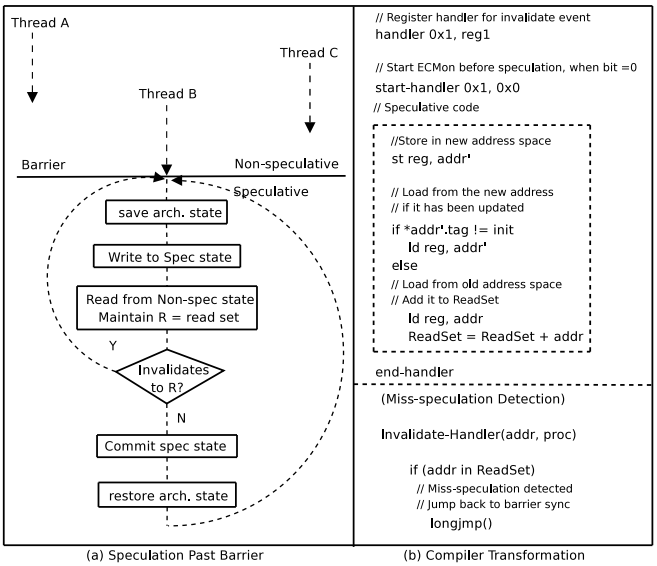


Figure 3: (a) High level Idea of Speculative Barrier and (b) Compiler Transformation

that can be executed safely past a barrier, [14] and [27] are speculative approaches that use hardware support to detect and recover from miss-speculations. Our approach is speculative, but takes advantage of exposed cache events so that detection and recovery from miss-speculations is performed in software in the handler. Our approach to speculatively executing a barrier using *ECMon*, as shown in Fig. 3(a), is as follows:

- When a thread reaches a barrier, we first checkpoint the architectural state using software (*setjmp*). This is done so that we can jump back to this state, when a miss-speculation is detected.
- We then create a safe address partition for the speculative thread to work on; in effect *isolating* the address space of the speculative and the non-speculative threads. The primary benefit of this isolation is that *name* dependences that manifest between the speculative and the non-speculative threads can be safely ignored, and do not cause a miss-speculation. Moreover, we do not need any rollback in case of a miss-speculation; we merely free the newly created address space and jump back to the saved safe architectural state. The compiler generates the speculative code and ensures thread isolation. One way to implement thread isolation is by making the speculative thread write to a different address space as shown in Fig. 3(b). All store instructions store their values into this new address space, each of whose memory words are tagged with *init* values. This will make sure that the speculative thread does not modify non-speculative state. However, loads have to be modified to ensure that they load from the new address space only if there has been a store to that location. A memory word tagged with an *init* value, implies that there has been no store to that memory location in the new address space; if this is the case, we proceed to load from the original memory address. We also add the address to the *ReadSet*, which we keep track to help in detecting miss-speculation.
- Since we dealt with name dependences using thread isolation, the only other way in which there is a miss-speculation is when a non-speculative thread (the one that is yet to arrive at the barrier), writes to a location that has already been read by the speculative thread. We detect this efficiently using *ECMon*. The compiler utilizes ECMon ISA support to ensure that miss-speculation is detected. The speculative thread registers a handler for the *receive invalidate* event and encloses the speculative code around `start-handler` and `end-handler` instructions. The *when* bit operand for the `start-handler` instruction is set to 0, meaning the handler is called as soon as the event is received. The speculative code, which ensures thread isolation as explained before, also tracks the set of memory locations that it reads, which is maintained in *ReadSet*. When a non-speculative thread writes to any of these locations, the coherence protocol will *invalidate* the corresponding block stored in the cache of the speculative thread. Using ECMon support, this invalidate event will interrupt the processor in which the speculative thread runs, and trigger the software handler associated with the *invalidate* event. Inside the handler we check whether the block has already been read by the speculative thread by checking if the block is a part of the *ReadSet*. If so, we know that there is a miss-speculation and hence we jump back to the state at the time of the synchronization as shown in Fig. 3(b).

3.1 An Example: Parallel Jacobi Iteration

Having explained our approach at a higher level, we illustrate our approach in more detail using the example shown in Fig. 4(a), that shows the parallel implementation of the *Jacobi iteration* using barriers. The code fragment describes the worker loop of the iterative solver wherein at each iteration, the value at each point is replaced by the average of the North, South, East and West neighbors. To parallelize Jacobi iteration, the grid is first partitioned into strips and each thread is given the task of performing the computation in its own strip. However, it is important to note that each strip needs to read values from its neighboring strips every iteration for computing the new values of the rows bordering other strips as shown in Fig. 4(a). It is precisely for this reason that barriers are introduced into the code, for they ensure that when a thread reads values from the neighboring strips, they are the values from the correct (previous) iteration. Finally, note that the main worker loop does two computations to avoid copying from one grid to another.

Why speculative barriers work?.

