SuperCoP: A General, Correct, and Performance-efficient Supervised Memory System

Bharghava Rajaram, Vijay Nagarajan, Andrew J. McPherson and Marcelo Cintra
Institute for Computing Systems Architecture, University of Edinburgh
[r.bharghava, vijay.nagarajan, ajmcpherson]@ed.ac.uk , mc@staffmail.ed.ac.uk

ABSTRACT

Supervised memory systems maintain additional metadata for each memory address accessed by the program, to control and monitor accesses to the program data. Supervised systems find use in several applications including memory checking, synchronization, race detection, and transactional memory. Conventional memory instructions are replaced by supervised memory instructions (SMIs) which operate on both data and metadata atomically. Existing proposals for supervised memory systems assume sequential consistency. Recently, Bobba et al. [4] demonstrated the correctness issues (*imprecise exceptions* and *metadata read reordering*) in naively applying supervision to Total-Store-Order, and proposed two solutions – TSOall and TSOdata – for overcoming the correctness issues. TSOall solves correctness issues by forcing SMIs to perform in order, but performs similar to SC, since supervised writes cannot retire into the write-buffer. TSOdata, while allowing supervised writes to retire into the write-buffer, works correctly for only a subset of supervision schemes. In this paper we observe that correctness is ensured as long as SMIs read and process their metadata in order. We propose SuperCoP, a supervised memory system for relaxed memory models in which SMIs read and process metadata before retirement, while allowing data and metadata writes to retire into the write-buffer. Since SuperCoP separates metadata reads and their processing from the writes, we propose a simple mechanism – in the form of cache block level locking at the directory – to ensure atomicity. Our experimental results show that *SuperCoP* performs better than *TSOall* by 16.8%. SuperCoP also performs better than *TSOdata* by 6%, even though TSOdata is not general.

Categories and Subject Descriptors

C.0 [**General**]: System Architectures; D.2.5 [**Software Engineering**]: Testing and Debugging—*Monitors*

Keywords

Atomicity, Memory Ordering, Memory Trackers, Metadata, Supervised Memory

CF'12, May 15–17, 2012, Cagliari, Italy.

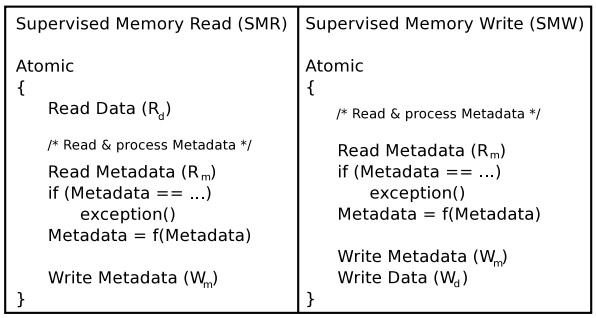

Figure 1: Supervised Memory Instructions

1. INTRODUCTION

In recent years, there has been renewed interest in memory systems which maintain additional data for each memory address accessed by the program. This additional data, or *metadata*, is used to store auxiliary information about the program memory, which is then used to control and monitor memory accesses issued by the program. Metadata is accessed and processed atomically with program data as shown in Figure 1. As we can see, each memory read (memory write) is associated with auxiliary memory operations which read metadata, process metadata (optionally) generating an exception, and (optionally) update metadata; furthermore, the entire sequence of data and metadata operations is performed atomically and is referred to as a supervised memory read - SMR (supervised memory write - SMW). Memory systems which support such supervised memory instructions (SMIs) are known as *supervised memory systems*. They serve as a foundation for important tasks such as enhancing security, reliability and programmability of applications – examples include memory trackers [2, 15, 17, 20, 21], transactional memory [3], fine-grained synchronization [22], and deterministic processing [7]. Supervised memory systems have become increasingly attractive with the emergence of multicore and manycore architectures which pose challenges in programmability and reliability.

(Correctness issues in supervised memory systems) Most current proposals for supervised memory systems, implicitly or explicitly, assume a sequentially consistent (SC) view of memory. Supervision can also be applied to Total-Store-Order (TSO) systems. While improving performance, supervision in TSO results in certain correctness issues. Bobba et al. [4] demonstrated the correctness issues in applying supervision to TSO consistency, resulting from retiring SMWs into the write buffer. An SMW that has retired into the write-buffer, performs its data and metadata op-

erations only when the SMW reaches the head of the write buffer. The resulting correctness issues are twofold. First, *metadata read reordering* – where an SMR may read and process its metadata, before an earlier SMW in the write buffer can read and process its metadata. Second, *imprecise exceptions* – where the exception raised by an SMW (if any) is imprecise. Since an SMW processes its metadata (and generates exceptions) only when it reaches the head of the write-buffer, subsequent instructions (which follow the SMW) may have already retired when the exception is raised.

Bobba et al. proposed a solution in the form of two systems: *TSOall* and *TSOdata*. TSOall solves the correctness issues by performing *all SMIs in order*. Since SMIs are performed in order, metadata reads of SMIs cannot be reordered; this also ensures that the exceptions raised by SMIs will be precise. On the other hand, an SMW cannot simply retire into the write-buffer, and will have to be fully performed before instructions that follow it can retire. *Thus TSOall, while solving the correctness issues, performs similar to SC on programs with frequent SMIs.* In contrast to TSOall, TSOdata allows an SMW to retire into the write buffer, thereby improving performance over TSOall. *TSOdata*, however, works correctly only for a subset of supervision schemes – it suffers from imprecise exceptions and metadata reordering which affects correctness in supervision schemes such as deterministic processing [7], full-empty bits [2], and DIFT [15]. *Thus,* TSOdata*, while performing better than TSOall, is not a generic solution to the correctness problem.*

(Our approach) While TSOall requires all SMIs to perform (as a whole) in order, we make the observation that *correctness is ensured as long as SMIs merely read and process their metadata in order*. In other words, we reduce the correctness requirement of supervised memory systems from SMI ordering to metadata read ordering. We propose *SuperCoP*, a supervised memory system in which SMIs read their metadata (data) and process them (generating an exception if necessary) before retirement; SuperCoP allows the resulting writes to metadata (and data) to be retired into the write-buffer. Since SuperCoP ensures correctness without making any assumptions about the supervision scheme, it is a generic solution to the correctness problem. At the same time, since SuperCoP allows data and metadata writes to retire into the write-buffer, it is efficient.

Since SuperCoP separates metadata reads (data reads) and their processing from the metadata writes (data writes), i.e, breaks the atomicity of the SMI, we provide additional mechanisms to ensure that the SMI appears atomic. For this purpose, we propose a simple and efficient atomicity scheme in which each SMI which modifies metadata locks the corresponding cache block in the directory before it retires; the lock is subsequently released when the corresponding metadata write is issued from the write-buffer; this ensures that any intervening SMI which tries to access this cache block is denied access until the metadata write of the original SMI completes. This, in turn, ensures that the original SMI appears atomic.

