

# Safe and Efficient Supervised Memory Systems

Jayaram Bobba  
Intel Corporation  
jayaram.bobba@intel.com

Marc Lupon  
Universitat Politècnica de Catalunya  
mlupon@ac.upc.edu

Mark D. Hill and David A. Wood  
University of Wisconsin-Madison  
{markhill,david}@cs.wisc.edu

## Abstract

*Supervised Memory systems* use out-of-band metabits to control and monitor accesses to normal data memory for such purposes as transactional memory and memory typestate trackers. Previous proposals demonstrate the value of supervised memory systems, but have typically (1) assumed sequential consistency (while most deployed systems use weaker models), and (2) used *ad hoc*, informal memory specifications (that can be ambiguous and/or incorrect). This paper seeks to make many previous proposals more practical.

This paper builds a foundation for future supervised memory systems which (1) operate with the TSO and x86 memory models, and (2) are formally specified using two *supervised memory* models. The simpler TSO<sub>all</sub> model requires all metadata and data accesses to obey TSO, but precludes using store buffers for supervised accesses. The more complex TSO<sub>data</sub> model relaxes some ordering constraints (allowing store buffer use) but makes programmer reasoning more difficult.

To get the benefits of both models, we propose *Safe Supervision*, which asks programmers to avoid using metabits from one location to order accesses to another. Programmers that obey safe supervision can reason with the simpler semantics of TSO<sub>all</sub> while obtaining the higher performance of TSO<sub>data</sub>. Our approach is similar to how data-race-free programs can run on relaxed systems and yet appear sequentially consistent. Finally, we show that TSO<sub>data</sub> can (a) provide significant performance benefit (up to 22%) over TSO<sub>all</sub> and (b) can be incorporated correctly and with low overhead into the RTL of an industrial multi-core chip design (OpenSPARC T2).

## 1. Introduction

Over the past decades, there has been sustained interest in memory systems that dedicate a small portion of their transistor budget to a few ‘out-of-band’ bits per data block. These *metabits* enable software to efficiently maintain metadata corresponding to program data and use it to *supervise* (control and monitor) data operations. We refer to this broad class of systems as *supervised*

*memory systems* (*supervised systems*, in short). This work deals with enabling correct and efficient supervised systems.

Memory supervision forms the basis of many proposals: unbounded transactional memory [6,7,9], log-based architectures [11,12], deterministic shared memory [15], memory-typestate trackers [33,36,42,43], information-flow tracking [34] and hardware-assisted garbage collectors [13]. Supervised systems facilitate the development of safe and efficient software, and are increasingly attractive with the emergence of multi-core architectures and their programmability challenges.

Current supervised system proposals demonstrate exciting opportunities, but suffer from at least one of two drawbacks that make it harder to incorporate them into industrial systems.

- **SC-centricity.** Most existing proposals for supervised systems ignore memory consistency issues [7,9,26,41,42,43]. They assume (either explicitly or implicitly) a strict *Sequentially Consistent (SC)* memory model. However, most deployed systems implement weaker consistency models like TSO [18] or x86 [19]. A supervised system that only works with SC hardware is much less attractive.
- **No formalism.** The few proposals that address relaxed memory systems do so informally on an *ad hoc* basis [15,28,36]. It is well-known from existing memory models research that such specifications often lead to ambiguous interpretations and incorrect implementations.

**We build a firm foundation for future supervised systems** by addressing the above two drawbacks. We begin by studying two supervised systems originally proposed for SC (empty/full-bits [5] and deterministic multiprocessing [15]) and show that moving them to TSO introduces both correctness and performance problems due to incorrectly reordered metabit reads, load bypassing, and late exceptions. Rather than address these issues for just these two systems, we propose a general and formal solution. *Thus, we seek to make numerous previous supervised proposals more practical.*

To this end, we define a *supervised memory* model that unifies and formalizes the memory aspects of existing and future supervised systems, while conforming to

existing uses at an intuitive level. Specifically, it extends data memory with per-location metabits, primitives to read/write them, and higher-level supervised loads/stores. A supervised load (store) atomically tests metadata state, optionally generates an exception, performs a data load (store), and optionally updates metadata state.

