Resolve: Enabling Accurate Parallel Monitoring under Relaxed Memory Models

Evangelos Vlachos[1,3], Sotiria Fytraki[3], Phillip B. Gibbons[2], Michael A. Kozuch[2] and Babak Falsafi[3]
[1]Carnegie Mellon University [2]Intel [3]École Polytechnique Fédérale de Lausanne

Abstract

Hardware-assisted instruction-grain monitoring frameworks provide high-coverage, low overhead debugging support for parallel programs. Unfortunately, existing frameworks are ill-suited for the relaxed memory models employed by nearly all modern processor architectures—e.g., TSO (x86, SPARC), RMO (SPARC), and Weak Consistency (ARMv7). For TSO, prior proposals hint at a solution, but provide no implementation or evaluation, and fail to correctly handle important corner cases such as byte-level dependences. For more relaxed memory models such as RMO and Weak Consistency, prior frameworks deadlock*, rendering them unable to detect any bugs past the first deadlock!*

This paper presents Resolve*, the first hardware-assisted instruction-grain monitoring framework that is complete, correct and deadlock-free under relaxed memory models. Resolve is based on the observation that while relaxed memory models can produce cycles of dependences that deadlock prior approaches, these cycles can be overcome by consulting the dataflow graph of the application threads being monitored, instead of their program order. Resolve handles all possible cycles arising in relaxed memory models, through a careful approach that uses both dataflow-based processing and versioning of monitoring state, as appropriate. Moreover, we provide the first quantitative characterization of the cycles arising under RMO, demonstrating that* such cycles are prevalent and persistent*, and hence deadlock is a real problem that must be addressed. Yet they are not so frequent or complex, so that Resolve's overheads are negligible. Finally, we present a simple and novel hardware mechanism for* properly synchronizing updates *to monitoring state under relaxed memory models, improving performance by up to 35% over the judicious use of memory fences.*

1. Introduction

Writing correct shared-memory parallel programs is a notoriously difficult task. Implicit data sharing and synchronization among concurrently running threads lead to non-deterministic behaviors, making reasoning about what a parallel program does at any given point in time extremely hard. As a result, not only do parallel programs tend to have many bugs, but it is much harder to locate and correct each one of them. Further, the relaxed memory models [1] delivered by modern processors, including x86 total store order (TSO) [16], SPARC v9 relaxed memory order (RMO) [30], and ARM v7 weak consistency (WC) [3], provide additional opportunities for unwary programmers to introduce subtle concurrency bugs.

As an example, Dekker's algorithm, shown in Figure 1, behaves properly under the sequentially consistent (SC) model that many programmers expect—but not on a machine that provides TSO (or weaker) consistency, such as the x86 architecture. The programmer may expect that the load operation in ❶ of *flag[i]* will execute after the store operation in ❸ (and ❷ after ❹), but if the stores are delayed in a store buffer, the load operations may bypass the stores. This is permitted in relaxed memory models, and implies that *both* loads may return the value "FREE", thereby enabling both threads to enter the critical section concurrently.

Initially: flag[i]=flag[j]=FREE

flag[i]=BUSY ❸	flag[j]=BUSY ❹
while(flag[j]==BUSY){ ❶	while(flag[i]==BUSY){ ❷
// if not i's turn, release	// if not j's turn, release
// flag and busy wait	// flag and busy wait
}	}
// critical section	// critical section
flag[i]=FREE	flag[j]=FREE

Figure 1: Dekker's algorithm executing on two threads seeking to enter a critical section. The numbers indicate a possible effective ordering of memory operations under a non-SC model (e.g., TSO), resulting in both threads entering the critical section.

Instruction-grain monitoring is a class of powerful debugging tools capable of helping programmers with many types of bugs, including concurrency bugs that would arise under sequential consistency as well as the even subtler bugs that arise with relaxed memory models. In such tools, a monitoring entity, "monitor," (e.g. a software thread or co-processor) is associated with each application thread and checks the validity of every relevant instruction executed by that thread. Different monitoring tools check for different notions of validity (e.g., memory safety), by maintaining suitable *metadata* (e.g., which memory has been allocated) as the application executes. While software-only instruction-grain monitoring [29, 20, 11, 7] suffers from high runtime overheads (often 30–100x), hardware-assisted instruction-grain monitoring [18, 36] achieves negligible overhead by running the application and the monitor on different resources (e.g., using a core for the application thread and a dedicated co-processor for its monitor) and providing hardware-assisted inter-thread data dependence tracking.

Existing Frameworks Deadlock and Are Incorrect. While prior hardware-assisted instruction-grain monitoring frameworks [18, 36] provide high-coverage, low overhead debugging support for parallel programs, they are limited to supporting only sequential consistency. The relaxed memory models provided by nearly all modern processors render these prior frameworks *ineffective* due to deadlock problems, and *incorrect* in their handling of metadata:

Problem 1: Dependence-cycle deadlock. Instruction-grain monitoring frameworks typically process monitored events in "dependence" order, which is a combination of inter-thread data dependences and program-order dependences. In contrast

to what is possible under SC, however, this dependence order may include cycles under relaxed memory models [40] (see also Figure 2). Prior work has briefly considered this problem for TSO, and introduced metadata versioning (i.e., copies of metadata) as a way to overcome the generated dependences and allow concurrent processing of the involved application events [18, 36]. However, these prior studies present no implementation or evaluation for TSO. Perhaps as a result, important corner cases are overlooked: [18] can suffer from deadlock when its versioning tables reach their capacity and [36] fails to correctly handle byte-level inter-thread dependences (see Section 3). For more relaxed memory models (i.e., RMO, WC), the problem of cyclical dependences becomes even more complex, and as a result, *prior hardware-assisted monitoring frameworks will deadlock!*

Problem 2: Improperly synchronized metadata access. Akin to Figure 1, if a monitor processing an event e issues a metadata update (corresponding to e) followed by a flag update (indicating done processing e) and the two writes update memory out of order under a relaxed memory model, then a different monitor waiting on the flag may read the un-updated metadata when processing an event e'. Because *prior hardware-assisted monitoring frameworks ignore this issue*, they may incorrectly judge the validity of application instructions (e.g., if e followed by e' is a bug that is missed due to reading the un-updated metadata). Moreover, fixing this problem by using locks or memory fences is too heavyweight.

Resolve: Deadlock-free, Accurate Monitoring under Relaxed Memory Models. This paper presents *Resolve*, the first hardware-assisted parallel monitoring framework that supports relaxed memory models, by solving both of the above problems. We only assume that the memory model supports cache-coherence; even arbitrary bypassing between loads and stores to different cache blocks (such as in RMO) is handled properly.

Resolve uses a novel and effective algorithm for detecting and resolving dependence-cycles, handling various subtle corner cases. Resolve overcomes the dependence-cycle deadlock problem based on the observation that such cycles can be overcome by consulting the dataflow graph of the application threads being monitored, instead of their program order. Resolve handles all possible cycles arising in relaxed memory models, through a careful approach that uses both dataflow-based processing and metadata versioning, as appropriate. While considerable complexities arise in developing an efficient parallel algorithm leveraging the above observation, monitoring tools are able to process most application events based on the strict program order, and Resolve imposes no additional overhead. For the remaining events where a potential non-SC behavior is encountered, the monitoring process is allowed to relax the event processing order to that dictated by the data-flow graph. Because this case is infrequent over the course of the application, there is no need for hardware support—the algorithm is incorporated into the monitoring framework runtime.

In addition, Resolve overcomes the metadata access synchronization problem, via a simple and novel hardware mechanism that provides low-overhead ordering for updates to shared metadata under relaxed memory models. Our study shows that the mechanism improves Resolve's performance (as measured by IPC) over the prior state-of-the-art approach (i.e., enforcing ordering by judicious use of memory fences) by up to 23% and 35% for the MEMLEAK and ATOMCHECK monitoring tools, respectively.