We compare the performance of SuperCoP, TSOdata, and TSOall, using the HARD [21] supervision scheme to test the different supervised systems. Our experiments show that SuperCoP performs 16.8% better than TSOall, and 6% better than TSOdata. It is worth noting that SuperCoP performs better than TSOdata, even thought TSOdata is not applicable to all supervised systems.

In summary, this work makes the following contributions to supervised memory systems.

- We reduce the correctness requirement of supervised memory systems to be metadata read ordering, rather than supervised instruction ordering. This correctness requirement enables the use of a write buffer, without compromising correctness.
- We propose SuperCoP a supervised memory system which reflects the above metadata read ordering requirement. Since SuperCoP separates reads and their processing from the writes, we propose a novel atomicity scheme to retain SMI atomicity.
- Since SuperCoP ensures correctness without making any assumptions about the supervision scheme, it is a generic solution to the correctness problem
- Our experiments show that SuperCoP is efficient, performing better than both TSOall (by 16.8%) and TSOdata (by 6%).

The rest of the paper is organized as follows. Section 2 provides some background information on supervised memory systems, and explains the working of a supervised system using the example of full-empty bits [2], including the issue of *metadata-data atomicity*. The correctness issues that manifest in supervised systems for TSO consistency are explained in section 3. We also provide our solution - SuperCoP - for these correctness issues in the same section. Our solution for the metadata-data atomicity issue is presented in section 4. Finally, section 5 presents the simulation results comparing the performance of SuperCoP, TSOall, and TSOdata.

2. BACKGROUND

2.1 Supervised Memory Systems

Supervised memory systems use metadata associated with each memory address to control and monitor access to those addresses. They make use of supervised memory instructions (SMI) to access and manipulate this metadata. The semantics of the metadata and the way it is used depends on the supervision scheme [1, 2, 3, 7, 15, 17, 20, 21, 22].

(Metadata, SMI) We use the full-empty bits supervision scheme [1, 2] throughout this paper in order to explain the working of supervised memory systems, and the issues associated with them. Full-empty bits is a supervision scheme which is typically used for word level producer-consumer synchronization. Here, each memory address is associated with a metabit (metadata with size 1 bit) which specifies whether the memory address is *full* (1) or *empty* (0). Processors make use of supervised memory writes (SMW) and supervised memory reads (SMR) to access data and metadata. A producer can write to the memory address only if the metabit is set to *empty*, and sets it to *full* once the write is complete. A consumer can read from a memory address only if the metabit is *full*, setting it to *empty* on completion. If an SMW encounters a *full* state, or if an SMR encounters an *empty* state, an exception is raised. An exception in the case of full-empty bits retries the memory access for a fixed number of times, and calls a trap handler if it still fails. The trap handler in turn decides to block the operation, retry the operation, or wake up the thread that is causing the exception to be raised in the first place.

(Metadata-data atomicity) One of the issues in supervised memory systems is that of metadata-data atomicity, which dictates that metadata operations should be atomic with respect to the corresponding data operations. Not observing this atomicity may lead to incorrect metadata values. Consider Figure 2, where both P_0 and P_2 perform an SMW to address A, while P_1 performs an SMR to the same address A. Assume that the initial metadata value for A is *empty*. If atomicity is preserved, the data and metadata operations

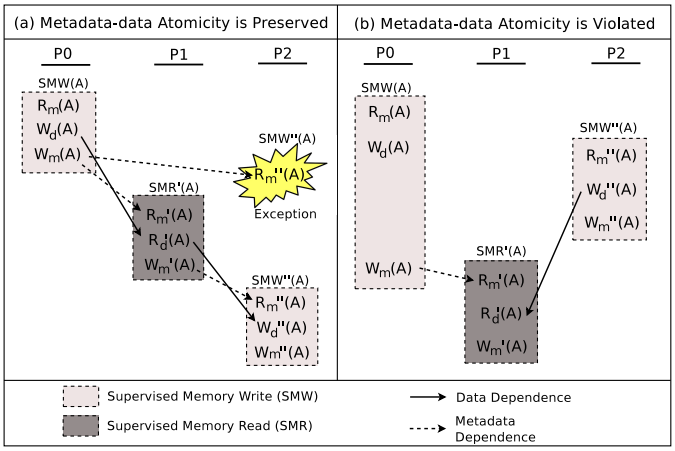

Figure 2: Metadata of A is initially *empty*. If metadata-data atomicity is preserved, SMR′(A) reads the value written by SMW(A). This is followed by SMW″(A), which updates A. *The exception raised by SMW″(A) prevents SMR′(A) from reading an incorrect value.* If metadata-data atomicity is violated, both SMW(A) and SMW″(A) are performed in an overlapped manner, and SMR′(A) ends up reading the data written by SMW″(A), instead of SMW(A).

of SMW(A) and SMW"(A) cannot interleave with each other. This is illustrated in Figure 2(a), where SMW(A) completes first, following which P_2 tries to perform SMW"(A). But, the metadata of A at this point in time will be *full* which causes SMW"(A) to raise an exception. SMR'(A) from P_1, however, can be performed and reads the value written by SMW(A). Following this, SMW"(A) is allowed to perform as SMR'(A) would have restored the metadata state to *empty*. Thus, preserving atomicity results in an execution pattern where SMR'(A) reads the value written by SMW(A), following which SMW"(A) updates the data in address A and the final metadata state of A is *full*. If atomicity is not preserved, as shown in Fig. 2(b), both SMW(A) (from P_0) and SMW"(A) (from P_2) can potentially interleave with each other. Indeed, the figure shows the scenario where SMW"(A), which performs after SMW(A), does not see the metadata update of SMW(A) (*full*) and thus proceeds without any exceptions being raised. This causes SMR'(A) to read the value written by SMW"(A) (as opposed to SMW'(A)) and the final metadata state of A is *empty* (as opposed to *full*). This sequence is incorrect as it violates the full-empty bits supervision scheme by allowing two consecutive writes to a memory location.

2.2 Types of Supervised Memory Systems

Supervised memory systems can be software based, or hardware assisted – the two differ in how SMIs are performed, the way in which memory space is allocated for metadata, and the way in which metadata-data atomicity is ensured.

(Software based supervised memory systems) In software supervised systems[11, 13, 12], SMIs are executed along with program instructions using the same processor pipeline i.e. metadata read, its processing and metadata write are all performed as separate software instructions. Some software based supervised memory systems track the order of data coherence requests and mirror this order for metadata as well. This is called coupled coherence or shadow coherence [10]. Other atomicity schemes include the use of transactional memory [6], where supervised instructions occur

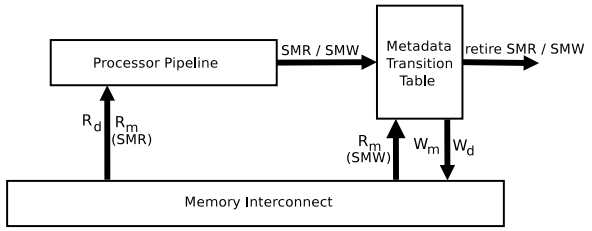

Figure 3: Memtracker implementation of a supervised memory system. An SMI (SMR/SMW) can be retired only after its metadata and data operations are completed.

as part of transactions which are either committed or re-executed depending on whether metadata-data atomicity is intact or is violated. The fact that software supervised systems execute additional instructions to operate on metadata, results in a heavy performance overhead which sometimes exceeds 100% [11].