We propose two formal consistency models for supervised memory— $TSO_{all}$  and  $TSO_{data}$ . Both models order data accesses with TSO [18] and then add constraints for metadata accesses. The simpler  $TSO_{all}$  requires that all data and metadata accesses conform to TSO, limiting the use of non-speculative store buffers (because a supervised store includes a metadata load).  $TSO_{data}$  relaxes  $TSO_{all}$  by allowing metadata accesses to *different* locations to appear out of program order.  $TSO_{data}$  permits using store buffers, but makes the programmer’s job more complex.

Inspired by earlier work on memory models [2,17], we propose *safe supervision* to resolve the tension between the simplicity of  $TSO_{all}$  and the performance of  $TSO_{data}$ . While formally defining safe supervision is subtle, its intuition follows common usage of many supervised systems:

A program is *safely supervised* if a supervised operation to a memory location is only used to control access to that memory location’s data.

For safely supervised programs, programmers can “get their cake and eat it too:” (a) reason with simpler  $TSO_{all}$  and (b) run on faster  $TSO_{data}$  systems. This is because safely supervised programs will not observe any reorderings a  $TSO_{data}$  system might do among metadata accesses to *different* locations.

Using execution-driven full-system simulation of two supervised systems—HARD [43] and TokenTM [9], we show that  $TSO_{data}$  can indeed provide significant performance benefits (up to 5% for HARD and 22% for TokenTM) over a  $TSO_{all}$  implementation.

Moreover, since a high-level simulator-based study alone can miss subtle but important implementation issues, we implement  $TSO_{data}$  in the RTL of the OpenSPARC T2, a publicly-available industry-designed multi-core system. We show that  $TSO_{data}$  can be implemented correctly, at low overhead, and delve into design issues such as handling late store buffer exceptions.

This work makes three contributions to supervised systems.

- **Support for TSO.** This paper is the first to closely examine supervised systems in the context of the widely-deployed TSO memory model (TSO is a legal implementation of x86) and identify problems with current proposals (Section 3).

- **Formal Approach.** We define **supervised memory**, a general memory model that supports a wide range of supervised systems (Section 4). We take a formal approach to specifying two relaxed supervised memory models ( $TSO_{all}$  and  $TSO_{data}$ ). We also specify a program property, **safe supervision**, that enables reasoning with simpler  $TSO_{all}$  while achieving the performance of  $TSO_{data}$ . We quantitatively demonstrate the performance potential of  $TSO_{data}$  with two case studies (Section 5).

- **Address Implementation Issues.** We explore  $TSO_{data}$  implementation issues with an RTL-level multi-core design and demonstrate solutions that enable the continued use of store buffers (Section 6).

## 2. Background

### 2.1. Supervised Systems

Memory supervision forms the basis of many proposals. Unbounded transactional memory systems (e.g., [6,7,9]) store transactional read- and write-sets in metadata and use it to detect transactional conflicts on memory accesses. Deterministic shared-memory multiprocessing [15] and interleaving-constrained multiprocessing [41] use metadata to track the last readers/writer to a memory location and control interactions between threads. Memory typestate trackers like HARD [43] and empty/full-bits [3,5,27,33] store dynamic state information about the data location. Dynamic information-flow tracking [14,34,37] proposals use metadata to trace ‘tainted’ data as they move through the memory system. Hardware-assisted garbage collectors (e.g., Azul’s Pauseless GC [13]) keep garbage-collection related state in metadata.

### 2.2. Architectural Support for Memory Supervision

Hardware metabits have been proposed and implemented as tagged memory architectures [16] since at least 1957. Early tagged memory machines (e.g., BNL’s MERLIN (*circa* 1957)) carried bits for identifying memory types. Later machines like HEP [33] and Tera [5] used tags to ease program synchronization.

More recently, Active Memory [21], MMP [39], iWatcher [42] and SafeMem [28] propose mechanisms for fine-grain access monitoring. While Active Memory targets memory system simulation, MMP focuses on memory protection. iWatcher and SafeMem seek to facilitate software debugging. However, these proposals do not support efficient metadata updates in hardware. Memtracker [36] addresses this issue with the addition of a software-programmable metadata transition table. Log-based Architectures [11,12] enable decoupled memory monitoring for security and debugging analy-

sis. While all the above proposals examine metadata implementations, we examine memory supervision at a more abstract level. Finally, past work addresses atomicity of metadata operations in both tagged [36,37] and decoupled [20,26] metadata implementations. We assume such atomicity mechanisms and focus on memory ordering issues, instead.