Desirata	**[14, 13]**	**[18]**	**[36]**	**Resolve**
Fast (speed)	N	Y	Y	Y
Supports many tools	Y	Y	Y	Y
Limited changes to core	Y	Y	Y	Y
For SC: accurate?	FP	Y	Y	Y
deadlock-free?	Y	Y	Y	Y
For TSO: accurate?	FP	Y	B	Y
deadlock-free?	Y	N	Y	Y
For RMO/WC: accurate?	FP	N	N	Y
deadlock-free?	Y	N	N	Y

Table 1: Comparison of flexible instruction-grain parallel monitoring frameworks. FP: Suffers from false positives (can flag events that are not errors), but not false negatives (unreported errors). B: Inaccurate in the presence of byte-level inter-thread dependences.

Finally, an important aspect of this work is in understanding the characteristics of the cycles arising under relaxed memory models. While prior work has qualitatively described the cycles arising under various memory models [1], we are interested in *how frequently* they actually occur in applications and *how complicated* they are to resolve. This paper provides a quantitative characterization of the cycles arising under RMO, such as their frequency and length, as a result of the monitored application exhibiting a potential non-SC behavior. Our study demonstrates that cycles are indeed prevalent and persistent and hence deadlock is a real problem that must be addressed. Yet they are not so frequent or complex, so that Resolve's overheads are in fact negligible.

Resolve's approach can be compared against the only other framework that handles relaxed memory models without suffering from deadlock: Butterfly/Chrysalis Analysis [14, 13]. Resolve (like [18, 36]) leverages hardware-assisted inter-thread data dependence tracking to avoid the false positives of Butterfly/Chrysalis Analysis, and to reduce overheads by up to two orders of magnitude.

Table 1 summarizes the advantages of Resolve over prior frameworks. Note that frameworks that deadlock are unable to detect *any bugs* past the first deadlock!

Resolve's design can not only be used to extend [18, 36] to handle TSO, RMO, and WC, but also the work by Nagarajan et al. [25, 24] that similarly exposes coherence events to a checker process. Deterministic record and replay [39, 40] can also benefit, as our study shows: (i) there is no need to generate versions or to identify non-SC behavior in hardware–during replay, the system has all the required state to generate versions, and recorded non-SC behavior will appear as a cycle of dependences; (ii) how to resolve a cycle during replay (Section 4.2); and (iii) how to reason about cycles under a wider set of relaxed memory models.

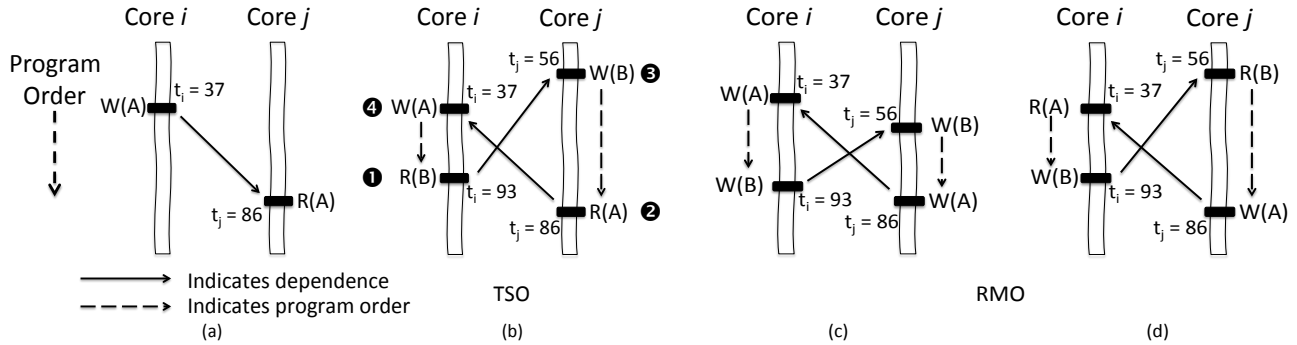


Figure 2: Ordering of concurrent instructions by observing cache coherence activity. (a) Identifying an inter-thread *Read-after-Write* dependence. (b, c) A cycle of inter-thread dependences under TSO and RMO. Read instructions in (b) and write instructions in (c) bypass the committed stores that reside in the store buffer, reaching the memory hierarchy first. (d) A cycle of inter-thread true (i.e., reads-from) dependences under RMO. Similar to (b), but in this case store instructions bypass earlier load instructions. ❶ - ❹ indicate memory order.

2. Parallel Monitoring Background

Hardware-assisted monitoring tools for multi-threaded applications organize monitoring as a distributed process. The monitoring task is assigned to multiple monitors, one for each application thread [8, 33, 34, 32, 18, 9, 36]. The monitoring infrastructure captures application events and delivers them to the monitors for further processing. Events are examined by the monitors in program order. They can represent instructions executed by the application or higher-level functionalities, such as function calls (e.g., *malloc()*). A monitor invokes an event handler for each event observed, which will typically check and/or update the shared metadata maintained by the monitors based on the contents of the event record. Monitoring frameworks have two requirements.

Requirement 1: Inter-Thread Event Ordering. To properly maintain the shared metadata, the order in which monitors process application events must "match" the order in which the same events occurred during application execution. That is, if the relative order of two application events has an impact on the monitors' metadata, then the monitors must process the two events in that order. For most monitoring tools, two application events from different threads have such impact only when there is a data dependence between them (e.g., two writes to the same virtual address). Thus, hardware-assisted monitoring tools have adopted the use of application cache coherence activity as a means of collectively directing the monitoring process [18, 36]. This approach was initially introduced by earlier work on deterministic record and replay [39, 40].

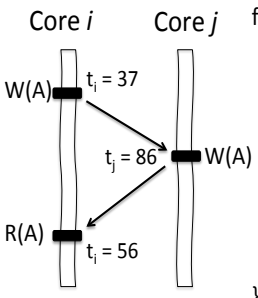
We outline the approach taken by [36] ([18] is similar). For concreteness, we will assume a multicore architecture in which each core has private L1 instruction and data caches and a slice of the unified last level cache (L2), although the approach generalizes to other processor architectures. Associated with each L1 cache block is a timestamp that records the last time an instruction accessed it. (A single timestamp for multiple cache blocks could also be used [36].) The notion of "time" is local to the core and represented by a monotonically increasing number (e.g., number of instructions committed so far). The combination of the core id i and its local "time" t_i is called a *dynamic instruction identifier* $I^i_{t_i}$, with older instructions having a lower-value identifier than younger ones. By associating dynamic instruction identifiers with the cache blocks those instructions access, inter-thread communication (and event ordering) at cache-block granularity can be inferred. L1 caches notify the L2 every time they evict a block. To avoid losing the sharing pattern of a cache block when evicted by an L1, the L2 caches eviction timestamps from different L1s.

Figure 2(a) presents an example of how cache coherence activity can be used to infer ordering of concurrent instructions. At local time $t_i = 37$, core i updates cache block A, setting the assigned timestamp to I^i_{37}. Later, core j requests shared access to block A, misses in its local L1 and sends a coherence request to core i, which replies by providing the requested cache block and its assigned timestamp. Upon receiving the coherence reply, core j sets the assigned timestamp for block A to I^j_{86}, where $t_j = 86$ is its local time for the read, and identifies a *(happens-before) dependence* between instructions I^i_{37} and I^j_{86}, denoted $I^i_{37} \rightarrow I^j_{86}$ in the figure. Such a dependence arc is delivered to the monitor that handles the events of application thread j. The monitor interprets the dependence as "wait until the monitor for application thread i completes the processing of I^i_{37} before processing I^j_{86} or any later events." In this way, the monitors process these events in the order they occurred during application execution.

Requirement 2: Metadata Access Synchronization. When monitors are not waiting on dependences, they are free to process events as quickly as possible. However, parallel monitoring, being a parallel application itself, must properly synchronize the monitors' accesses to the shared metadata, given the processor's memory model.

Figure 3 describes the steps a monitor takes when processing an event in existing frameworks [18, 36], as well as an example of two application threads accessing an address A. Monitor j will initiate processing of application instruction I^j_{86}, by first checking if there are any inter-thread dependences that have to be satisfied. Assuming event I^i_{37} has been processed, it will then proceed with performing the actual processing of I^j_{86}, potentially updating metadata. Monitor j will then update its progress by incrementing some shared variable (not shown), allowing monitor i to process event I^i_{56}.