(Decoupled supervised memory systems) In decoupled systems described in [5, 8, 18], the application program and metadata processing are performed in separate processors, called *application core* and *metadata core* respectively. The application core feeds a stream of committed instructions to the metadata core, which then performs the metadata operations for those instructions. Decoupled systems are similar to software based supervision in the way SMIs are performed, metadata storage is allocated, and how atomicity is ensured. Decoupled systems, however, require one metadata core for every application core to provide the best performance [18].

(Hardware assisted supervised memory systems) The performance overhead of software based supervision and the fact that decoupled systems require an additional core, for every application core, to process metadata has led researchers to adopt hardware assisted supervised memory systems. We concentrate on such *hardware assisted supervised memory systems* [14, 15, 16, 17, 21], where SMIs are performed entirely in hardware by modifying the processor pipeline or adding extra hardware inside the processor itself. We consider the case of **Memtracker**[17], which is the state-of-the-art hardware assisted supervised memory system. Memtracker performs the metadata operations after the *commit* stage of the pipeline, as shown in Figure 3. *Here, the memory system is a tagged memory system, where the data width of each address is extended to store the metadata along with program data.* Thus, a read operation to the data address also fetches the metadata. Similarly, a data write operation can also write the metadata during the same access. An SMR instruction reads its data and metadata as part of the processor pipeline. Once the instruction is ready to retire, the *metadata transition table* operates on this metadata and generates an updated metadata value according to the supervision scheme. For an SMW instruction, the metadata read is performed after the instruction is ready to retire. Both data and metadata are written back once the metadata processing is complete. An instruction is retired only when both data and metadata operations associated with it have completed. This implementation does not incorporate a write buffer, which implies that all instructions are performed **in-order**. Thus Memtracker performs like an SC system.

For an SMI to be atomic in Memtracker, its metadata write should be atomic with respect to its metadata read. Memtracker uses load replay to ensure metadata read write atomicity. If the metadata value read by an SMI is modified by another processor before the write completes, the metadata read is replayed. This is typically done by observing the coherence requests from other processors.

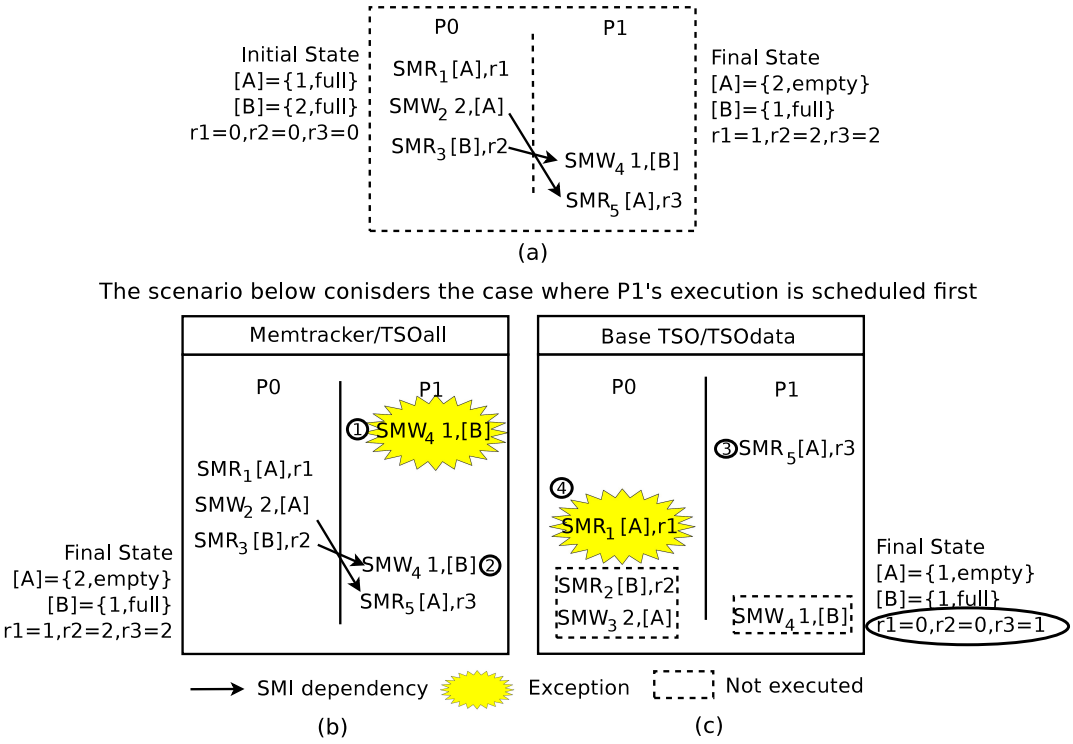

Figure 5: (a) shows the intended sequence of execution. (b) and (c) show the execution sequence in Memtracker and the base TSO model, when P_1's execution is scheduled before P_0. In Memtracker/TSOall, SMW_4(B) results in an exception ① blocking the thread until SMW_3(B) is performed. This results in SMR_5(A) reading the value written by SMW_2(A) ②. In the base TSO model/TSOdata SMR_5(A) is ordered before SMW_4(B) and reads the initial value of A ③, which is incorrect. Also, SMR_1(A) raises an unnecessary exception ④

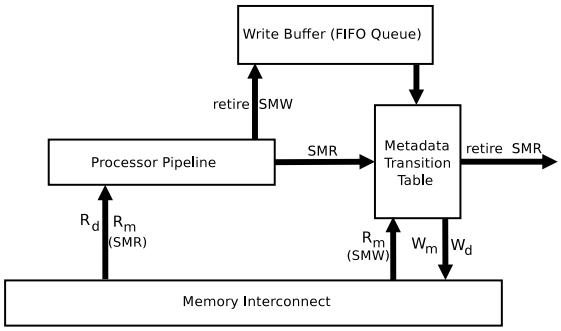

Figure 4: Base model for a TSO based supervised memory system. An SMR is retired after it has completed its data and metadata operations. An SMW is retired into the write buffer, and performs its data and metadata operations on reaching the head of the write buffer.

3. CORRECTNESS ISSUES IN TSO BASED SUPERVISED MEMORY SYSTEMS

(Base model) The TSO consistency model is widely used in present day systems, including Sun's SPARC, Intel's/AMD's x86 and its variants. The base system model considered in *Safe Supervised Memory* [4] is a TSO-based system, with a supervision mechanism similar to Memtracker [17]. The Memtracker proposal itself does not make use of a write buffer. The base TSO model for supervised systems, however, includes a write buffer into which all SMW instructions are retired. This results in an implementation where an SMW instruction reads its metadata only when it is issued from the write buffer i.e. when it reaches the head of the write buffer. Then, the metadata is processed and the resulting metadata (if any) and data are written back to memory. SMR instructions are processed in the same manner as in Memtracker. The resulting system model is described in Figure 4. Metadata-data atomicity is preserved in the same way as in Memtracker.