### 2.3. Memory Models

Hardware-centric memory models like SC, TSO and x86 define reorderings that are allowed or disallowed by hardware. In contrast, programmer-centric models [1,17] specify memory models in terms of a contract between the programmer and hardware. If the programmer provides certain information about memory operations, hardware guarantees that the execution will obey a simple model like SC.

Adve *et al.* [1,2] introduce the idea of *data-race-free* programs as one such contract. Programmers identify memory operations as ‘data’ or ‘synchronization’ accesses. A data race occurs if, in an SC execution, two data operations access the same location, at least one of them is a write and they are not ordered by intervening synchronization accesses. If a program is data-race-free, then its executions on efficient systems like TSO would still be equivalent to SC. Thus, programmer reasoning about hardware models is greatly simplified. Recent work to specify memory models for higher-level languages like Java and C++ has adopted the programmer-centric approach [10,22]. We present hardware-centric specifications in this work. Nevertheless, inspired by data-race-freedom, we define a program property, safe supervision, which could enable future programmer-centric specifications.

## 3. Motivation

Most existing supervised systems proposals (e.g. [7,9,26,41,42,43]) assume (either explicitly or implicitly) a stricter SC memory model. This section highlights three issues that arise when supervised systems are implemented on prevalent non-SC systems and motivates a formal approach for solving them.

### 3.1. TSO-lite, a simple TSO-consistent system

To present issues with weaker memory systems, we introduce *TSO-lite*, a simplified TSO-consistent system that abstracts away most implementation details except the presence of private non-speculative store buffers. Note that, throughout this paper store buffers refer to the entity that holds stores that are retired and committed by the processor and awaiting global memory ordering (also known as *senior stores*). We choose TSO since it is implemented by SPARC architectures, is formally spec-

ified, and represents a valid implementation of the x86 memory model (c.f., x86-TSO [31]).

The TSO-lite system consists of a set of processors, a per-processor store buffer and shared memory (Figure 1). It is a TSO-consistent system that executes and retires instructions in program order. TSO-lite places stores in a store buffer on retirement. When a store reaches the head of the store buffer, TSO-lite issues it to the mem-

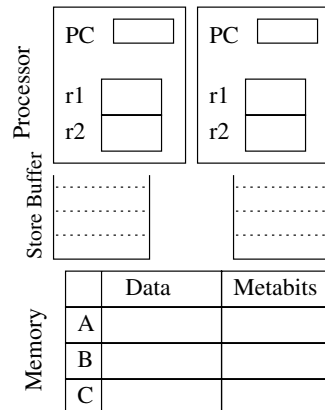


Figure 1. TSO-lite

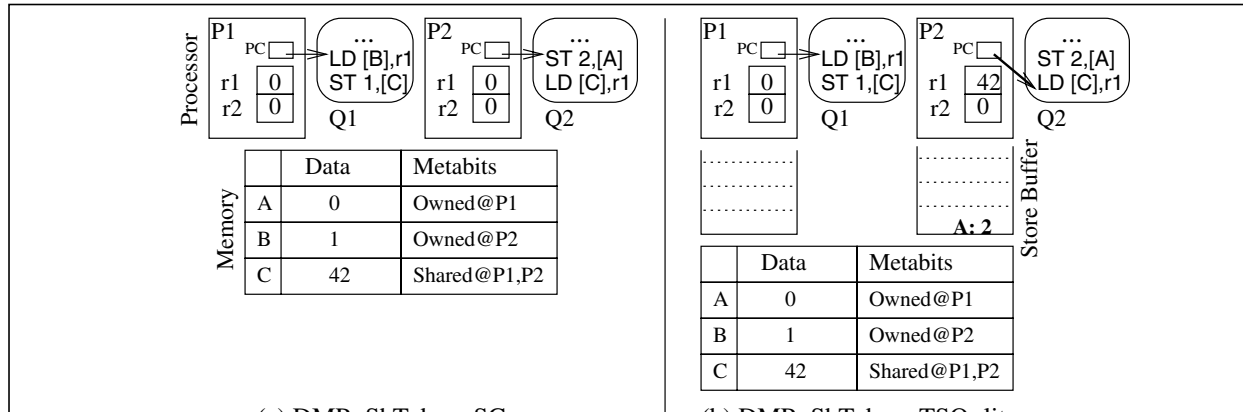
ory system and globally orders it. Loads can bypass their value from the youngest entry in the store buffer, if present. Otherwise they get the value from memory and retire. Thus, younger loads can get reordered before older stores in the global memory order. In addition to regular data, memory also carries metabits. Supervised systems typically require metabits to be accessed atomically along with regular memory operation [20,26,36,37]. *TSO-lite ensures atomicity* by accessing metabits (in global memory order) along with data accesses (similar to [36,37]). This atomicity requirement leads to subtle correctness and implementation issues.