For metadata access synchronization, prior work makes the observation that for a wide range of monitoring tools,




```
for all events do {
  if( event.hasDependences() )
    wait_for(event.source, event.time);

  // enter_critical_section
  process_event(); // update metadata
  // exit_critical_section

  update_progress();
}
```

Figure 3: (left) An example execution of two application threads accessing address *A*. (right) The set of steps taken on every event processed. Updating progress must appear to happen after the effects of processing the event are globally visible.

metadata have a 1-to-1 mapping to application data, and are *updated* only as a result of processing an application store instruction [36]. Because such store instructions are always ordered implicitly or explicitly with regard to other dependent application instructions due to cache coherence, ordering events (dependence arcs) are always present when attempting to update metadata. Thus, because monitors wait on such dependence arcs, at any time only one monitor for a contended address will be allowed to proceed (multiple contending monitors can proceed if all are only reading). Consequently, *under sequential consistency, no additional synchronization is required* [36, 18, 25], because SC ensures that the store in update_progress() in Figure 3 happens only after the loads and stores in process_event() are globally visible.[1]

3. Problems under Relaxed Memory Models

Prior hardware-assisted parallel monitoring frameworks follow the design described in Section 2, which works correctly under SC, using the commit time of the instruction accessing a block as the block's timestamp. Unfortunately, under relaxed memory models, such frameworks are ineffective due to deadlock problems and incorrect due to racing metadata accesses, as discussed in this section.

Problem 1: Deadlock due to Dependence Cycles. Under any relaxed memory model, the time when a memory instruction commits can be different from the time its effects are globally visible (i.e., ordered in memory). For example, a store instruction may commit and reside in the store buffer for some time (not yet visible to other cores) before updating a block in L1. In contrast, a typical load instruction is considered to be globally visible at commit time. Thus, load and store instructions often update cache block timestamps at different times; loads at commit time and stores at the time they leave the store buffer [40].

The lack of a global order of events among concurrent threads, reflected on the different times cache block timestamps are updated for different instructions, results in observing cycles of happens-before relationships. Figures 2(b)–(d) show three scenarios of execution under relaxed memory models, where the observed coherence activity together with the program order generates a cycle of dependences. Under TSO, load instructions are allowed to commit and bypass earlier committed store instructions that reside in the store buffer. In Figure 2(b), core *i* first accesses block B, updating its timestamp to I^i_{93}. Similar to that, core *j* accesses block A updating its timestamp to I^j_{86}. Store instructions to blocks A and B are then released from the store buffers of core *i* and *j* respectively, and both experience a coherence miss. The replies to their request are carrying the timestamps of the corresponding load instructions that are logically later in time. Two inter-thread dependences are generated, $I^i_{93} \rightarrow I^j_{56}$ and $I^j_{86} \rightarrow I^i_{37}$, which combined with the program order at each thread form a cycle of dependences. Figures 2(c) and 2(d) show two similar cycles of dependences, caused by stores bypassing earlier stores and loads, respectively, under RMO. In addition, RMO (and Release Consistency [1]) allows load instructions to bypass earlier loads, which can also lead to a cycle of dependences with stores from other cores (not shown).

The examples in Figures 2(b)-(d) show cycles of dependences, formed in the cache-block level. These cycles represent potential non-SC behaviors, depending on whether the instructions involved access the same parts of the cache blocks or not. In either case, these cycles of dependences cause prior hardware-assisted parallel monitoring frameworks to *deadlock*, because the monitors for the threads in the cycle are each stuck waiting on the progress of another monitor in the cycle.

State-of-the-art Solutions to Dependence Cycles. Prior work has identified and briefly studied the problem of dependence cycles under a TSO memory model [18, 36]. Following a similar approach as the one described in RTR [40], they introduce "*versioning*" of metadata in order to overcome a cycle of dependences. Under this approach, the two instructions at the ends of an inter-thread dependence that is part of the cycle can be processed in any order, as long as two distinct copies of the shared metadata location are provided to the corresponding monitors. In Figure 2(b) a copy of the metadata for application address *B* (we focus on B for now) takes place before processing the *W(B)* and is used when processing *R(A)*. Processing of *W(B)* can proceed out of order, as the versioning already reflects the inter-thread dependence, producing a new version for the metadata of address *B*. After processing both events, versioning control ensures the latest version of the metadata location propagates to global state (the one produced when processing *W(B)*).

Although the intuition behind metadata versioning for TSO is correct, neither prior works present a correct, deadlock-free solution. First, [36] fails to perform correct metadata versioning in the presence of byte-level inter-thread dependences, as illustrated in Figure 4. Under this scenario, a cycle of dependences is generated between two threads sharing memory locations *A* and *B*, however one of the dependence arcs connects instructions that access the same cache block but different bytes (*R(B)* $\rightarrow$ *W(B+8)*). Because only the *receiving* core observes the dependence (i.e., core *j* for $I^i_{93} \rightarrow I^j_{56}$), it cannot identify the mismatch between the two addresses.

[1] Note: Atypical monitors that update metadata while processing load instructions must recognize that (a) the relative ordering of loads is not well-defined, even under SC, and (b) because the concurrent loads in this case do induce (read-after-read) dependence arcs, proper metadata serialization *does* require additional synchronization [36].

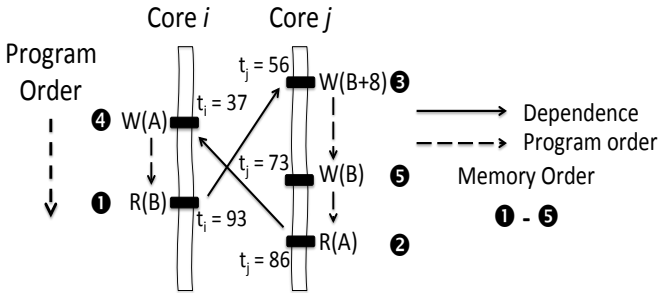


Figure 4: [36] fails on byte-level dependences.

(Tracking and passing along with dependence arcs the set of bytes touched in each cache line would be prohibitively expensive.) Thus, not only a version for the wrong location of metadata will be produced, but also the inter-thread dependence will be ignored (as the versioning is assumed to already reflect it), introducing a race between the processing of *R(B)* and *W(B)*.

Second, [18] too ignores the issue of byte-level dependences, but because its approach already tracks application data addresses, it may be possible to modify it to handle such dependences without additional overheads. However, it can suffer from deadlock, as follows. It uses two hardware tables for storing information associated with coherence events performed by the application but not yet "processed" by the monitors. These tables must be kept a small, bounded size because of the requirement for supporting associative lookups. If a monitoring thread falls behind, the number of unprocessed events grows and can quickly reach the capacity of the tables. (This was not a problem in the performance study in [18] because only SC was studied, but TSO has larger windows of reordered events [12].) This in turn can mean that a memory event needed to unblock a cycle cannot be inserted into its table, resulting in deadlock!

Moreover, neither prior approach considers the types of cycles arising in more relaxed memory models such as RMO and WC, as noted earlier in Table 1.

Problem 2: Racing Metadata Accesses. Recall from Section 2 that because SC ensures that the store in update_progress() happens only after the loads and stores in process_event() are globally visible, there are no data races in accessing metadata. Under memory models that relax the order of stores to different addresses (e.g., WC, RMO), however, the progress update store may bypass the metadata loads and stores, allowing monitor *i* in Figure 3, for example, to proceed with processing I^i_{56}, causing a metadata race. Because *prior hardware-assisted monitoring frameworks ignore this issue*, they may miss a bug due to reading stale metadata, for example. A naive solution for preventing such races is through the use of high-level synchronization primitives such as locks, but acquiring and releasing a lock for processing every event is too costly. An alternative solution would be to interleave a memory fence between processing the actual event and updating progress; however, this approach is overly expensive, as we will show in Section 7.3.