(Correctness issues) Bobba et al. [4] pointed out that retiring SMW instructions into the write buffer can causes correctness issues in the supervision scheme. It is possible that an SMR (that follows an SMW) can read its metadata before the preceding SMW in the write buffer can read its corresponding metadata. This is called *metadata read reordering* and can cause incorrectness in the supervision scheme. Furthermore, any exception caused by an SMW in the write buffer will not be raised until the SMW reaches the head of the write buffer. Thus the exception raised will be *late* or *imprecise*, since subsequent instructions (which follow the SMW) may have already retired when the exception is raised, again causing incorrectness in the supervision scheme. Since non-supervised writes do not read metadata

We use the full-empty bits supervision scheme to illustrate both these correctness issues. Consider the sequence of instructions shown in Figure 5(a). Here, SMR_3(B) (from P_0) synchronizes with SMW_4(B) (from P_1), to ensure that SMR_5(A) (from P_1) reads the

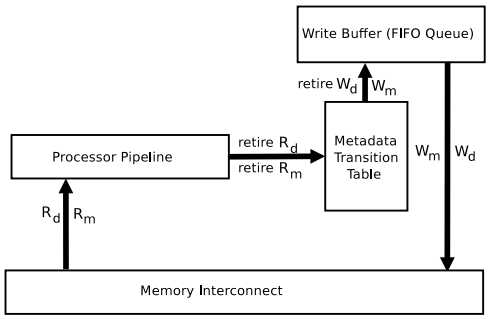

Figure 6: Implementation of SuperCoP. All SMIs are separated into their constituent operations and are retired separately. For an SMR, R_m/R_d are retired once they are completed, and W_m is retired into the write buffer. For an SMW, R_m is retired once the read is complete. W_d/W_m are retired into the write buffer.

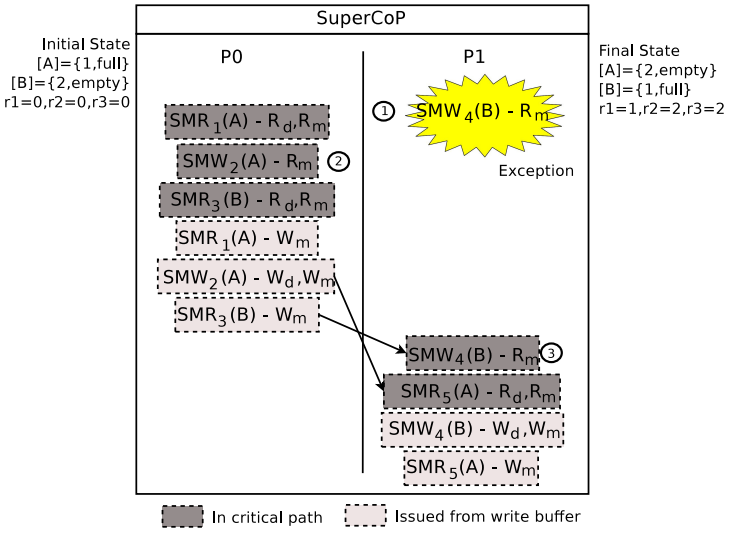

Figure 7: Resolving correctness issues in SuperCoP. If P_0 is scheduled before P_1, SMW_4(B) performs its R_m first, which raises an exception ①. The thread is then blocked until this R_m can succeed. In P_0, SMR_1(A) retires its metadata write to the write buffer, from which SMW_2(A) reads its metadata ②. SMW_4(B) can finally perform after SMR_3(B) completes its metadata write ③. Now, when SMR_5(A) is performed, it reads the value written by SMW_2(A). Thus, there is no incorrectness.

value written by SMW_2(A) (from P_0). Initially, both A and B are in the *full* state. In the expected execution sequence, P_0 performs SMR_1(A) reading the initial value (1) into r_1. SMR_1(A) also sets the metadata of A to *empty*. Then SMW_2(A) writes the value 2 into A, reverting its metadata to *full*. P_0 then performs SMR_3(B) which *empties* address B. This is followed by P_1 writing into B (SMW_4), and then reading from A (SMR_5). No exceptions are raised in this execution sequence. *The result of the execution is that SMR_5(A) from P_1 reads the value written by SMW_2(A) i.e. r_3=2.*

Now, it is possible that P_0 is stalled or blocked, resulting in P_1 executing its instructions first. In such a case, P_1 first tries to perform SMW_4(B). In Memtracker (Figure 5(b)), this immediately raises an exception ① and P_1 is blocked until this instruction can successfully execute. P_1 can resume its execution ② only after SMR_3(B) completes i.e SMR_3(B) updates the metadata of B to *empty*. Once P_1 resumes, it performs SMW_4(B) followed by SMR_5(A). *This results in SMR_5(A) reading the value written by SMW_2(A), as in the expected execution sequence.*

In the base TSO model (Figure 5(c)), however, P_1 retires SMW_4(B) into the write buffer, and proceeds to perform SMR_5(A) ③. Since SMR_5(A) reads a metadata state of *full*, it does not raise any exception and reads the value 1 into r_3 (setting metadata of A to *empty*). The exception on writing to B (which is in *full* state) is raised only when SMW_4(B) is issued from the write buffer. This *imprecise exception* combined with the *metadata read reordering* that occurs when SMR_5(A) reads its metadata before SMW_4(B), *results in an incorrect value being read into r_3*. Also, with this execution sequence, when P_0 eventually performs SMR_1(A), it reads a metadata value of *empty* resulting in an exception ④. Subsequent instructions are not performed till SMR_1(A) is successful, which does not happen within this execution sequence.

Bobba et al. outline two systems, namely TSOall and TSOdata, to tackle these correctness issues. TSOall is the same as the Memtracker system model, with the only difference being that TSOall allows unsupervised write instructions to retire into the write buffer. Thus, TSOall does not suffer from any correctness issues, but has a high performance overhead as the write buffer is not used efficiently. TSOdata is the same as the base TSO model explained in Figure 4. Thus all correctness issues that afflict the base TSO model also manifest in TSOdata. Bobba et al. prescribe TSOdata, however, only for supervision schemes that tolerate metadata reordering, like HARD [21]. Unfortunately, not all supervision schemes tolerate metadata reordering, including full-empty bits. In summary, TSOall is correct but inefficient, and TSOdata is not applicable to all supervision schemes, even though it is efficient. We propose a general solution to the correctness issues in TSO based supervised systems without compromising on performance.