**Issue I1 (Incorrect Metabit-read Reordering).** A store reads its metabits along with data on reaching the head of the store buffer. Thus, younger loads could read their metabits from memory prior to the store’s metabit read. This reordering can lead to correctness issues in some supervised systems as explained in Figure 2.

**Issue I2 (Load Bypass).** Loads can bypass their data value from the youngest matching store in the store buffer. However, until this store reaches the head of the store buffer, it cannot determine the correct metabit state. Hence, the load cannot bypass metabits value and has to stall until the matching store completes.

**Issue I3 (Late Exception).** When a store reads metabits at the head of the store buffer, they might indicate a trap to software. However, a precise exception is infeasible since the processor could have retired instructions that are younger than the trapping store. Figure 3 illustrates these last two issues.

While we use the (arguably) naive TSO-lite system to illustrate these issues, they arise in any system that exposes store buffers to software. In particular, they arise irrespective of whether the system uses a tagged or decoupled metadata implementation.



(a) DMP-ShTab on SC (b) DMP-ShTab on TSO-lite  
**Figure 2. Quanta Serialization in DMP-ShTab.**

Deterministic shared memory multiprocessing (DMP) [15] ensures that given an input, a multi-threaded execution always produces the same result. It divides execution into pre-determined chunks of instructions, known as *quanta*, that communicate only at the end in a deterministic order. Efficient implementations like *DMP-ShTab* use metabits to track data accesses and determine the first point of communication for each quantum. Quanta execute in parallel as long as they do not communicate and serialize thereafter. Figure 2 (a) shows an example of a two-processor DMP-ShTab execution on an idealized SC system. While processor P1 executes quantum Q1, P2 executes quantum Q2. In the proposed system, they execute in parallel until they reach the accesses to shared locations B and A respectively. DMP-ShTab blocks the processors and serializes their execution such that Q1 logically precedes Q2.

Figure 2 (b) shows the DMP execution running on TSO-lite. As earlier, P1 blocks when it reaches the first shared access (load to B). Meanwhile, P2 executes quantum Q2 and reaches its first shared access (store to A). The TSO-lite system places the store in the store buffer and continues execution. P2 detects the need to block only when this store reaches the head of the store buffer and reads the metabits. Meanwhile, P2 could execute the subsequent load to B or not, leading to two possibilities. If P2 executes the load, it reads the value ‘42’ from memory (as shown in the figure). If it does not, then it blocks and continues only after Q1 completes. In this case, P2’s read to B returns ‘1’ (the value written by P1’s store to B). Consequently, the two executions could diverge and the system is no longer deterministic.

### 3.2. Existing Proposals for Non-SC Supervised Systems

A few existing proposals make a commendable effort to address relaxed memory systems [15,28,36,40]. However, they are either *ad hoc* or informal and hence incomplete. We present some examples.

**Incorrect Metabit-read Reordering (I1).** The DMP paper attempts to mitigate this issue by providing the following guidance: “depending upon the consistency model of the underlying hardware, threads must perform a memory fence at the edge of a quantum” [15]. Unfortunately, this statement is ambiguous and can be interpreted in many ways, both correct and incorrect. A valid interpretation is that it suffices to insert a dynamic fence at the end of every quantum, which can lead to non-deterministic executions as shown in Figure 2. A correct interpretation is to insert a fence at the first point of communication for each quanta.<sup>1</sup> However, because any