Let us assume that Thread B, working on strip B, has arrived at the barrier and speculatively executes past it. By doing this, thread B is speculating that whenever thread B reads from strip A (row n) and C (row 1), they have already been updated by threads A and C respectively and are the current values. It is important to note that this speculation will definitely succeed if threads A and C arrive at their barriers by the time *row n* from *strip A* and *row 1* from *strip B* are read by thread B. Furthermore, even if thread C has not arrived at the barrier, the speculation will succeed if thread C, merely, has updated *row 1* of its strip, by the time it is read by thread B; it is worth noting that this is very likely, since *row 1* of each strip is updated first. This explains why speculatively executing past a barrier is a good idea. First, barrier synchronizations maybe placed conservatively

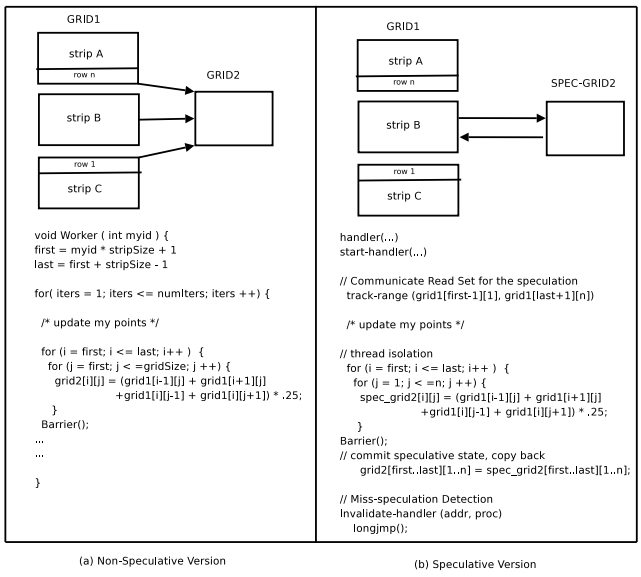


Figure 4: (a) The code for parallel execution of Jacobi Iteration: The grid is divided into strips and each parallel thread works on its strip, reading values from the boundary rows of neighboring strips. (b) shows how optimized code is generated for thread isolation and miss-speculation detection by utilizing static analyses

by the programmer or the compiler. Second, even if a barrier synchronization is required, the thread that has arrived at a barrier may only require that a subset of threads arrive at a barrier, before it can proceed. Finally, even if that subset of threads have not arrived at a barrier, they (the subset of threads) may have already updated the memory locations that are eventually going to be read by the speculative thread. We also observed in our experiments, for well partitioned parallel programs that use barriers, each thread works on shared data in its partition mostly, accessing data from other partitions infrequently (those that are in the boundaries of its neighbors).

Optimizing speculative code.

Our baseline implementation of thread isolation and detection of miss-speculation (Fig. 3) introduces overhead for the speculative thread, since it involves instrumenting loads for (a) maintaining the ReadSet (b) getting the recent most value. Furthermore, the speculative thread's handler needs to be called for every invalidate message it receives – it would be helpful if there was a way to filter out several messages which we are sure would not cause a miss-speculation. For several scientific codes that deal with arrays, static analyses can be used to optimize code generation for thread isolation. In the above example, when thread B reaches the barrier, it speculatively executes past the barrier and writes to the preallocated *spec-grid2* as shown in Fig. 4(b), instead of *grid2*. In other words, all the stores are transformed to write into new address space (*spec-grid2*). However, there is no need to transform the loads, since it can be figured out statically that all the loads get their values from *grid1*. Likewise, *ReadSet* can be figured out by the compiler statically and this can be used to optimize miss-speculation detection. The compiler figures out the *ReadSet* and communicates it via the `track-range` instruction. Recall that the track-range instruction specifies a range of addresses for which the handler should be called. Thus the handler is called only if the block that is invalidated is either from *row n* of *strip A*, or from *strip B* or *row 1* of *strip C* – the set of memory locations that is speculatively read by thread B, as shown in Fig. 4(b). Once the handler is called, we know there is a miss-speculation and we can simply jump back to the architectural state at barrier synchronization.

4. RECORDING ISMD

Deterministic Replay Debugging (DRD) [22, 35, 40] is a technique that helps programmers debug their program by replaying the exact execution that causes the bug to manifest itself. Naturally, the first step of DRD is the online recording of program information as it executes, so that replay can be enabled. Recording the execution of multithreaded programs involves the recording of ISMDs, since their order is non-deterministic. Software based approaches [35] are unable to record these shared memory dependences for multithreaded programs executing in multiprocessors. On the contrary, specialized hardware support, involving changes to the cache, cache coherency and processor pipeline, has been proposed in prior work [22, 40] to efficiently record these dependences. In this section, we discuss how ECMon support can be used to record these dependences. First we review how recording is performed in hardware based approaches and then derive our own ECMon based implementation.