(Our Approach) Since Memtracker/TSOall does not suffer from correctness issues, we can conclude that inorder execution of SMIs is a sufficient condition to solve imprecise exceptions in TSO based supervised systems. This *inorder* execution of SMIs, however, results in a high performance overhead. We observe that inorder execution of SMIs is not a necessary condition for correctness. Intuitively, the necessary condition to avoid metadata read reordering is to guarantee that metadata reads (and processing) occur in program order. This will also ensure that metadata reads of SMW instructions will be performed before subsequent instructions retire. Thus, by enforcing metadata read ordering, both correctness issues that plague TSO based supervised memory systems can be solved. We, therefore, reduce the correctness requirement of a supervised system. The proposed correctness requirement is that metadata reads should be performed inorder, rather than entire SMIs being performed inorder. Since only metadata reads are to be performed inorder, metadata and data writes can be retired into the write buffer, which reduces the performance overhead as compared to Memtracker. Hence we propose SuperCoP - Supervision with Correctness and Performance, which ensures metadata read ordering by separating metadata reads from metadata writes.

The implementation of SuperCoP is illustrated in Figure 6. In SuperCoP, an SMR performs its data read (R_d) and metadata read (R_m) as part of the processor pipeline. Both read operations retire once they are completed. Then, the metadata is processed and the resultant metadata write (W_m), if any, is retired into the write buffer. Then the processor continues with its execution. This removes W_m from the critical path. An SMW also performs its R_m

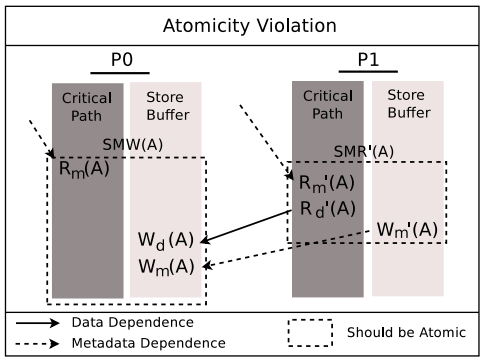

Figure 8: Atomicity Violation in SuperCoP. R_m (from P_0) retires and inserts W_m and W_d into the write buffer. Meanwhile, R'_m (from P_1) is performed and reads the same metadata value as R_m, instead of reading the metadata value written by W_m, as W_d is performed after R'_d.

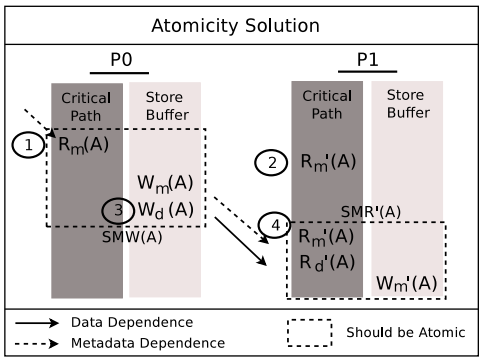

Figure 9: Solving Atomicity by fine grain locking. R_m locks address A ① prohibiting R'_m from being performed ②. W_m unlocks A ③ following which R'_m is performed successfully ④. Thus, R'_m correctly reads the metadata written by W_m.

as part of the pipeline, once address computation is complete. The metadata is then processed following which R_m retires. Then, both data write (W_d) and metadata write (W_m) are retired into the write buffer. As we can see, all metadata reads are performed in order, and instructions need only to wait for the metadata read and processing to be complete for them to be retired. Unlike TSOdata, we do not make any assumptions about the supervision scheme itself. This makes SuperCoP applicable to all supervised memory systems.

Consider the same sequence of instructions as in Figure 5. The execution order as per SuperCoP is shown in Figure 7. Assuming the same scenario (where P_0 has been stalled or blocked), P_1 begins with the execution of $SMW_4(B)$. Since SuperCoP performs metadata read of an SMW in the critical path, an exception is raised ① as the SMW is to a *full* location. This stalls P_1 until the metadata of B becomes *empty*. Now when P_0 eventually begins execution, it performs the metadata and data read for $SMR_1(A)$, and retires the metadata write to the write buffer. The next SMI being an SMW to A can read its metadata from the write buffer itself through a read bypass ②. The resulting metadata write and data write are retired into the write buffer. This is followed by $SMR_3(B)$. Once $SMR_3(B)$ completes its metadata write and updates the metadata of B to *empty*, P_1 is unblocked and proceeds with its execution ③. It is evident that SuperCoP's execution order is the same as Memtracker/TSOall. *It is worth noting that $SMR_5(A)$ in P_1 reads the value written by $SMW_2(A)$ which is the expected result.* This example shows how SuperCoP deals with the correctness issues that manifest in TSO based supervised systems.

4. ENSURING ATOMICITY

A consequence of separating metadata reads and writes in SuperCoP is that it can violate metadata-data atomicity which in turn leads to incorrectness in metadata. An example of this metadata-data atomicity violation is illustrated in Figure 8. For ease of explanation, we use a generic supervision scheme instead of the full-empty bits scenario.

Here, P_0 performs SMW(A), and P_1 performs SMR′(A). First, P_0 performs and retires $R_m(A)$ in the critical path. Once the metadata processing is done, P_0 retires $W_m(A)$ and $W_d(A)$ into the write buffer, which may have other entries above it. Meanwhile, SMR′(A) is performed in P_1. $R'_m(A)$ reads the same metadata value read by $R_m(A)$. The metadata is processed and $W'_d(A)$ is retired into the write buffer and is immediately issued to the memory. $W_d(A)$-$W_m(A)$ is then issued to memory. Here, even though $W_m(A)$ is ordered after $R'_d(A)$, $R_m(A)$ does not read the metadata value written by $W'_m(A)$. This violates metadata-data atomicity.

(Atomicity based on fine grain locking) To ensure atomicity, either SMR′(A) should be allowed to perform only after SMW(A) completes, or SMW(A) should be re-executed after SMR′(A) completes (similar to Memtracker). In case of the latter, R_m cannot be allowed to retire until its metadata write has completed. Since this obviates any performance gain in separating metadata read and write, we choose the former approach to implement atomicity. We ensure that an SMI can be performed in an uninterrupted fashion, by using a fine grain locking mechanism to lock the address accessed by an SMI when the metadata read is performed, and relinquish the lock when the metadata write is performed. All coherence requests to locked addresses will be denied/delayed until the unlock operation occurs. This will guarantee that no other SMI can access the metadata location locked by another SMI until it completes. An execution pattern for a generic fine grain locking based atomicity scheme is illustrated in Figure 9. ① First, address A is locked by $R_m(A)$. ② Now, when $R'_m(A)$ is issued, its coherence request is denied owing to the lock on A. ③ The lock on A is relinquished when $W_m(A)$ is issued to memory. ④ Now, when $R'_m(A)$ is issued to memory, it locks A. Here, SMR′(A) reads both its data and metadata from SMW(A), thus preserving metadata-data atomicity. A is later unlocked when $W'_m(A)$ completes in the memory.