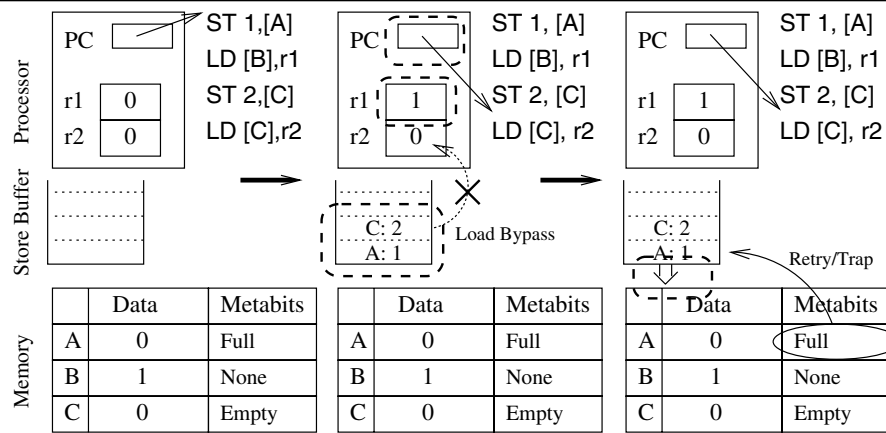
memory operation may result in communication, this means that all stores during the parallel phase must be followed by a fence, essentially eliminating any benefit of the store buffer in the parallel phase (the store buffer may be used as normal during the sequential phase).

Memtracker [36], another supervised system, addresses weaker memory models with store buffers as follows: “the simplest way to ensure correct behavior is to flush the write buffer when there is a state update and do the write directly to the cache.” While this is accurate for all the uses presented in their work, it does not address metabit-read reordering(I1).

**Load Bypass (I2).** To the best of our knowledge, none of the existing proposals address this issue.

**Late Exception (I3).** While some older proposals [29] tackle the late exception issue, modern supervised system proposals have largely ignored it.

1. Confirmed by DMP authors in a private communication as the interpretation assumed in their work.



**Figure 3. Empty/Full-bits on TSO-lite**

Empty/full-bits [33] enable lightweight synchronization among different threads of execution. They represent one of the three states—*None*, *Empty* and *Full* and are used as follows: loads (consumers) to a *Full* location complete and change the state to *Empty*. Similarly, stores (producers) to an *Empty* location complete and change the state to *Full*. However, if a load encounters an *Empty* state or a store encounters a *Full* state, hardware does not allow the memory operation to complete. It retries the operation for a fixed number of times before trapping to user-level software for higher level actions. Finally, loads and stores to blocks with the state *None* proceed as normal.

The figure illustrates the execution sequence of an empty/full-bits based program on the TSO-lite system with one processor. As the processor executes memory operations, the state changes from the previous step are shown in dashed circles. While the load to C runs into the Load Bypass (I2) issue, the subsequent draining of the store buffer leads to the Late Exception (I3) issue.

### 3.3. Discussion

A naive solution to issue I1 is to prohibit store buffers in the presence of memory supervision which can lead to significant performance loss (as we show in Section 5). On the other hand, permitting store buffers can cause some (but not all) supervised systems to exhibit incorrect execution. Which current and future supervised systems can safely reorder metabit-reads and thus use performance-enhancing store buffers? While a few supervised systems (e.g., DIFT) that do not read metabits as part of a store operation, trivially avoid this problem, the answer is non-trivial for the rest. Section 4 seeks a formal answer.

Even if a supervised system can safely reorder metabit reads, designers still need to address issues I2 and I3. Based on their dynamic frequency, designers have to pick solutions that minimize performance impact. We present a quantitative analysis in Section 5 and demonstrate low-cost solutions in Section 6.

## 4. Supervised Memory

We define a memory model that unifies and formalizes the memory aspects of existing supervised systems.

### 4.1. Definition

Supervised systems typically consist of three common components—metabits for storing metadata, memory operations that use metadata for data supervision and finally, a mechanism to jump to software handlers on encountering specific metadata values.

*Supervised memory* consists of a set of memory locations. Each location  $A$  is a two-tuple  $\langle A.d, A.m \rangle$  with  $A.d$  representing the data value and  $A.m$  representing the corresponding metadata value. A memory *access* is a read or write to a location's data or metadata component. Thus, an access to location  $A$  could be one of a data read ( $R_{A.d}$ ), data write ( $W_{A.d}$ ), metadata read ( $R_{A.m}$ ) or a metadata write ( $W_{A.m}$ ).