In Sections 4 and 5 we show how Resolve solves the deadlock due to dependence cycles problem and the racing metadata accesses problem (without the high overheads of locks or fences), respectively.

4. Resolving Cycles of Dependences

The examples presented in Figure 2 highlight the two components of the cause of cyclical dependences: *i*) dependences that appear to causally relate future instructions from one thread with current instructions from the dependent thread due to re-ordering of application instructions, and *ii*) the in-program-order view of the execution of all the threads by the monitors. Being able to resolve a cycle requires negating one of the two components. Because the inter-thread events may be true (i.e., reads-from/RAW) dependences, which may be impossible to ignore, our solution enables monitoring tools to process events out of program-order.

In this section we present a complete framework that is able to resolve all cycles discussed earlier, while requiring no additional hardware support beyond that described in Section 2 (also adopted by earlier work [36, 18]). The solution accommodates relaxed memory models that provide cache-coherence (i.e., a partial order on all accesses to a single cache block, where all such pairs of accesses involving at least one write are totally ordered), regardless of the type of bypassing (loads bypassing loads/stores and stores bypassing loads/stores) allowed by the model. For the remainder of the paper we focus on RMO, because it is the most challenging as it allows for all four possible instruction re-orderings. Any solution for such a model also applies to any less relaxed memory model (e.g., TSO).

4.1. Cycles of True Dependences

The cycle presented in Figure 2(d) represents the most challenging case that prior work cannot accommodate and involves two true inter-thread dependences. Because the inter-thread dependences are true, causality ensures that at most one of the dependences inferred from program order involves an actual dataflow dependence. We observe that the load instruction of such a dependence (e.g., I^j_{56}) is not able to complete its execution until the value from the producing store arrives. Along with that load, any data-dependent instruction in program order is also blocked. If the later store I^j_{86} on the same core that participates in the cycle is data-dependent on the load, then we can deduce that the two stores are indirectly ordered as well ($I^i_{93} \rightarrow I^j_{56} \rightarrow I^j_{86}$). Following the same reasoning, because the load instruction I^i_{37} is data dependent on the store I^j_{86} from core *j*, it is also indirectly ordered with the later store from the same core ($I^i_{93} \rightarrow I^i_{37}$). From this, we can safely conclude that the store instruction I^i_{93} is data-independent of the earlier load; otherwise, the core would not be able to re-order it with respect to I^i_{37}.

More formally, we observe that: the dataflow graphs of the threads executing on cores *i* and *j* remain acyclic after connecting the nodes that are related by true inter-thread dependences. Figure 5(a) depicts this property (which holds for any number of cores participating in a cycle of true inter-thread dependences) for the cycle in Figure 2(d). Note the lack of an arc from I^i_{37} to I^i_{93}. This property of acyclic dataflow graphs

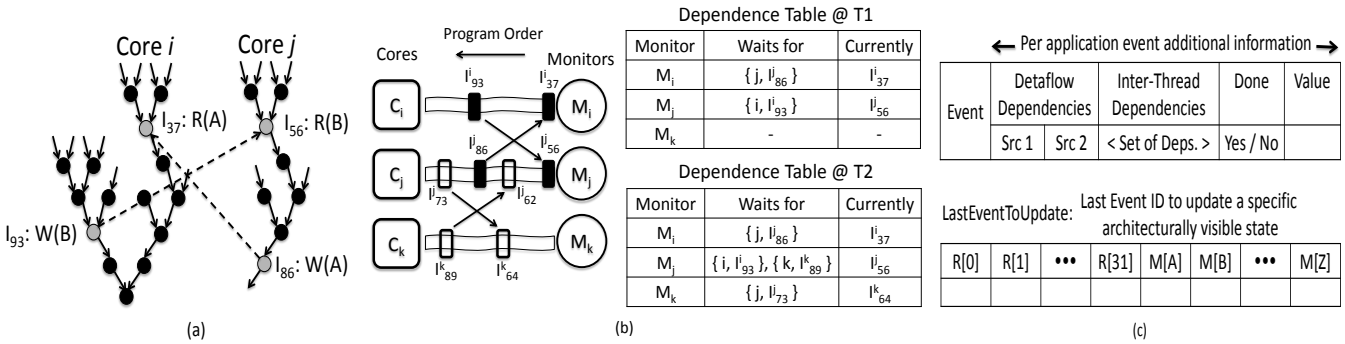


Figure 5: (a) Dataflow representation of the example in Figure 2(d). (b) The monitoring threads, $M_{\{i,j,k\}}$, consume application events produced by cores, $C_{\{i,j,k\}}$, in program order. At (wall-clock) time T1, M_i and M_j deadlock, and at time T2, a separate deadlock occurs between M_j and M_k. During step 1, monitors update the *Dependence Table* to indicate on which dependence(s) they are waiting. (c) Additional information collected per application event throughout the execution of the algorithm (top), and keeping track of the latest event that updates a specific architectural visible state during Step 3 (bottom).

enables the monitoring system to decide which events can be processed out of program order in the presence of a cycle of true inter-thread dependences, without losing a precise view of the application execution.

4.2. The Algorithm

Overview. In the common case, monitors process events serially until a deadlock is experienced as a result of potentially non-SC behavior. When a deadlock is encountered, cycle resolution is initiated. The algorithm is organized as a 5-step process. During Step 1, the monitors experiencing a deadlock, identify the ranges of execution that collectively constitute the cycle. Because cycles may occur due to arcs being induced by cache line false-sharing, during Step 2, the monitors substitute each recorded (cache-block-level) dependence with the equivalent set of byte-level dependences. If the cycle persists, monitors independently build the data-flow graph for each range of execution involved in the cycle (Step 3). Observing the data-flow order ensures that there are no cycles of true dependences, as shown is Section 4.1. During Step 4, the monitors break the cycle by processing the application events involved based on the data-flow, rather than program, order. Furthermore, use of metadata versioning and metadata address renaming allows us to ignore any inter-thread dependences due to WAW and WAR, ensuring the absence of cycles with any type of inter-thread dependences. After all events are processed, the monitors' global state is updated (Step 5).

Step 1: Identify the cycle. A *Dependence Table* is maintained in a shared space and tracks for which event completions each monitor is waiting; this table is organized as a single-writer/multiple-readers space, to avoid synchronization costs. After a monitor has spent a predefined amount of time waiting for a dependence, it updates its entry with the dependence it is waiting for and its current progress. It then checks if the monitor that is responsible for satisfying its dependence is waiting on some other monitor. By walking the dependence path, each monitor independently can identify if there is a cycle of dependences. (Our study indicates that a timeout period of 10K cycles ensures a minimal number of table accesses without imposing significant stall overhead.) Figure 5(b) shows such an example, where at time T1 monitors M_i and M_j have experienced a cycle, while monitor M_k continues uninterrupted.

Upon a deadlock, the monitor with the lowest *id*, of those involved, assumes the role of *master* to coordinate actions throughout the steps of the algorithm. The master consults the *Dependence Table* to identify the regions of execution from every core that participates in the cycle. For the example in Figure 5(b), the master will initially identify regions $\{I^i_{37}$ - $I^i_{93}\}$ and $\{I^j_{56}$ - $I^j_{86}\}$ for monitors M_i and M_j respectively.

The slave monitors will scan their corresponding regions to identify: i) if there is an event dependent on the progress of a monitor other than the ones involved in the cycle (e.g., I^j_{62}), or ii) if there is an event dependent on the progress of a monitor past the identified regions (not shown in the figure). If the first scenario is true, cycle resolution has to wait for the non-involved monitor either to satisfy the dependence or to eventually be part of the cycle. Figure 5(b) shows the latter case, where at time T2 monitor M_k forms a cycle with monitor M_j. Under this scenario, the resolution process expands its search *in space* to include monitor M_k. If on the other hand, the second scenario is true, the identified regions are just expanded to involve the additional events that need to be processed. We refer to this as expansion *in time*.

Conceptually, at the end of step 1, the monitors involved in the cycle will have created a graph data-structure, with no incoming edges, for the cycle.