We now briefly overview the steps involved in recording dependences in hardware recording systems as shown in Fig. 5(a). Each processor keeps track of its instruction count in an on-chip counter *instr-count*. Furthermore, each cache block is augmented with space to hold the instruction count in *cache-block[addr]*. Whenever the processor accesses that memory block, it writes the current instruction to it. This is done so that when that memory block is involved in an inter-processor dependence, the time in which it was last accessed can be remembered. Dependences are expressed as edges between processor ids along with each of their instruction counts [22, 40]. The key idea of recording ISMDs is based on the observation that in a multiprocessor with local caches, the cache coherence protocol is actually responsible for enforcing the above dependences and thus the coherence messages reveal the dependences. Thus hardware recorders piggyback cache coherence messages with instruction counts and the hardware takes care of recording these edges. Finally, before recording the dependency the hardware checks if the dependency that is currently recorded is automatically implied by a previously recorded dependency; if so, it will not log the current dependency. We illustrate the above steps with the following simple example with two processors.

4.1 An Example

Fig. 5(b) shows read and write operations prefixed by their dynamic instruction counts and also indicates the dependency exercised between processors. There are two dependences exercised in the above example: a WAR dependency with the write from processor 2 at instruction count 175 and the read from *proc 1* at instruction count 125. Likewise, there is a RAW dependency with the write from *proc 1* at

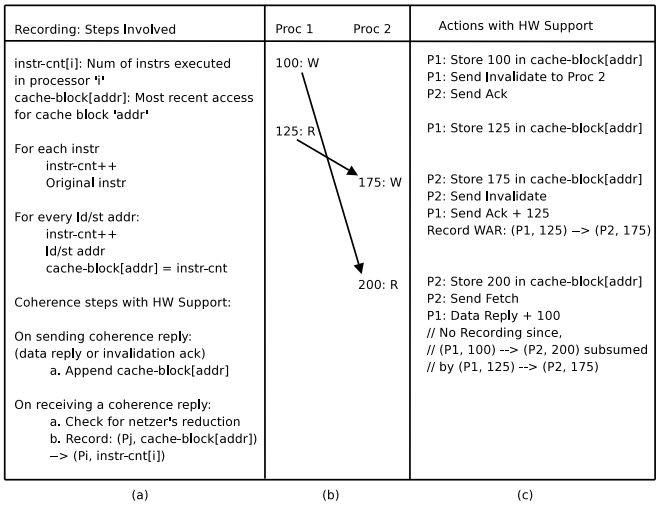


Figure 5: Recording ISMDs using HW support

instruction count 100 and the read from *proc 2* at instruction count 200. Recording ISMDs involves remembering the instruction counts at the time of the write and read, so that the same dependences can be enforced during replay. Let us first consider the actions required for recording the WAR dependency. When the read occurs in *proc 1*, we need to somehow remember the instruction count at the time of the read, so that the RAW dependency can be recorded when it is subsequently written in processor 2. To remember this count, an instruction count is added to every cache block. Thus as the read is executed, the value 125 is stored in the cache block associated with the read address. When the write is executed in processor 2, the instruction count (175) is similarly remembered in the cache block. Since it is a write, the coherence controller sends an invalidate message to invalidate shared copies of the block in other processors. Consequently, the same cache block that already resides in *proc 1* is invalidated. Once it is invalidated, *proc 1* sends an invalidate acknowledgment, *appending to it* the instruction count when it was last read. Using this, the dependence (P1, 125) → (P2, 175) is recorded.

Now let us consider the outer RAW dependency. When the write in *proc 1* is executed, the instruction count (100) is remembered in the cache block. It then proceeds to invalidate shared copies in *proc 2*. Thus when there is a read in *proc 2*, there is a read miss and the value is sent as a *data reply* from *proc 1*, as usual appended with it the instruction count when the block was written (100). However, before recording the dependency, hardware recorders perform the *Netzer's transitive reduction* [24] test, in which they basically check if the current dependence is actually implied transitively by previous dependences. In fact, in the above example, there is no need to record the outer RAW dependence since it is automatically enforced by the WAR dependence that has already been recorded. The recording of WAW dependences (not shown) takes place similar to RAW dependences.

4.2 Recording ISMDs using ECMon

Recording using ECMon is motivated by the fact that all the steps involved in recording dependences, excluding the ones dealing with coherence messages, can already be performed in software; with ECMon support, we can also now deal with coherence messages under software control. Using shadow memory support [23] we associate per processor instruction counts (*instr-cnt[p]*), which are incremented in software for every memory instruction. Likewise, we also maintain instruction count for every processor's cache block (*cache-block[addr]*) as well as memory block (*directory[addr]*) in software. We instrument all stores and loads in the program, with instructions that copy the processor's instruction count to the shadow memory associated with the processor's cache block for that memory access. The basic steps involved in maintaining these counters are same as shown in Fig. 5; only all the variables are actually stored in memory and the counters are maintained using instructions. We then use ECMon support by programming the software handlers to record ISMDs.