(Atomicity using local cache locking) In order to implement fine grain locking, all SMIs can be considered on the lines of conventional Read-Modify-Write (RMW) instructions. Conventional RMW instructions obtain write permissions to the cache block they address, and lock the cache block in the local cache. Similarly, an SMI instruction should obtain exclusive permissions to the cache block it addresses, lock the cache block locally, and then retire its write(s) into the write buffer. If a metadata write is not generated, the cache block is unlocked immediately. Otherwise, the lock is released when the metadata write is issued from the write buffer. Referring back to Figure 9, in ①, $R_m(A)$ gets write permissions for A and locks the address in its local cache. $R'_m(A)$'s request for A is denied ②, until the lock is relinquished ③. Then $R'_m(A)$ obtains write permissions for A by invalidating the copy in P_0. An SMI performs invalidations only when a metadata write is gener-

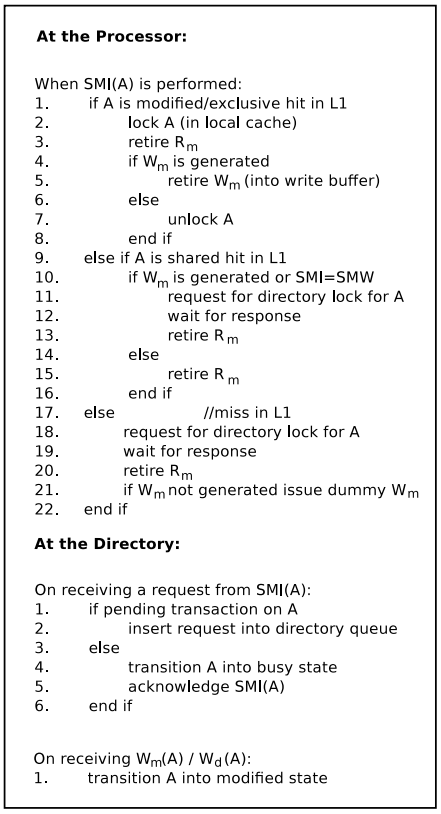

Figure 10: Protocol for the proposed Directory locking mechanism to preserve metadata-data atomicity.

ated. The invalidation, however, for both SMRs and SMWs occur in the critical path, owing to which the cache locking scheme will suffer a performance overheard similar to that of TSOall.

(Atomicity using directory locking) To address this drawback, we propose a novel atomicity scheme which reduces the number of invalidations and pushes the remaining invalidations out of the critical path. We make use of the underlying coherence protocol to implement this atomicity scheme, which in our case is the directory protocol. Invalidations occur when the metadata address is in the *shared* coherence state. If an SMI is issued to a *shared* address, then the cache block it addresses is locked in the directory instead of obtaining write permissions to it and locking it in the local cache. The lock is relinquished on completion of the SMI's metadata/data write. For an SMR, invalidations are carried out only if a metadata write is generated, thereby reducing the number of invalidations as compared to the local cache locking scheme. For both SMW/SMR, invalidations in the critical path are replaced by a directory access which is much cheaper. The invalidation itself is removed from the critical path and is performed as part of the write buffer logic.

The protocol followed for the directory locking mechanism is outlined in Figure 10. Let us assume an SMI to address *A*. If *A* is in modified/exclusive state in the local cache, the locking happens in the local cache (L1) itself. The corresponding metadata write unlocks the cache block. If no metadata write is generated, then the cache block is unblocked immediately. Like in local cache locking, all requests to a locked cache block are denied by the processor.

The request is forwarded to the directory if a) *A* is not present in L1, or b) *A* is in *shared* state and the SMI is a read which generates a metadata write, or c) *A* is in *shared* state and the SMI is a write. The directory checks if there are any pending requests to *A*. If there are pending requests to *A* in the directory, the request is inserted into the directory queue and is serviced when there are no other requests to *A* ahead of it in the queue. When the request gets serviced the cache block is transitioned into a *busy* state, and an acknowledgement is sent to the requesting processor. The processor retires the read operation on receiving a response from the directory. If the directory receives a coherence request to a cache block in *busy* state, the request is queued in the directory. The cache block is transitioned out of the *busy* state when the directory receives a corresponding metadata write/data write. Thus, the *busy* state acts like a lock. If an SMI which has obtained a directory lock does not generate a metadata write (a miss in the local cache), a *dummy* write has to be issued to unlock the address in the directory. It is worth noting that this *dummy* write need not be issued in the critical path.

(Example) The working of the directory locking based atomicity scheme is illustrated in Figure 11, where P_0 performs SMW(A) and P_1 performs SMR′(A) (as shown in Figure 9). Assume that *A* is initially shared between P_0 and P_1. First, SMW(A) (from P_0) performs its metadata read. ① Since, *A* is in *shared* state, the request is forwarded to the directory and ② locks *A* (goes to *busy* state) in the directory. ③ The directory responds to the request so that R_m can be retired. ④ The metadata is then processed (in P_0) and the resulting metadata write is retired into the write buffer. It is worth noting that *A* is still in the *shared* state in the directory. ⑤ Now, when SMR′(A) (from P_1) sends a read request to the directory, ⑥ it is inserted into the directory queue, as *A* is in *busy* state. The metadata read of SMR′(A) is not retired as it does not receive a response from the directory. ⑦ When W_m is issued from P_0's write buffer, ⑧ P_1's copy of *A* is invalidated, and ⑨ the lock on *A* is relinquished by P_0. The directory transitions the cache block *A* to modified state owned by P_0. Now, ⑩ SMR′(A)'s request is serviced by the directory, which ⑪ (& ⑫) obtains the updated copy of the data and metadata from P_0 and ⑬ locks *A* again, by transitioning it to the *busy* state. ⑭ The directory acknowledges SMR′(A), so that its metadata and data read can be retired. *A* is now again in *shared* state with both P_0 and P_1 having copies of it. ⑮ Eventually, SMR′(A) issues its metadata write to unlock *A* in the directory.

5. EXPERIMENTAL RESULTS

(System Specification) We built a hardware simulation infrastructure using the PIN tool [9], to simulate 16 processors connected in a mesh network. The interconnect has a link latency of 1 cycle and router latency of 4 cycles. Each processor has a 32-entry write buffer, and a private 4-way 32KB L1. A MESI-based directory protocol is used to keep all L1 caches coherent. The L2 cache (16-way 1MB/core) and the directory are static address interleaved. Each instruction takes 1 cycle to execute, and it takes a total of 4, 20, and 200 cycles to access the L1, L2, and main memory, respectively. As mentioned in previous sections, we use a *tagged* memory system to store metadata for the supervised system, where both metadata and data are stored together in the same address. All memory operations are considered as supervised memory operations. We evaluate the performance of TSOall, TSOdata, and SuperCoP using the SPLASH-2 [19] benchmark suite. The benchmarks and their respective input sizes are listed in Table 1. We evaluate SuperCoP with both cache based locking and directory based locking.