Step 2: Expanding to byte-level dependences. The recorded dependences indicate how the application threads share data on a cache block granularity, which may result in cycles caused by false-sharing (and other problems—recall Figure 4). However, arcs due to false-sharing may not be naively eliminated because they may subsume true dependences. In Figure 6(a) cores *i* and *j* access different addresses from block *B*. The recorded dependence between I^i_{87} and I^j_{56} explicitly orders these two events, but also implicitly orders any event that preceded I^i_{87} with any event that followed I^j_{56}.

In the following steps of the algorithm we will process events from different dataflow paths out of program order. This introduces the danger of losing the implicit ordering between I^i_{87} and I^j_{77}, if, for example, I^j_{77} is part of a different

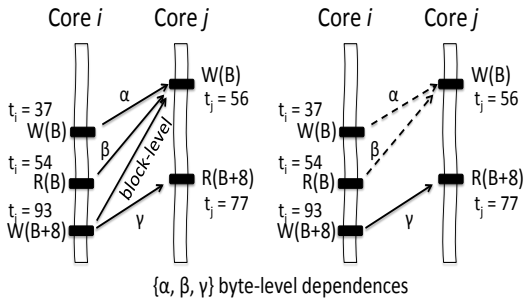


Figure 6: (a) In step 2, the block-level dependence (the only arc, initially) is eventually replaced by the three byte-level ones. (b) During Step 4, WAW and WAR inter-thread dependences (α, β) are ignored due to metadata versioning.

dataflow subgraph than I^j_{56}. To avoid losing such information, we replace block-level dependences with byte-level ones, in order to relate all the application events that accessed the same addresses from all the participant threads. A byte-level dependence is one that connects two instructions from different threads that access the same bytes of memory. Figure 6(a) shows how the block-level dependence $I^i_{87} \rightarrow I^j_{56}$ will be replaced by the three byte-level ones α, β and γ.

To identify all the byte-level dependences, all participating monitors calculate, for every block-level dependence $I^i_x \rightarrow I^j_y$ on a cache block B, the transitive closure between all instructions within the region of the cycle $I^i_{x'}$ that precede I^i_x in program order ($x' \leq$ x) and all instructions within the region of the cycle $I^j_{y'}$ that follow I^j_y in program order ($y' \geq$ y), which access block B. From the set of dependences generated, only those that connect instructions accessing at least one common byte are kept. The set of byte-level dependences generated accurately represents all the happens-before relationships existing at byte-level granularity.

At the end of step 2, the cycle exposed and captured during step 1 will include only byte-level dependences. If the cycle still persists, the algorithm will move to the next step. Otherwise, the algorithm terminates and the monitors process the events in the region of the cycle while considering the new byte-level arcs rather than the block-level ones.

Step 3: Building the dataflow graph. Each monitor thread independently builds the dataflow graph of the execution it is assigned. For every event in that execution region, it maintains the additional information shown in Figure 5(c top). The monitor scans the events assigned to it in program-order, and for every event, it performs two tasks. First, it checks if the source operands of the current event are produced by a previously examined event by maintaining and consulting a table of architectural visible state (*LastEventToUpdate* in Figure 5(c bottom)). If the source operands are found in that table, it copies the *Event IDs* to the *Dataflow Dependences* field, marking the dataflow dependence with the events providing the source operands to the current event. Second, it marks the entry of the destination operand to the *LastEventToUpdate* with the current Event ID. By performing these two tasks for all the events in the range examined, each monitor has a view of the execution it is assigned similar to the one in Figure 5(a).

Step 4: Out of program-order processing of events. Based on the dataflow dependences marked during step 3, each monitor thread has the ability to identify independent events from different dataflow paths. To do so, the monitor iterates over all events multiple times, and on every iteration processes the ones that are independent. In this way the notion of program order is preserved within each dataflow path.

For every event, the monitor checks if there are any dataflow or inter-thread dependences. If all source operands are ready (the events noted in the *Dataflow Dependences* field are marked *Done* and have produced a value) and there are no inter-thread dependences then the event can be processed independently of the state of any preceding event. The value produced after processing the event is kept in private space (*Value* entry in Figure 5(c top)) to avoid modifying the precise global view the monitoring system had up to the point when the algorithm was initiated. We refer to that *Value* as a *version* of the specific metadata location. In Step 5 we discuss what version of a metadata location should propagate to global state, if more than one is available.

If one or more inter-thread dependence arcs are incident upon the current event, the type of the dependence is considered. Any output (WAW) or anti-dependence (WAR) (dashed dependences in Figure 6(b)) over memory locations between different threads can be safely ignored for synchronization purposes because (i) the latter event does not consume a value generated by the former event and (ii) resulting values from processing events (versions) are saved in thread-private space (*Value* entry in Figure 5(c top)). This is analogous to register renaming, but for memory locations. However, if there is a true (RAW) dependence, the monitor must check if the source event has been processed, and if so, it can read the requested value from the *Value* entry of the source monitor. Otherwise, the event cannot be processed, and the monitor tries to find the next event that is ready to be processed. Because only true dependences are considered during this step, freedom from cycles is guaranteed as described in Section 4.1.

Step 5: Update global state. Output values produced by processing a cycle's events are kept in the per-monitor private space. During the last step, the global metadata state needs to be updated with these results. To do so, the master iterates over all regions to identify the last updates to any location, and updates the global state accordingly. The last updates to registers are easy to locate as they correspond to the end state of the *LastEventToUpdate* table used during step 3. Last updates to private memory (e.g., stack) or shared memory accessed only by only one core (e.g., one sharer heap space) are also easy to locate, because these addresses are found in the *LastEventToUpdate* table of a single monitor. For updates to shared locations by more than one thread, the master thread identifies the last event to update a location by leveraging the inter-thread dependence arcs for a particular address to identify the last update (similar a topological sort). After step 5 is completed, the monitor system returns to its initial mode of processing the multiple event streams in program order.

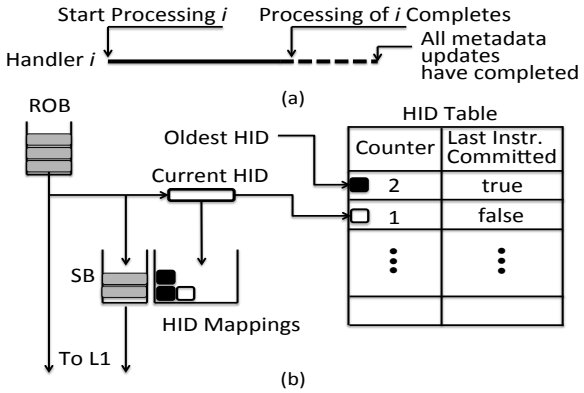


Figure 7: (a) A timeline of the events identified by our mechanism. An event is advertised as processed when all the metadata updates are globally visible. (b) The mechanism to identify when the effects of a handler are globally visible. Store instructions belonging to different handlers are monitored, and the corresponding store buffer entries they occupy are being tracked.

5. Fast Metadata Access Synchronization for Relaxed Memory Models

In Section 3 we introduced the problem of racing metadata accesses under relaxed memory models. To solve this problem, we observe that for a given monitor, most of the time, no other monitor is actively waiting on its current progress. Specifically, the times a monitor will require knowledge of the progress of some other monitor are at least as infrequent as the L1 misses of the monitored application. This property enables us to delay the advertisement of event-handler completion until the system can provide some guarantees about whether or not metadata updates have completed. Hence, we propose a simple hardware mechanism that keeps track of the store instructions that reside in the store buffer in order to identify when all the updates of a handler are globally visible. As soon as this happens, the mechanism marks the corresponding event as completed and advertises progress accordingly.