To illustrate the process at a high level let us consider the RAW dependency exercised between the two processors in Fig. 5(b). When the read, accessing block address, *addr*, occurs in *proc 2*, it sends a fetch message to *proc 1*, which contains the block in exclusive state because of the earlier write that happened in *proc 1*. Upon receiving the fetch message, *proc 1* attempts to send a *data-value reply* to *proc 2*. This triggers the software handler; within the handler, we access two values: the instruction count corresponding to the block address, *cache-block[addr]*, and the current instruction count of *proc 2*, *instr-count[2]*. Upon accessing the values, we are able to record the dependency (or skip recording), after checking for Netzer's transitivity condition. Recording of the WAR dependency in Fig. 5(b) proceeds among similar lines. When the write accessing block address, *addr*, occurs in *proc 2*, it sends an invalidate message to *proc 1*. Upon receiving the invalidate message, the software handler is triggered. Within the handler, we again access two values: the instruction count corresponding to the block address, *cache-block[addr]*, and the current instruction count of *proc 2*, *instr-count[2]*. Upon accessing the values, we are able to record the dependency, after checking for Netzer's transitivity condition.

4.2.1 Correctness Issues

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

(**Atomicity**) The first issue concerns the atomicity of the original and shadow memory operations [3, 23]. Recall that we require a separate store (denoted W_1') to update the instruction count of *cache-block[addr]*, for the original store, denoted by W_1. This means that these two operations are not atomic anymore. To see how this can cause correctness problems, let us consider the example shown Fig. 6(a), which shows the same RAW dependency. Let us first assume that we perform the shadow store after the original store (scenario 1). In the example, let us suppose that the read R from *proc 2* happens before W_1' but after W_1. This implies that *cache-block[addr]* is yet to be updated and contains a stale value. Consequently, $W_0 \rightarrow R$, is recorded instead of $W_1 \rightarrow R$. A similar correctness issue manifests itself, even if we perform shadow store before the original store, as shown in scenario 2 of Fig. 6(a), if read R from *proc 2* happens after the shadow store W_2', but before W_2. This implies that *cache-block[addr]* would have been updated by W_2'; but the read R still gets its value from W_1. Conse-

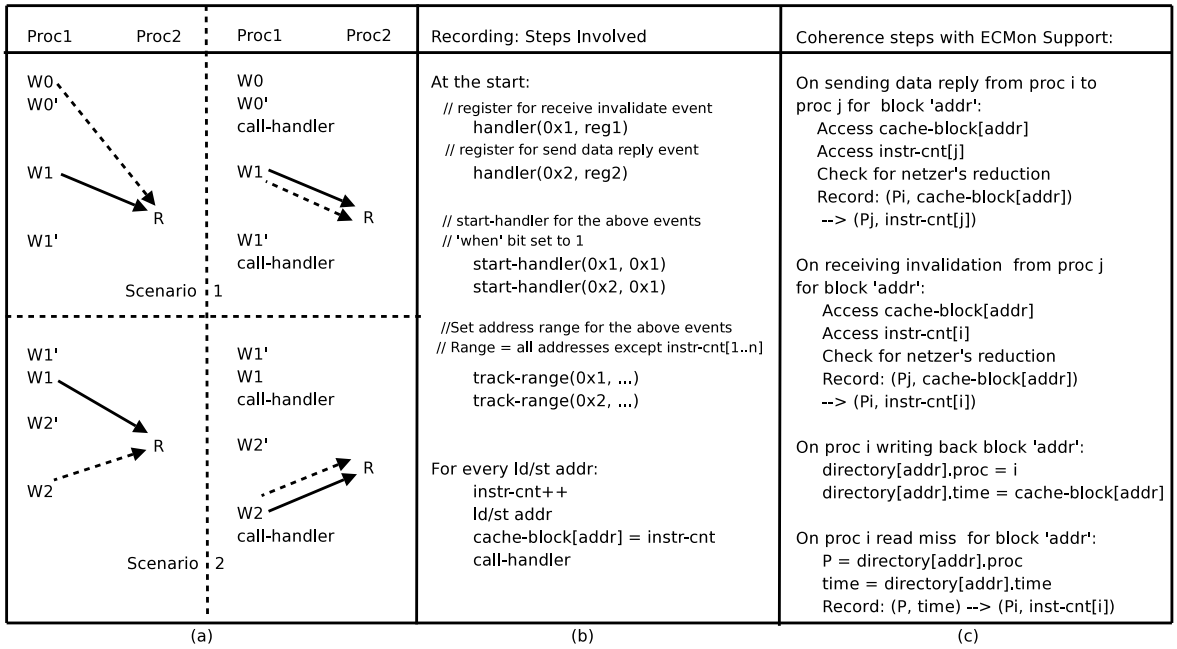


Figure 6: (a) Problems due to lack of atomicity and its solution, solid lines represent the exercised dependences, while dotted lines show the recorded dependences (b) Instrumentation involved for recording using ECMon (c) Work done in the software handlers.