(HARD supervision scheme) We demonstrate the efficacy of SuperCoP as compared to TSOall and TSOdata using the HARD supervision scheme proposed by Zhou et al. [21]. HARD is used

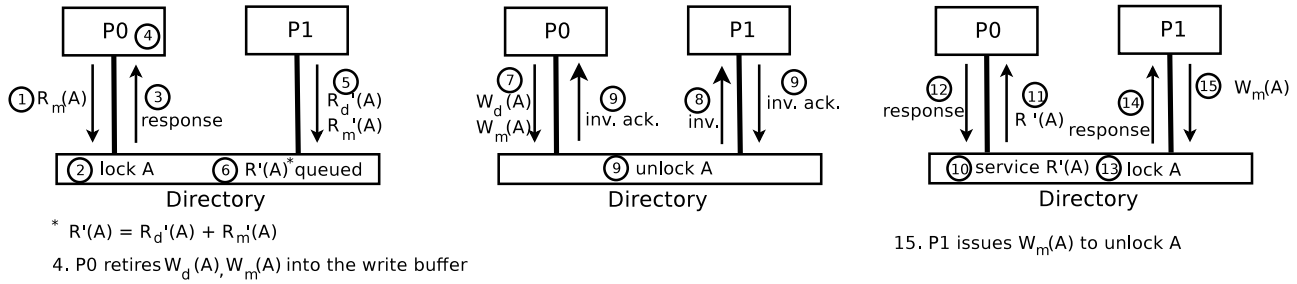

Figure 11: Metadata-data atomicity with Directory locking. Here, P_0 performs SMW(A) and P_1 performs SMR′(A). The sequence of events are numbered in the ascending order. The figure on the left shows how SMW(A) locks address A, and SMR'(A) is queued in the directory. The figure in the center shows steps involved in unlocking A. In the figure on the left, SMR′(A) locks A, when it is serviced by the directory. Eventually, SMR′(A) issues its metadata write to unlock A.

Table 1: Splash-2 Benchmark Suite

Code	Problem Size
Barnes	16K particles
Cholesky	tk29.O
FFT	64K points
FMM	16K particles
LU (contiguous)	512x512 matrix, 16x16 blocks
LU (non-contiguous)	512x512 matrix, 16x16 blocks
Ocean (contiguous)	258 x 258 ocean
Ocean (non-contiguous)	258 x 258 ocean
Radiosity	room, -ae 5000.0 -en 0.050 -bf 0.10
Radix	1M integers, radix 1024
Raytrace	car
Volrend	head
Water-Nsq	512 molecules
Water-Sp	512 molecules

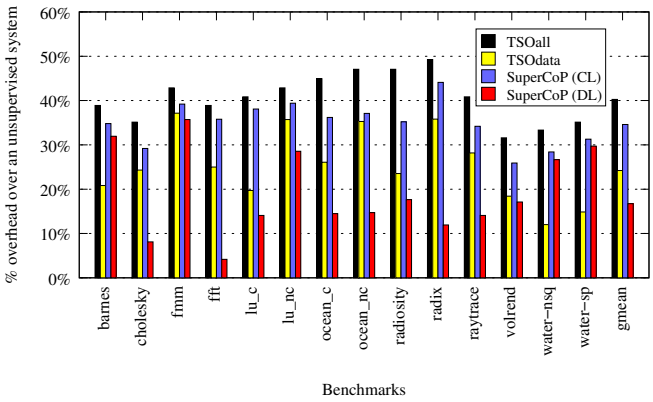

Figure 12: Performance comparison for the HARD supervision scheme. CL and DL represent the cache locking scheme and directory locking schemes, respectively.

for race detection in multi-threaded software. It ensures that all accesses to a shared variable are protected by at least one common lock. Each thread maintains a variable called LockSet which is the union of all the locks currently held by the thread. Each variable is protected by a Candidate Set which is the set of locks used to protect the variable thus far. On every memory access, the candidate set is updated to include the LockSet of the thread reading the variable. Candidate sets are written at cache-block granularity, and form the metadata in this system along with the state of the variable. A simple finite state machine is used to transition variable states, that initializes blocks in private states and transitions them to a shared state when they are accessed by multiple threads. On every data access, the LockSet, the CandidateSet, and the variable state are used to detect a race, and an exception is raised when a certain set of conditions are met. HARD uses Bloom filters to efficiently represent them in hardware. We chose the HARD supervision to compare the various supervised systems as it is an example of a supervision scheme which reads processes and updates metadata. Also, the earlier work by Bobba et al. [4] uses HARD to compare TSOdata and TSOall.

(Simulation results for HARD) We compare the performance of TSOall, TSOdata, SuperCoP (with cache locking), and SuperCoP (with directory locking) for the HARD supervision scheme. The experimental results are shown in Figure 12. The percentage of SMIs which update metadata varies from 0.2% (*fmm*) to 57.8% (*lu-contiguous*) as shown in Table 2, with an average (geometric mean) of 4.5%. We observe from Figure 12 that TSOdata consistently performs better than TSOall as it retires SMWs to the write buffer, while incurring the same latency for SMRs. On an average, TSOdata performs better than TSOall by 11.4%. With cache locking, SuperCoP performs worse than TSOdata for all benchmarks (average of 8.3%), as invalidations for SMRs and SMWs which update shared metadata are in the critical path. With directory locking, however, SuperCoP outperforms TSOdata by 6% across all benchmarks.

It is worth nothing that, TSOall performs invalidations for both SMRs which update metadata and SMWs in the critical path. TSOdata performs invalidations for SMRs which update metadata in the critical path, while SMWs are completely performed in the write buffer. With the cache locking scheme, SuperCoP performs invalidations for all SMIs which update shared metadata in the critical path. Since, invalidations of SMRs and SMWs are performed in the critical path, the cache locking scheme incurs a penalty close to TSOall. With the directory locking scheme, only SMRs which update metadata to shared locations, and SMWs to shared locations incur a directory access in the critical path, thereby providing better performance than even TSOdata.

We analyze the directory locking scheme in more detail. We observe that SuperCoP with the directory locking scheme performs much better than TSOdata for benchmarks which have a higher percentage of SMIs that update metadata (*fft* - 39.5%, *lu-contiguous* - 57.8%, *lu-noncontiguous* - 48.8%, *ocean-contiguous* - 14.5%,

Table 2: Characteristics of Supervised Instructions for HARD

Code	% of SMIs updating metadata
Barnes	8.5
Cholesky	0.7
FFT	39.5
FMM	0.2
LU contiguous	57.8
LU noncontiguous	48.8
Ocean contiguous	14.5
Ocean noncontiguous	20.9
Radiosity	7.2
Radix	5.6
Raytrace	11.3
Volrend	0.2
Water-Nsq	6.2
Water-Sp	2.1

ocean-contiguous - 20.9%, *raytrace* - 11%). For *radix*, even though the percentage of SMIs which update metadata is comparatively less (5.5%), SuperCoP (with directory locking) performs much better than TSOdata as a larger percentage of SMIs which update metadata are shared SMRs (recall that shared SMRs which update metadata are more expensive in TSOdata than in SuperCoP) and the *number* of SMWs which update metadata to shared locations is negligible. Similarly, in *cholesky*, the *number* of SMRs updating metadata are much larger compared to SMWs which update shared metadata. SuperCoP (with directory locking) performs worse than TSOdata for *barnes, water-nsquared* and *water-spatial* as these applications issue a comparatively larger number of SMWs to shared locations. In case of *fmm*, even though the number of shared writes is large, the SMWs which update shared metadata is very few (0.06%) in number compared to the number of SMWs which update metadata in exclusive locations (21%). The overhead that these SMWs cause in SuperCoP is offset by the number of SMRs which update metadata, resulting in SuperCoP performing on par with TSOdata.