Figure 7(a) shows a timeline of the relevant events as well as a block diagram of the mechanism. The last instruction of every handler is marked to assist identifying handler boundaries. The *Current Handler ID* (HID) register uses this mark to keep track of the handler that is currently committing instructions. The size of the register depends on the maximum number of handlers that may have committed (or they are in the process of doing so), but their corresponding store instructions still reside in the store buffer. Fortunately, handlers spend most of their instructions on metadata address calculation or value comparisons, putting low pressure on the L1 and the store buffer, and thus allowing the store buffer to naturally drain by itself. Based on our evaluation, 4 bits are more than sufficient. Every entry of the store buffer is extended with a set of HIDs, one for every handler that has a store instruction residing in the corresponding entry. More than one HID may be assigned to a store buffer entry if coalescing of stores is allowed by the memory model (e.g., RMO). Fortunately, a store buffer entry remains *active* for a very short period of time, during which no more than three handlers are executed. Based on our evaluation, no more than 4 HIDs per store buffer entry are required. The *HID Table* tracks the number of store buffer entries associated with a specific handler, as well as if the last instruction of the handler has committed. It is organized as a circular buffer and indexed by the handler ID. For HID registers of 4 bits in size, a store buffer of 64 entries and counters of 6 bits in size we introduce approximately 140 bytes per monitoring core.

Every time a store instruction commits, it occupies an entry in the store buffer, which is tagged by the *HID*. If no other store of the current committing handler resides in that entry, then the *HID Table* is indexed based on the value of the *Current HID* register, and the corresponding counter is incremented. In Figure 7, stores of the current handler (noted by the white box) occupy one store buffer entry, while the previous handler (noted by the black boxes) occupies two. When an entry leaves the store buffer, the counters corresponding to the HIDs occupying the specific entry will be decremented. Finally, when the last instruction of a handler commits, its entry in the *HID Table* is also updated. We can safely conclude that the effects of a handler are globally visible when the counter value is 0 and its last instruction has committed. A handler is considered completed when its effects are globally visible and it is the oldest in the *HID Table*. After this happens, a corresponding progress update can take place, resulting in properly synchronized metadata under relaxed memory models. We assume the update is initiated by hardware as described in [36]. In the form of a generated store, it can be then inserted into the store buffer and allowed to drain as any other store.

Table 2: Experimental Setup

Simulator description	
Core	Ultra Sparc III ISA 3-wide dispatch / retirement 96-entry ROB, 64-entry SB, RMO model
L1 Caches	64KB, 4-way, 64B block 2-cycle load-to-use, 2-ports, 32 MSHRs
L2 NUCA Cache	Unified 1MB per core, 16-way 64B block, 10 cycles hit latency
Main Memory	90 cycles access latency
Interconnect	4x4 2D mesh
Benchmarks	**Input**
PARSEC	simlarge
FMM	64K particles
Ocean	1026 x 1026 matrix
Water	2197 mols

6. Experimental Setup

Simulation Setup. We use cycle-accurate full system simulation of a 16-core shared-memory CMP system using *Flexus* [37]. Flexus extends Simics [35] with cycle-accurate models of out-of-order cores, memory hierarchy and on-chip interconnect. Flexus models the SPARC v9 instruction set architecture, and can simulate the RMO memory model. To evaluate our metadata access synchronization mechanism, we extend Flexus to model a parallel monitoring architecture similar to ParaLog [36], which includes a hardware-supported dependence tracking mechanism based on RTR [40] (as de-

scribed in Section 2). Detailed parameters of the simulated system are shown in Table 2(top). We dedicate half of the simulated cores to the 8 application threads and half to the 8 monitoring threads. The events of each application thread are communicated to the paired monitoring thread through a memory-mapped event stream buffer [6].

We follow the SMARTS sampling methodology [37], and the execution samples are selected to cover a representative part of the application's parallel section. In all benchmarks studied, the parallel section dominates the application's execution time. For parallel sections organized as multiple iterations, we cover at least one iteration, and for some benchmarks two iterations. For the rest of the benchmarks that are not organized in this fashion, we cover at least 1 billion instructions. For each measurement, we start simulation from a warm state (warm caches, branch predictors, etc.), run 2 million cycles of detailed cycle-accurate simulation to warm up queues and interconnect states, and ensure that all metadata are properly allocated and initialized. Then, we collect measurements for the subsequent 1 million cycles.

Benchmarks. We include benchmarks from the SPLASH-2 [38] and PARSEC [4] benchmark suites, as listed in Table 2(bottom). PARSEC benchmarks are used unmodified. For SPLASH-2 benchmarks, we use efficient spin-based synchronization primitives, appropriate for the high performance computing domain, which do not cause an application thread to be scheduled off. All benchmarks use correct synchronization primitives for the RMO memory model.

Monitoring Tools. For performance evaluation we used two instruction-grain monitoring tools. A parallel version of AtomCheck [19], an atomicity violation detection tool, and a parallel version of MemLeak, a memory leak detection tool [21]. AtomCheck is looking for accesses from multiple threads to a single address, the interleaving of which may cause atomicity violation. MemLeak tracks the use of pointers throughout the execution of a program, and uses reference counting to identify objects that are leaked.

AtomCheck threads share a global table of metadata (1 byte per application word) to save the previous thread id that accessed a specific address. In addition, every thread has a private table of metadata (1 byte per application word) to save the type of its last access (read/write) to a specific address. AtomCheck has the property of possibly producing metadata updates when processing application loads, which requires the corresponding handlers to be organized in two paths; a fast one (synchronization-free) and a slow one, which requires synchronization [36]. MemLeak requires 1 metadata word per application word (32-bit), to mark the addresses that keep a pointer, as well as the object ID they are pointing to.

7. Evaluation

Our experimental study seeks to provide support for two key aspects of our mechanisms:
1) *Necessity and efficacy:* We provide a quantitative characterization of the cycles arising under RMO, as a result of the application exhibiting potential non-SC behavior. This will demonstrate that cycles are indeed prevalent and persistent and hence deadlock is a real problem that must be addressed. Yet their relative infrequency ($<$ once every 10M instructions) implies that cycle-resolver (Section 4) is invoked only rarely.
2) *Performance:* We provide two sets of performance results. First, we evaluate the overhead of the cycle-resolver mechanism, showing that it is negligible. Second, we provide an IPC-based evaluation demonstrating the performance gains of our proposed synchronization mechanism.

Table 3: Statistical characteristics of cycles. All cycles involved 2 participant cores, except that 12.5% of the FMM cycles involved 3 participants.

Benchmarks	Average Length	Frequency (per MI)	Persistence (of 200 runs)
Canneal	86	0.041	52
Fluidanimate	157	0.0003	54
FMM	144	0.008	200
Ocean	75	0.02	178
Streamcluster	230	0.0003	2
Swaptions	43	0.017	62

7.1. Quantitative Characterization of Cycles under RMO

Just because cycles *might* occur in RMO runs does not indicate *how frequently* they actually occur in applications, *if at all*. In this section, we study the cycles that arise under RMO. To this end, we run our benchmarks uninterrupted on the system (with no monitoring taking place) under the RMO memory model, and observe cycles of inter-thread dependences being formed as a result of the application exhibiting potential non-SC behavior. For every benchmark we report the following metrics: number of participant cores, average length, frequency and persistence. We observed potential non-SC behavior in 6 out of 8 benchmarks in this study. For the remaining two, we were not able to locate potential non-SC behavior in the parts of the applications sampled (and running the entire parallel region in cycle-accurate mode is infeasible). The statistics collected are shown in Table 3.

Number of Participants, Average Length and Frequency. For all benchmarks, we observe that, apart from FMM, cycles are formed by only two participants. FMM largely follows this trend, with 87.5% of the cycles formed by two participants and 12.5% formed by 3. The second column of Table 3 reports the average number of application events included in the cycle. We observe that cycles tend to be short, averaging no more than 230 application events in total, when considering all the participants of the cycle. We also did not observe any cycle larger than 244 events. The third column of Table 3 reports the frequency of cycles we observed in our experiments, as the number of cycle occurrences per one million instructions collectively executed by all application threads. We observe that the frequency of cycles is low across all benchmarks.