quently, $W_2 \rightarrow R$, is recorded instead of $W_1 \rightarrow R$. To deal with this issue, we simulate the effect of the original and shadow memory instruction executing atomically via ECMon support. We use the ECMon facility to control *when* the handler is invoked, to simulate atomicity. Instead of forcing the handler to be invoked immediately, upon reception of the event, we make it call the handler at certain key points. More specifically, we instrument the `call-handler` instruction after every memory instruction/shadow memory instruction pair W_1/W_1'. This will ensure that if an event is received in between (between W_1 and W_1'), we still have to wait for both of them to finish executing, before the handler is called. In the above example, Fig. 6(a), let us first consider the case where shadow store is performed after original store. When the event is received after W_1 but before W_1', the handler is not called then; it is only called when the `call-handler` is encountered which is after W_1'. This ensures that *cache-block[addr]* is updated and contains the current value. Similarly, in the second scenario, when the event is received after W_2', but before W_2, the handler is not called yet; it is only called after W_2. This means that the data value reply is after W_2 and hence R gets its value from W_2, which is the dependency that is recorded.

(**Correct Instruction counts**) The second issue concerns the value of *instr-count[2]*, the instruction count of *proc 2* when it is accessed in the handler from *proc 1* for recording the dependency. Recall that *proc 1* is providing a *data value reply* in the first place as *proc 2* experienced a miss due to R. The value of *instr-count[2]* will thus be correct, if it is not changed by a future write, from *proc 2*. To ensure this, we make sure (via software) that *proc 2* waits until the handler finishes in *proc 1* as shown in Fig. 7. To accomplish this synchronization, we maintain a synchronization variable $ready(i, j)$ for every processor pair i and j.

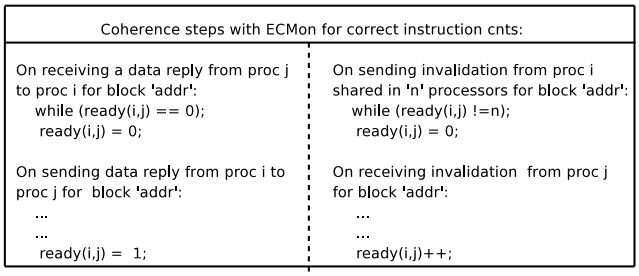


Figure 7: Maintaining correct instruction counts

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

(**Avoiding nested handler invocations**) It is worth noting that the access of *instr-count[2]* within the handler of *proc 1* will cause a miss, since the variable *instr-count[2]* will be in exclusive state in *proc 2*. Thus, the value of *instr-count[2]* will finally be obtained via a *data value reply* from *proc 2*. However, we do not want this *data value reply* to cause a handler call in *proc 2*. This is because we are not interested in recording this dependency, since it is not due to the original program. We deal with this problem by taking advantage of the `track-range` instruction – we make sure that the range of addresses provided does not include *instr-count[1..n]*. This will make sure that the handler is not called for such events.

(**Cache block replacements**) When cache blocks are replaced, the dependences are then exercised through the memory, rather than coherence system. We are able to record dependences exercised through memory, since ECMon is also able to expose such dependences using the *write-back* and *read-miss* events. First, let us consider the recording of the RAW dependency shown in Fig. 5(b). Let us suppose that *proc1* writes back the block to the directory before it is read by *proc 2*. The *write-back* event triggers the software handler, within which we save the processor id and the instruction count corresponding to the block address, *cache-block[addr]*, into the directory[2]. Later, when the read occurs in *proc 2*, it results in a *read-miss*. This triggers the software handler associated with the read-miss within which we are able to record the RAW dependency by accessing the information we saved in the directory, during the *write-back* event.

(**Thread switches**) When a thread is scheduled out of a processor, the processor can still hold cache blocks accessed by the previous thread. For example, *proc 1* can hold an exclusive block dirtied by *thread 1*, even after *thread 1* is swapped out of *proc 1*. Then later when there is cache miss for the same block in *proc 2* running *thread 2*, the block may be provided by *proc 1* as a data value reply, even though it is currently idle. In this case, we may not be able to record the above RAW dependency, since *proc 1* is idle. To handle this case, whenever a thread is scheduled in or out of a processor, we *record instruction counts of each of the processors*. This dependency, also known as *strata* [21] in prior work, will transitively subsume the missed dependences due to thread switches. We also record the mapping between the thread ids and processor ids, at the time of a thread switch. Recall that whenever we record dependences, we record the dependences between processors; using this mapping between the processors and threads, we can then derive the dependences between the threads.