To enable supervision of data accesses, supervised memory provides higher-level memory *operations*. Each operation consists of a set of accesses. In addition to regular data operations like load and store, two supervised operations are provided: supervised loads ( $sLD_A$ ) (defined in Figure 4) and supervised stores ( $sST_A$ ), which are similar to supervised loads except that the data read is replaced with a data write. Thus, a supervised operation can be viewed as an atomic compound operation containing a metadata read access, a data access and (possibly) a metadata write access. The NEXT function is a programmable metadata transition

```

sLDA =>
Retry:
  atomic {
    curm = Val[RA.m]
    nextm = NEXT(OPLoad, curm)
    if (nextm == EXCEPTION) then
      Jump to Handler
    RA.d
    if (nextm ≠ curm)
      WA.m nextm
  }
Handler:
...

```

**Figure 4. Supervised Load**

function to efficiently update metadata in hardware. Finally, EXCEPTION is a special metadata value which, if read, transfers control to a user-level handler where appropriate software-specified actions can be taken. Control could optionally be transferred back to the supervised operation. While not complete, we find the above definition sufficient to formalize the shared memory aspect of a supervised system.

## 4.2. Consistency Models for Supervised Memory

A memory consistency model is essential to accurately specify the behavior of a shared-memory model like supervised memory. We focus on hardware-centric axiomatic specifications that are most accessible to hardware designers. The memory model is specified as a set of rules (*axioms*) that restrict the set of legal executions on a memory system.

Sequential consistency is widely accepted as the simplest model, closest to programmer intuition of shared-memory operation. However, it prohibits non-speculative reordering of *independent* accesses (i.e., accesses to different memory locations) and consequently, optimizations like store buffers. Thus, most popular architectures (e.g., x86) do not implement SC. On the other hand, TSO permits the use of private store buffers. While systems that implement TSO can still lead to non-intuitive behaviors, most programs (except well-known corner cases like Dekker’s algorithm) behave identically on both SC and TSO systems. Thus, TSO axioms provide a good baseline model to explore supervised memory models.

We now present informal definitions of two TSO-based supervised memory models. Bobba’s thesis (Appendix A, page 116 [8]) gives formal definitions.

### 4.2.1. TSO<sub>all</sub>: A simple TSO-like memory model

An execution conforms to TSO<sub>all</sub> if it obeys the TSO axioms (Table 1) with respect to **all** accesses in a

**Table 1. TSO axioms**

Store Order	Total order on all write accesses
Atomicity	No intervening accesses for atomic operations
Termination	All write accesses eventually complete
Load Value	Read returns latest write from memory or store buffer
Memory Barrier	No reordering across a barrier
ReadAny	Accesses cannot pass outstanding reads (Reordering Axiom 1/2)
WriteWrite	Writes cannot pass outstanding writes (Reordering Axiom 2/2)

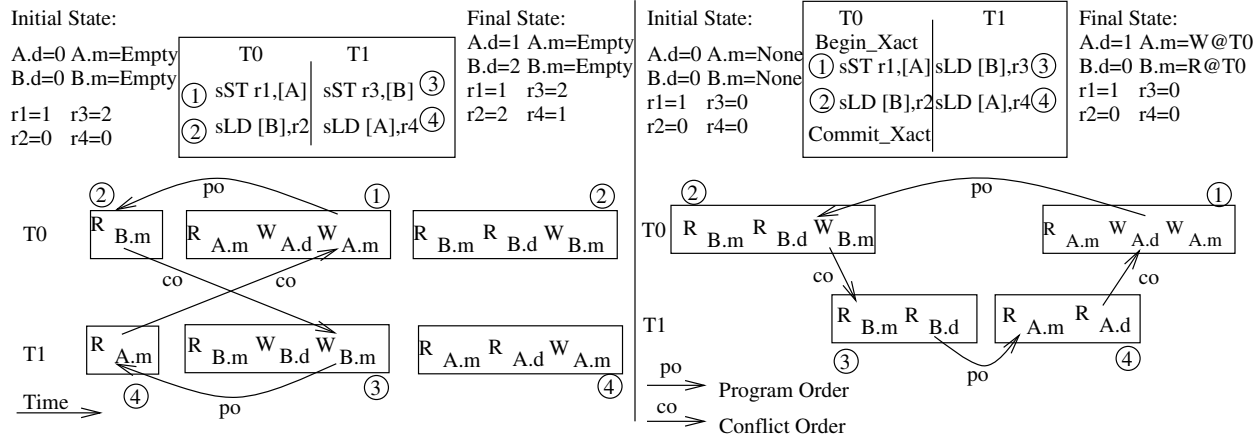
supervised program. Hence, in a TSO<sub>all</sub> execution, there is (a) a total order on all writes, (b) reads are not reordered with respect to earlier reads from the same processor, (c) writes are not reordered with respect to earlier reads or writes from the same processor, and (d) reads get their value from either the latest earlier write on the same processor or the latest earlier write in the total order.