Persistence. We also studied the persistence of potential non-SC behaviors by altering dynamic executions and observing if the behaviors recur. For each benchmark, we randomly select an execution sample that exhibits one or more cycles and re-run it 200 times, perturbing the execution in a different

way every time. Specifically, a perturbation corresponds to a wasted cycle of execution, where a core is not allowed to commit any instructions, with that happening every N processor cycles. We force each core to waste different distinct cycles of execution, and repeat this experiment for different values of N [2]. The fourth column of Table 3 presents the number of runs (out of the 200 total) exhibiting one or more cycles, per benchmark.

We observe that all six benchmarks exhibit potential non-SC behavior in at least one additional run out of the 200 experiments, and often in a significant fraction of the 200 runs. FMM and Ocean exhibit a high probability of recurring potential non-SC behavior, while Canneal, Fluidanimate and Swaptions also exhibit a reasonably high probability. Streamcluster is the exception, as only 2 of the 200 perturbations caused cycles.

Implications. Based on the frequency and persistence metrics, we can conclude that cycles are prevalent and persistent in most of the benchmarks, and hence prior monitoring frameworks that deadlock when cycles arise are ineffective for these benchmarks (no debugging past the first deadlock). Based on the participants and average length metrics, we can conclude that the cycles are transient events that involve few participants racing over some shared data within a very short window of time. This has positive implications on the performance of our solution, as discussed in Section 7.3.

7.2. Sources of Cycles

We further investigated the conditions allowing cycles to be formed, by examining the code paths of the participating application threads involved in the race, along with the data structures accessed at that point. Out of all benchmarks that experienced cycles of dependences, Swaptions had occurrences from application code as well as library code (frequent use of malloc/free). The rest of the benchmarks had cycles generated exclusively by application code.

Five out of six benchmarks have cycles generated due to false sharing of cache blocks. In Canneal, Fluidanimate and Swaptions thread-private structures were allocated so as the parts of different structures fall in the same cache block. For example, in Canneal different instances of the random number generator class share the same cache blocks. Similarly, in Fluidanimate the beginning and ending of different dynamically allocated tables (e.g., "cnumPars" and "cells2") fall in the same cache block. These behaviors arise either because the programmer did not consider these structures to be performance critical or because he did not realize the probability of this behavior taking place (lack of feedback). Ocean appears to have a similar behavior when each thread is processing boundary elements (in *copy_border()*), only in this case it is probably expected by the programmer. Streamcluster implements its own synchronization primitives (barrier), which makes threads access different barrier variables falling in the same cache block close in time. Note that there is no prior solution for efficiently handling even false sharing cycles—always-on byte-level coherence tracking is prohibitively expensive [39] (Resolve does byte-level analysis only in the presence of cycles.) Finally, the remaining benchmark, FMM, has reported data races, which under relaxed memory models introduce SC violations [23].

7.3. Performance Results

Cycle Resolution Cost. To estimate the cost of deadlock resolution, we implemented the algorithm presented in Section 4 as a stand-alone application and time it on real hardware while resolving several cycles of 100 application events each. The implementation of the algorithm is single-threaded although several steps would benefit from parallelism (e.g., Steps 2–4). On a 3.0GHz Core i5 processor, resolution of deadlocks caused by false sharing (i.e., the algorithm completes after Step 2) requires no more than 30K cycles. Deadlocks for which the entire algorithm executes require no more than 110K cycles.

We then evaluated the effect of resolving a cycle to the performance of the monitoring system. We run each application monitored by a dummy monitoring tool (e.g., NullGrind), while limiting the buffering space between an application thread and a monitor thread to 256 events. This organization exercises the worst case scenario as it exposes immediately to the rest of the system the stalling of the monitoring threads that participate in the cycle (i.e., other threads might end up waiting for the participating ones in the absence of useful work to do). Upon experiencing a cycle, the participating monitoring threads stall for a number of cycles. The buffering spaces are quickly exhausted putting back-pressure to the application, which also stalls. Figure 8 shows the effect of resolving cycles on application performance for three different costs ranging from 10K to 1M cycles. Performance is normalized over the ideal case of resolving the cycle instantaneously.

As expected, even for the unrealistic scenario of spending one million cycles to resolve a cycle, the average slowdown is no more than 3%, which can be considered negligible. This is expected because cycles are infrequent, and because only very few monitoring threads have to do extra work, while others are allowed to proceed uninterrupted.

Fast Metadata Synchronization. We evaluated the performance of the proposed mechanism by monitoring our benchmarks with AtomCheck and MemLeak, under 2 different configurations. The first configuration shows the performance impact of having a memory fence at the end of every handler that updates shared metadata (*Fences*). For MemLeak only a subset of event handlers update potentially shared metadata, namely handlers that process store instructions or function returns. For the rest of the instructions monitored (e.g., ALU), the metadata of the destination registers are private among different monitoring threads and require no further action. The second configuration shows the performance of our hardware mechanism from Section 5. For both configurations, we report the Instructions-per-Cycle metric (IPC) for each tool-benchmark combination (under RMO). We calculate IPC using the number of busy user-level instructions (i.e., instructions executed by the tool while processing application events) throughout the experiment.

Figure 8(b left) shows the performance of MemLeak for the two configurations. For Blackscholes and Water we observe the highest potential as our mechanism achieves 23.2%

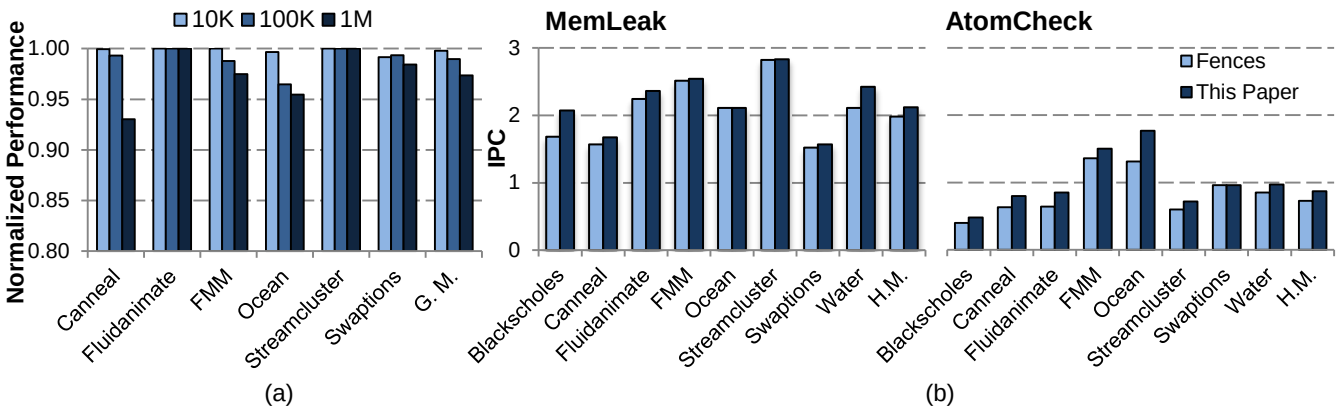


Figure 8: (a) Normalized application performance for different cycle resolution costs over ideal case (zero penalty). (b) Performance of MEMLEAK (left) and ATOMCHECK (right) expressed as *Busy* Instructions-per-Cycle for two configurations: *Fences* execute a memory fence at the end of every handler that updates shared metadata, and *This Paper* is our solution presented in Section 5.

and 14.7% higher IPC. For these two benchmarks, we observe a high percentage (close to 33%) of the total application events monitored to be store instructions, which translates to frequent stalls. On the other extreme, benchmarks that are highly compute intensive, such as FMM, Ocean and Streamcluster, experience no performance degradation, due to few updates to memory metadata. On average (rightmost bar), our mechanism achieves 6.9% higher IPC compared to *Fences*.

Figure 8(b right) shows the performance of ATOMCHECK for the two configurations. ATOMCHECK differs from MEMLEAK in two ways: it monitors only memory operations and it may update metadata state when processing a load instruction. We first observe that ATOMCHECK achieves a very low IPC compared to MEMLEAK. There are two fundamental reasons for that. First, updating metadata for application load events requires the monitoring threads to acquire exclusive access to metadata cache blocks that maintain state for application data that are potentially read-only. For several benchmarks we observed a large fraction of application load instructions to access global data, which caused metadata cache blocks to *ping-pong* among monitoring cores. Second, application load events that cause metadata state update follow the slow path, penalizing monitoring performance due to the synchronization overhead. These two overheads account for 50% and 40% of the total monitoring time for *Fences* and *Resolve*, respectively.