4.2.2 Summary

We summarize the instrumentation involved for recording ISMDs using ECMon in Fig. 6(b). At the start of the program we register handlers for both events: *(i) a processor receiving an invalidate message*, (ii) *a processor about to send the data value reply*. We then add the `start-handler` instruction, with the *when* bit set to 1, to handle the atomicity problem. The *track-range* instruction is added, so that addresses involving accesses to *instr-cnt[1..n]* do not cause handlers to be called. As discussed earlier, we increment *instr-cnt* and update *cache-block[addr]* for every memory instruction. For the purpose of simulating the effect of atomicity, we introduce the *call-handler* after this update. The handlers for the events are summarized in Fig. 6(c) and are self-explanatory. The first two events are for the purpose of recording dependences exercised through coherence controller, while the last two events are for the purpose of recording dependences exercised through memory.

5. EXPERIMENTAL EVALUATION

In this section, we present the results of our experimental evaluation of ECMon support in supporting speculative barriers and recording ISMDs.

[2]We associate additional memory with each directory entry in SW for this purpose.

5.1 Implementation

We implemented our ECMon support in the SESC [29] simulator, targeting the MIPS architecture. The simulator is a cycle accurate multiprocessor (supports multicore mode) simulator which also simulates primary functions of the OS including memory allocation and TLB handling. To implement ISA changes, 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. Finally, we implemented our coherence algorithms for a MOSI based directory protocol for an 8 core processor with a shared L2 cache, which holds the directory entries. In our implementation of the coherence protocol, the L1 cache does not notify the directory on shared cache replacements. Furthermore, when a processor *proc i* writes back its owned block to the directory, the directory continues to store the id of the processor in the *owned* field, in the record for the block. Recall that the above steps helped us to deal with cache block replacements. The architectural parameters for our implementation are presented in Table 3. We now briefly overview our implementation and the benchmarks used for speculative barriers and recording of ISMDs.

5.1.1 Speculative Barrier

The benchmarks we chose for speculative barrier application are listed in Table 4 In choosing benchmarks to show the efficacy of speculative barrier, we were looking at those that have numerous dynamic barrier synchronizations. We first looked at the popular SPLASH [39] benchmarks suite, and found that they did not have significant number of dynamic barrier synchronizations. Then we wanted to look at parallelizing sequential codes with loops, since parallelizing these codes typically involves the addition of barrier synchronizations. This led us to use *Livermore Loops*, which have a set of challenging kernels, with hard to extract fine-grained parallelism. We used kernel 2, which is a part of a *Cholesky conjugate gradient* code and kernel 6, which is a *general recurrence relation*. We used the parallelized version of these codes, which were described in [31]. We then looked at the SPEC *openmp* benchmarks, which contain compiler directives for parallelizing loops. The semantics of openmp entail that processors synchronize with each other at the end of a parallel section, which can result in numerous barrier synchronizations. We used *equake* and the *swim* programs from the above suite. We also used *Bitonic Sort* and *MST* from the *Olden* [30] benchmarks suite which performs parallelizable sorting. It is important to note that the Olden benchmarks, which operates on pointers and irregular data structures, offer a contrast with the Livermore loops and spec programs which operate on arrays. In parallelizing Bitonic sort, we faithfully followed the annotations provided in the source code [30], which resulted in the introduction of several barrier synchronizations. For our final benchmark, we used the parallelized version of the *Jacobi iteration*, which we already discussed.

5.1.2 Recording ISMDs

We chose the popular SPLASH [39] benchmark suite (in Table 5) to evaluate our ECMon support for helping in recording ISMDs. We performed instrumentation by modifying the assembler output generated by the gcc-4.1 com-

Table 3: Architectural Parameters

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

Table 4: Speculation: Benchmarks

Programs	Description
Jacobi	Iterative solver
Livermore 2	Cholesky gradient
Livermore 6	Linear recurrence
Equake	Earth quake simulation
Swim	Weather Prediction
Bisort	Bitonic sort
MST	Spanning tree

Table 5: Recording: Benchmarks

Programs	Description
BARNES	Barnes-Hut alg.
FMM	fast multipole alg.
OCEAN	ocean simulation
RADIOSITY	diffuse radiosity alg.
RAYTRACE	ray tracing alg.
WATER-NSQ	nsquared
WATER-SP	spatial

piler. We did not actually output the dependences to a file, but maintained them in a specially maintained circular buffer, similar to Bugnet [22]. We maintained time stamps by instrumenting loads and stores with additional instructions that incremented a counter and stored the time stamps in shadow memory. We reserved one register specifically for maintaining these time stamps, so that it need not be spilled and restored for every memory instruction. Finally, 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.