**Simple to reason about.** Since TSO<sub>all</sub> restricts certain memory access reorderings, it can prevent non-intuitive behaviors. For example, consider the DMP example presented in Section 3.1. Since TSO<sub>all</sub> prohibits reordering two metadata read accesses, it would prevent incorrect metabit-read reordering (I1). Hence DMP on TSO<sub>all</sub> systems would not run into this correctness issue. Moreover, it is easy to see that programmers who are familiar with the conventional TSO model can also reason about TSO<sub>all</sub>.

**Whither performance?** TSO<sub>all</sub>, though simple, can introduce performance problems in real implementations. Since TSO<sub>all</sub> prohibits reordering any two read accesses, it prevents an independent supervised load from passing a prior supervised store (both contain a metadata read). In the absence of speculation, supervised loads have to stall until all outstanding supervised stores complete, thus making store buffers ineffective. Consequently, TSO<sub>all</sub> implementations perform like SC systems on programs with frequent supervised operations.

Fortunately, many supervised memory programs do not seem to rely on all the ordering guarantees provided by TSO<sub>all</sub>. We, in the same spirit as programmer-centric models, observe that **if we capture the orders that can be safely ignored during an execution, we could devise much more efficient models.**

To help identify these unnecessary orders, we examine some supervised memory executions that would be illegal in a TSO<sub>all</sub> system but are nevertheless semantically correct. The first example (Figure 5 (left))



shows a program that uses empty/full-bits. In the program, A and B can be viewed as one-entry queues. Each thread places an element in one queue and picks a new element from another queue. At the bottom, we show a non-TSO<sub>all</sub> execution that, nevertheless, leads to a semantically correct result with both threads reading the latest values from the queues. Observe that metadata accesses to A.m are only used to ensure that operation ①'s data write happens before ④'s data read. Similarly, accesses to B.m are used to only order data accesses from ② and ③. Thus, it is not wrong to order ②'s metadata read before ①. The second example (Figure 5 (right)) shows a transactional memory (TM) program wherein one thread is executing a transaction while another concurrently reads locations that are accessed by the transaction. Metadata is used to carry the transaction's read- and write-set information and supervised operations prevent conflicting accesses. We again, observe that accesses to A.m and B.m are intended only to control their corresponding data accesses.

#### 4.2.2. Safe Supervision

To generalize these observations, we propose a program property, *Safe Supervision*, that encapsulates the ordering flexibility provided by supervised programs.

A program is safely supervised if a supervised operation to a memory location is only used to control access to that memory location's data. Bobba's thesis provides the formal definition (Appendix A.2.1, p. 122 [8]).

Since safely supervised programs use metadata to control access to a memory location's data (and not to order access to other locations), hardware can safely reorder metadata operations to one address with respect to accesses to other addresses without this reordering being observed by the programmer. **Thus, safe supervision enables a precise specification to when hardware can safely reorder metabit-reads (issue I1).**

**Which programs are safely-supervised?** Safe supervision is a contract between supervised memory hardware and supervised software (consisting both supervised system software and the high-level program). We postulate (but do not formally prove) that most existing uses of metabits like empty/full-bits, transactional memory, MMP, Active Memory, tagged memory, memory tpestate trackers like HARD and Memtracker, and hardware-assisted garbage collection obey safe supervision since metadata carries information related only to its corresponding data location. However, exceptions exist. Consider the DMP example in Section 3.1. Metadata accesses are used by the underlying supervised system software to both track a location's sharing status **and** to detect the first non-private data access. This 'first-access' requirement implies that a metadata read relies on the completion of all previous metadata reads. Hence, DMP programs are not safely-supervised. With current supervised systems, in rare cases, high-level programmers (Figure 6) can explicitly choose to violate safe supervision. Such programs forgo the performance benefits of more relaxed memory models such as the one we propose next.