As far as the relative performance of the two configurations is concerned, we observe that over all benchmarks but Swaptions, our mechanism improves performance by 10% (FMM) to 35% (Ocean), and on average by 19%. For Swaptions specifically, the stall component due to frequent inter-thread dependences is high enough to hide any performance gains.

8. Related Work

This paper builds on top of prior work on hardware-based deterministic record and replay systems [39, 40, 27, 22, 15, 28, 31], which propose observing coherence activity to infer ordering between conflicting instructions. The Resolve techniques may also be incorporated back into much of this prior work to support relaxed memory models.

Prior work on software-only tools for monitoring parallel programs [17, 26, 11, 10, 29, 20] resorts to either *time-slicing* application execution on a single core or using high-level synchronization primitives, which incur high runtime overhead and also alter the behavior of the application, potentially masking bugs. Chung, *et al.* [7] overcome these limitations by coupling a binary instrumentation tool with software transactional memory, and enclosing in a transaction both the monitored and the monitoring code. Bobba, *et al.* [5] overcome the need for transaction-based execution by surrounding each monitored instruction with a load-store pair that access metadata, taking advantage of the limited set of instruction re-orderings allowed by a TSO memory model. However, this approach can not be extended to support any model weaker than TSO.

There have been several proposals of hardware-assisted tools for monitoring parallel applications, either targeting a specific checker [41, 33, 34], or targeting flexible instruction-grain monitoring supporting a variety of checkers [36, 18, 25]. All of them were designed for SC, and only a few [34, 36, 18] discuss the challenges of enabling parallel monitoring under relaxed memory models. None solve the monitoring-order and metadata-synchronization problems for relaxed memory models weaker than TSO.

9. Conclusion

The memory model of the system is one of the most important design parameters when building flexible hardware-assisted monitoring frameworks, having both correctness and performance implications. In this paper, we highlighted limitations of prior frameworks that caused them to deadlock and produce incorrect results under relaxed memory models, and presented *Resolve*, the first monitoring framework that overcomes these limitations. Resolve uses novel, low-overhead techniques for reasoning about ordering of application events in the presence of cyclical dependences and for ensuring access to shared metadata is properly synchronized. Finally, we present a quantitative characterization of cyclical dependences for parallel benchmarks running under RMO, demonstrating that cycles are prevalent and persistent, yet not too frequent or complex, so that Resolve's overheads are negligible.

References

[1] Sarita V. Adve and Kourosh Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, 1996.

[2] Alaa R. Alameldeen and David A. Wood, "Variability in architectural simulations of multi-threaded workloads," in *HPCA*, 2003.

[3] "ARMv7-M Architecture Reference Manual," ARM.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *PACT*, 2008.

[5] Jayaram Bobba, Marc Lupon, Mark D. Hill, and David A. Wood, "Safe and efficient supervised memory systems," in *HPCA*, 2011.

[6] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos, "Flexible hardware acceleration for instruction-grain program monitoring," in *ISCA*, 2008.

[7] Jaewoong Chung, Michael Dalton, Hari Kannan, and Christos Kozyrakis, "Thread-safe dynamic binary translation using transactional memory," in *HPCA*, 2008.

[8] Michael Dalton, Hari Kannan, and Christos Kozyrakis, "Raksha: a flexible information flow architecture for software security," in *ISCA*, 2007.

[9] Daniel Y. Deng, Daniel Lo, Greg Malysa, Skyler Schneider, and G. Edward Suh, "Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric," in *MICRO*, 2010.

[10] Cormac Flanagan and Stephen N. Freund, "FastTrack: efficient and precise dynamic race detection," in *PLDI*, 2009.

[11] ——, "The RoadRunner dynamic analysis framework for concurrent programs," in *PASTE*, 2010.

[12] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar, "Is SC + ILP = RC?" in *ISCA*, 1999.

[13] Michelle L. Goodstein, Shimin Chen, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry, "Chrysalis analysis: Incorporating synchronization arcs in dataflow-analysis-based parallel monitoring," in *PACT*, 2012.

[14] Michelle L. Goodstein, Evangelos Vlachos, Shimin Chen, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry, "Butterfly analysis: adapting dataflow analysis to dynamic parallel monitoring," in *ASPLOS*, 2010.

[15] Derek R. Hower and Mark D. Hill, "Rerun: Exploiting episodes for lightweight memory race recording," in *ISCA*, 2008.

[16] "Intel-64 and IA-32 Architectures Software Developer Manuals," Intel.

[17] Changhee Jung and Nathan Clark, "DDT: design and evaluation of a dynamic program analysis for optimizing data structure usage," in *MICRO*, 2009.

[18] Hari Kannan, "Ordering decoupled metadata accesses in multiprocessors," in *MICRO*, 2009.

[19] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou, "AVIO: detecting atomicity violations via access interleaving invariants," in *ASPLOS*, 2006.

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.

[21] Jonas Maebe and Michiel Ronsse, "Precise detection of memory leaks," in *International Workshop on Dynamic Analysis*, 2004.

[22] Pablo Montesinos, Luis Ceze, and Josep Torrellas, "DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently," in *ISCA*, 2008.

[23] Abdullah Muzahid, Shanxiang Qi, and Josep Torrellas, "Vulcan: Hardware support for detecting sequential consistency violations dynamically," in *MICRO'12*.

[24] Vijay Nagarajan and Rajiv Gupta, "Architectural support for shadow memory in multiprocessors," in *VEE*, 2009.

[25] ——, "ECMon: exposing cache events for monitoring," in *ISCA*, 2009.

[26] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for C," in *PLDI*, 2009.

[27] Satish Narayanasamy, Cristiano Pereira, and Brad Calder, "Recording shared memory dependencies using strata," in *ASPLOS*, 2006.

[28] Satish Narayanasamy, Gilles Pokam, and Brad Calder, "BugNet: Continuously recording program execution for deterministic replay debugging," in *ISCA*, 2005.

[29] Nicholas Nethercote and Julian Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *PLDI*, 2007.

[30] "The SPARC Architecture Manual," Oracle, 2000.

[31] Gilles Pokam, Cristiano Pereira, Klaus Danne, Rolf Kassa, and Ali-Reza Adl-Tabatabai, "Architecting a chunk-based memory race recorder in modern CMPs," in *MICRO*, 2009.

[32] F. Qin, C.Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu, "LIFT: A low-overhead practical information flow tracking system for detecting security attacks," in *MICRO*, 2006.

[33] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic, "FlexiTaint: A programmable accelerator for dynamic taint propagation," in *HPCA*, 2008.

[34] ——, "Memtracker: An accelerator for memory debugging and monitoring," *ACM TACO*, 2009.

[35] Virtutech Simics, http://www.virtutech.com/.

[36] Evangelos Vlachos, Michelle L. Goodstein, Michael A. Kozuch, Shimin Chen, Babak Falsafi, Phillip B. Gibbons, and Todd C. Mowry, "Paralog: enabling and accelerating online parallel monitoring of multi-threaded applications," in *ASPLOS*, 2010.

[37] T.F. Wenisch, R.E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J.C. Hoe, "Simflex: Statistical sampling of computer system simulation," *IEEE Micro*, 2006.

[38] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *ISCA*, 1995.

[39] Min Xu, Rastislav Bodik, and Mark D. Hill, "A 'Flight Data Recorder' for enabling full-system multiprocessor deterministic replay," in *ISCA*, 2003.

[40] ——, "A Regulated Transitive Reduction (RTR) for longer memory race recording," in *ASPLOS*, 2006.

[41] P. Zhou, R. Teodorescu, and Y. Zhou, "HARD: Hardware-assisted lockset-based race detection," in *HPCA*, 2007.