(Scalability with respect to metadata updates) It can be seen from the results for HARD that the percentage of SMIs that update metadata critically influences the performance of a supervised system. Thus, we can study the scalability of the supervised systems by implementing a generic supervision scheme, and varying the percentage of SMIs which update metadata. The cost associated with a metadata update depends on whether the SMI is an SMR or an SMW. Now, an SMR which updates metadata has a higher latency than an SMR which does not update metadata. This is evident from the implementations of TSOall and TSOdata where SMRs updating metadata result in an invalidation in the critical path. Also, in SuperCoP, an SMR updating shared metadata must access the directory. The cost of an SMW, however, depends on the coherence state rather than whether it updates metadata or not. Thus, the performance overhead of the supervision scheme increases proportionally with the percentage of SMRs updating metadata, which in turn depends on the supervision scheme. This means that scalability of a supervised system can be represented on the lines of *metadata update percentages*. In our experiments, we vary the percentage of SMIs which update metadata from 5% to 100% .

(Simulation results for scalability) Figure 13 shows how the performance of SuperCoP scales with percentage of SMIs updating metadata as compared with TSOdata. The execution time is averaged across all SPLASH-2 benchmarks and are normalized with respect to the execution time of TSOall. We have not represented

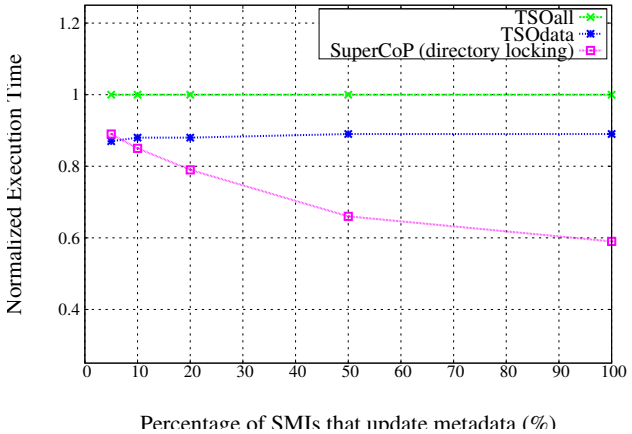

Figure 13: Scalability of supervised systems with respect to the number of SMIs which update metadata. The execution time is normalized to that of TSOall to better represent the scalability.

SuperCoP with the cache locking scheme as it is evident from the results of the HARD supervision scheme that the cache locking scheme performs as bad as TSOall.

Figure 13 shows that TSOdata provides the same performance improvement over TSOall as metadata update percentage increases. It is worth noting that the only performance improvement that TSOdata provides over TSOall is for SMWs, and even if these SMWs update metadata, performance improvement for TSOdata over TSOall will be the same. And SMRs are performed in the same manner for TSOall and TSOdata. Therefore, the only advantage that TSOdata provides over TSOall is the write buffer. In the case of SuperCoP, however, the latency of SMRs updating metadata is reduced as compared to TSOall and TSOdata. SMWs incur a small penalty of accessing the directory if it is to a shared location, which is constant across varying metadata update percentage. Thus, SuperCoP performs better than TSOall and TSOdata as the percentage of SMIs that update metadata increases.

6. CONCLUSION

Existing supervised memory systems, implicitly or explicitly, assume SC [4]. Bobba et al. proposed two systems; TSOall which has significant performance overhead, TSOdata which is not general and still suffers from correctness issues for certain supervision schemes. Bobba et al.'s work on safe supervised systems implicitly assumes that inorder execution of SMIs ensures correctness. We reduce this correctness requirement to *metadata read ordering*. To this end, we develop SuperCoP, which separates metadata reads and writes, and ensures *metadata read ordering* by performing the metadata reads in the critical path. We also propose a directory locking scheme to ensure metadata-data atomicity at a lower cost.

We demonstrate the efficiency of SuperCoP with respect to TSOall and TSOdata using the HARD supervision scheme. Our experimental results using HARD show that *SuperCoP* performs better than *TSOall* by 16.8% and *TSOdata* by 6%. We also analyze the scalability of supervised systems with respect to the percentage of SMIs which update metadata. It is evident from our experiments that SuperCoP scales better than TSOall or TSOdata. Thus we show that SuperCoP is a correct and performance efficient supervised memory system that is general, in that it is applicable to any supervision scheme.

7. ACKNOWLEDGEMENTS

We would like to thank the reviewers for their helpful comments. This work is supported by the Centre for Numerical Algorithms and Intelligent Software, funded by EPSRC grant EP/G036136/1 and the Scottish Funding Council.

8. REFERENCES

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The mit alewife machine: architecture and performance. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 2–13, New York, NY, USA, 1995. ACM.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *Proceedings of the 4th international conference on Supercomputing*, ICS '90, pages 1–6, New York, NY, USA, 1990. ACM.

[3] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 127–138, Washington, DC, USA, 2008. IEEE Computer Society.

[4] J. Bobba, M. Lupon, M. Hill, and D. Wood. Safe and efficient supervised memory systems. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 369 –380, feb. 2011.

[5] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, ASID '06, pages 63–65, New York, NY, USA, 2006. ACM.

[6] J. Chung, M. Dalton, H. Kannan, and C. Kozyrakis. Thread-safe dynamic binary translation using transactional memory. In *Proceedings of the 2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 279 –289, feb. 2008.

[7] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 85–96, New York, NY, USA, 2009. ACM.

[8] H. Kannan. Ordering decoupled metadata accesses in multiprocessors. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 381–390, New York, NY, USA, 2009. ACM.

[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[10] V. Nagarajan and R. Gupta. Architectural support for shadow memory in multiprocessors. In *Proceedings of the 5th international conference on Virtual execution environments*, VEE '09, pages 1–10, New York, NY, USA, 2009. ACM.

[11] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 65–74, New York, NY, USA, 2007. ACM.

[12] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[13] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS'05, 2005.

[14] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society.

[15] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, pages 85–96, New York, NY, USA, 2004. ACM.

[16] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Proceedings of the 2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 173–184, feb. 2008.

[17] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Memtracker: An accelerator for memory debugging and monitoring. *ACM Trans. Archit. Code Optim.*, 6:5:1–5:33, July 2009.

[18] E. Vlachos, M. L. Goodstein, M. A. Kozuch, S. Chen, B. Falsafi, P. B. Gibbons, and T. C. Mowry. Paralog: enabling and accelerating online parallel monitoring of multithreaded applications. In *Proceeding of the 15th international conference on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 271–284, New York, NY, USA, 2010. ACM.

[19] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.

[20] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iwatcher: Efficient architectural support for software debugging. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 224–235, Washington, DC, USA, 2004. IEEE Computer Society.

[21] P. Zhou, R. Teodorescu, and Y. Zhou. Hard: Hardware-assisted lockset-based race detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 121–132, feb. 2007.

[22] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 35–45, New York, NY, USA, 2007. ACM.