5.2 Performance: Speculation past Barriers

In this section, we wanted to measure the execution time reduction with speculation past barriers with ECMon support. Secondly, we also wanted to see the effect on the execution time if we did not have ECMon support. But first, we briefly discuss the characteristics of the programs used. As we can see from the table in Fig. 8, the above programs spend significant percentage of their execution times waiting for other processors to reach the barrier, ranging from 18% for Bitonic sort to as high as 61% for Livermore loop 2. This in turn gives scope for processors reaching the barrier early, to speculate past the barrier. The graph in Fig. 8 represents the percentage reduction in execution times by speculating past active barriers with and without ECMon support. Recall that ECMon support provides support for efficient detection of miss-speculation; whereas without ECMon, we would have to add additional checks at commit time. More specifically, we would have to check at commit time, that the values read by speculative code have not since been updated. As we can see, with ECMon support, we are able to reduce the execution time significantly. The percentage reduction in execution times ranges between 6% (Bitonic sort) and 24% (livermore loop 2). On an average we could achieve a 12% reduction in execution time by speculating past barriers. However, without ECMon support we observe that the program *slows down* by around 10%. To gain further insight as to why we were getting the speedup, we measured how the original time spent in synchronization (without speculation) was now being spent with speculation using ECMon support. As we can see from Fig. 8(c), about 37% of the original time spent in barrier is now channeled into performing useful work. We can also see that the time spent inside the handler recovering from miss-speculation is relatively less (about 5%), owing to small number of miss-speculations. However, significant time (about 58%) was spent performing copies.

From the above set of experiments we were able to observe that (for the programs considered) speculating past barriers can lead to significant savings in execution time. However, to efficiently speculate past barriers we needed support for efficient miss-speculation detection and ECMon could be used for the above purpose.

5.3 Performance: Recording Dependences

In this experiment, we wanted to measure the execution time overhead of performing recording in software with ECMon support. As we can see from Fig. 9, the overhead for performing recording ranges from 2.2 fold execution time overhead for the *BARNES* benchmark through 3.6 fold overhead for the *FMM* benchmark. On an average (harmonic mean), recording causes a 2.8 fold execution time overhead.

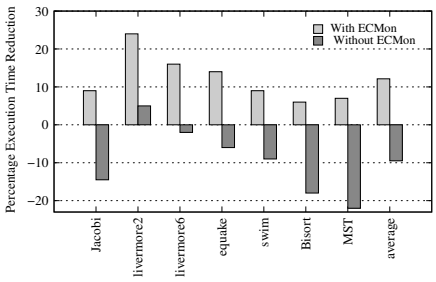
To understand the causes for this overhead, we split the overhead into several contributing categories as shown in Fig. 9. The first category is the overhead due to the execution of additional instructions to maintain instruction counts; recall that in the software version we actually needed to instrument loads and stores to maintain these counts. As we can see, this is the major contributing factor of the overhead, accounting for 89% of the overhead on an average. We are now in a position to reason why the overhead was high for the *FMM* program – since it had a large percentage of memory instructions (around 50%), significant time was spent to maintain the instruction counts. On the contrary, *BARNES* and *OCEAN* programs, with relatively fewer memory instructions (around 30%), causes lesser overhead. It is important to note that, since most of time is spent on instrumentation, *only 11%* of the execution time is spent executing the handler recording the ISMDs. This vindicates ECMon's main motivation: efficient support to expose dependences to the software. We additionally split the time spent in the handler into 3 categories: time spent to log RAW, WAR and WAW dependences. As we can see most of the time (out of 11% time spent recording dependences) is spent recording RAW dependences.

From the above set of experiments, we could observe that with ECMon support we could efficiently record ISMDs at only 2.8 fold execution time overhead. It is worth noting that without ECMon support it would be prohibitively expensive to perform recording; several additional instructions are needed for each memory instruction to derive dependences, in addition to thread serialization to handle the atomicity problems which could result in overhead at least an order of higher magnitude [20, 23].

6. RELATED WORK

(**SW based monitoring**) Although there are several software based monitoring tools, they are thwarted by ISMDs, which makes them either inapplicable [25, 26, 35] or inefficient [7, 23] on multicores. For example, *Flashback* [35] is a software based *record-replay* tool applicable for (single and) multithreaded programs running on a uniprocessor. Software tools that perform monitoring [23, 25, 26] need sepa-

Programs	% time barrier
Jacobi	25
Livermore2	61
Livermore6	42
equake	32
swim	24
Bisort	18
MST	28



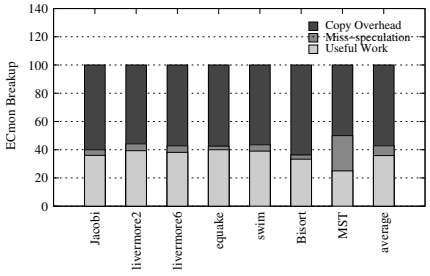
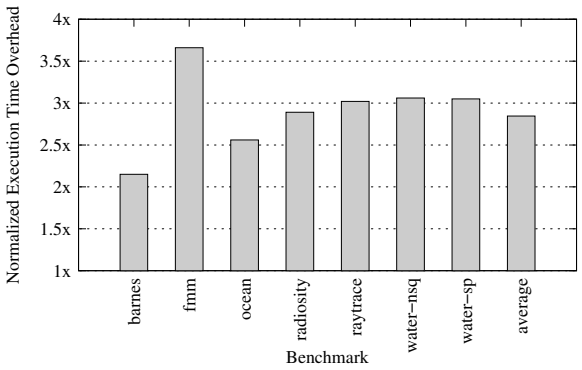


Figure 8: (a) Speculative Barrier Characteristics (b) Execution Time Reduction with and without ECMon (c) Breakup of how original time spent in barrier, is now spent with ECMon



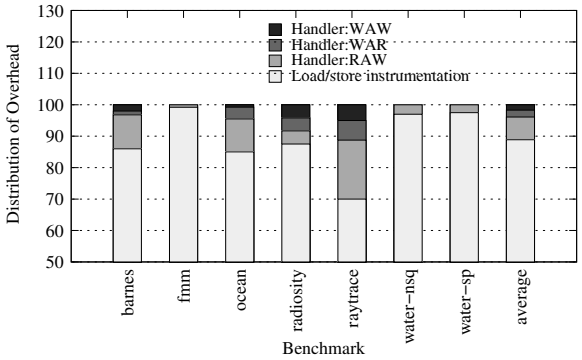


Figure 9: Dependence Recording Overhead and Break up of Overheads

rate instructions to perform monitoring, giving rise to races between data and meta-data when executed on multicores [3]. Since the ISMDs can be captured using *ECMon*, these races could be avoided. In this work, we showed how *ECMon* could be used to program two very different monitoring applications: speculation past barriers and record-replay for debugging. In [20], we showed how a class of monitoring applications that utilize *shadow memory* [23] can be programmed for multicores using ECMon.

(**HW based monitoring**) There has been several proposals that use specialized hardware support to ensure the reliability of software [5, 21, 22, 36, 40]. The hardware support involved in each of the above proposals is non-trivial and involves changes to the processor pipeline, caches, cache coherence and memory subsystem. For example, *FDR* [40] and *Bugnet* [22] involve augmentations to the cache coherence protocol to capture dependences, changes to processor pipeline to maintain instruction counts, addition of per block counters to the caches and addition of other hardware structures to optimize recording. In our work, we isolate the minimal hardware support needed (exposing cache events), and perform all other tasks required in software. This approach, in addition to increasing the flexibility and programmability, makes *ECMon* applicable to a variety of monitoring tasks. Recently there has been work to design general purpose hardware support for a variety of monitoring applications [2, 38]. However, the applications described in this work (speculation and recording ISMDs) can not be directly handled with the above proposed support.

(**Transactional memory**) The problem of detecting cross-thread dependence violations at run time is known as *conflict detection* under *Transactional memory* (TM) [10] parlance. STM systems [1, 13] instrument loads and stores with *read/write barriers* to detect conflicts. On the contrary, HTM systems [9, 10, 28] like TLS systems [4, 9] rely on hardware support (modifications to caches/cache coherence) to detect conflicts. Hybrid TMs [6, 16, 33] use hardware to perform the simple and common case and rely on software support to handle the uncommon case. Recent proposals on hybrid TM have proposed hardware support for conflict detection. While *SigTM* [16] uses hardware signatures for conflict detection, *RTM* proposed the *Alert-on-Update* [34] mechanism which triggers a software handler when specified lines are modified remotely. Whereas the hardware support involved in *ECMon* is similar to *Alert-On-Update*, we additionally show how other cache events (in addition to remote update), can be used for performing a variety of monitoring applications including speculation bast barriers and recording of ISMDs.

7. CONCLUSIONS

In this paper, we proposed *ECMon*, a simple interface for exposing cache events to the software for capturing ISMDs. We then showed how these dependences could be used for a variety of applications including speculation and recording shared memory dependences. More specifically, we showed how ECMon support could be used to speculatively execute past barriers, and cause a performance improvement of 12% for the parallel codes considered. We also showed how ECMon support could be used to perform recording of ISMDs on a multicore, at 2.8 fold execution time overhead.

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

8. REFERENCES

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

[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*, 2008.

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

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

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

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

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

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

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

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

[11] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing memory operations: Memory performance feedback mechanisms and their applications. *ACM Trans. Comput. Syst.*, 16(2):170–205, 1998.

[12] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *ISCA*, pages 265–276, Washington, DC, USA, 2008. IEEE Computer Society.

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

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

[15] M. Martonosi, D. Ofelt, and M. Heinrich. Integrating performance monitoring and communication in parallel computers. In *SIGMETRICS*, pages 138–147, 1996.

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

[17] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *ISCA*, pages 289–300, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[18] T. C. Mowry and S. R. Ramkissoon. Software-controlled multithreading using informing memory operations. In *HPCA*, pages 121–132, 2000.

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

[20] V. Nagarajan and R. Gupta. Runtime monitoring on multicores via oases. In *Operating Systems Review*, 2009, to appear.

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

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

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

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

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

[26] 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.

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

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

[29] 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.

[30] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. Program. Lang. Syst.*, 17(2):233–263, 1995.

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

[32] 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.

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

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

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

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.