#### 4.2.3. TSO<sub>data</sub>: An Efficient Memory Model

Informally, an execution conforms to TSO<sub>data</sub> if it obeys TSO's reordering axioms (Rows 6&7, Table 1) with respect to **data** accesses and obeys the rest of the axioms (Row 1-5, Table 1) for all the accesses. Hence, in a TSO<sub>data</sub> execution there is (a) a total order on all writes, (b) data reads are not reordered with respect to earlier data reads from the same processor, (c) data writes are not reordered with respect to earlier data reads or data writes from the same processor, and (d) all reads get their value from either the latest earlier write on the same processor or the latest earlier write in the total order.

**Good performance.**  $\text{TSO}_{\text{data}}$  can be implemented efficiently. Since  $\text{TSO}_{\text{data}}$  does not prohibit reordering independent metadata reads, independent supervised loads can pass supervised stores that are placed in store buffers. Thus, implementations can effectively use store buffers. It can be seen that  $\text{TSO}_{\text{data}}$  is a strictly weaker model than  $\text{TSO}_{\text{all}}$  since every  $\text{TSO}_{\text{all}}$ -consistent execution is also a  $\text{TSO}_{\text{data}}$ -consistent execution and there exist  $\text{TSO}_{\text{data}}$ -consistent executions that are not  $\text{TSO}_{\text{all}}$ -consistent (Figure 5). By virtue of allowing more legal executions,  $\text{TSO}_{\text{data}}$  gives more flexibility to build efficient hardware.

**Whither simplicity?**  $\text{TSO}_{\text{data}}$  is, however, potentially trickier to reason about. Consider the empty/full-bits based non safely-supervised program in Figure 6. Operation ② has to change the metadata to ‘Empty’ before operation ③ can complete. This is used to ensure that operation ① happens before operation ④. A  $\text{TSO}_{\text{data}}$ -consistent execution need not enforce this order since T1’s data read to A can pass its earlier metadata read to B. Consequently, it leads to an incorrect execution. Note that  $\text{TSO}_{\text{all}}$  would prohibit such an execution since T1’s reads would not be reordered.

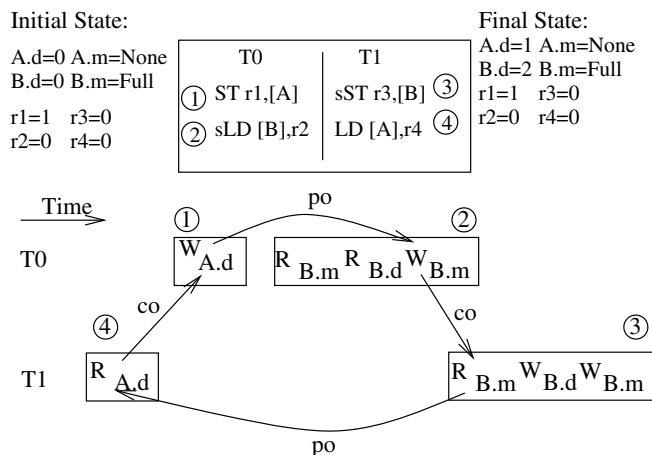
Bobba’s thesis (Appendix A.3, p. 126 [8]) presents a proof that **for programs that obey the safe supervision property,  $\text{TSO}_{\text{data}} = \text{TSO}_{\text{all}}$** , i.e., the program would behave identically on both the systems. Similar to data-race-freedom, software programmers that adhere to safe supervision can reason about  $\text{TSO}_{\text{all}}$  systems and yet get the performance of  $\text{TSO}_{\text{data}}$  systems.

## 5. Quantitative Case Studies

It is well-known that store buffers provide tangible performance benefits in conventional systems. In this section, we demonstrate the performance potential of store buffers (facilitated by  $\text{TSO}_{\text{data}}$ ) for supervised systems using two previously-proposed safely-supervised systems—HARD [43] and TokenTM [9].

### 5.1. Methodology

The performance benefit of store buffers depends primarily on the supervised system of interest and the



**Figure 6. Unsafe Supervision**

specific core design. We use execution-driven full-system simulation based on the Wisconsin GEMS toolset [23] in conjunction with Virtutech Simics [38] for evaluation. Table 2 gives the key system parameters for the two supervised systems.

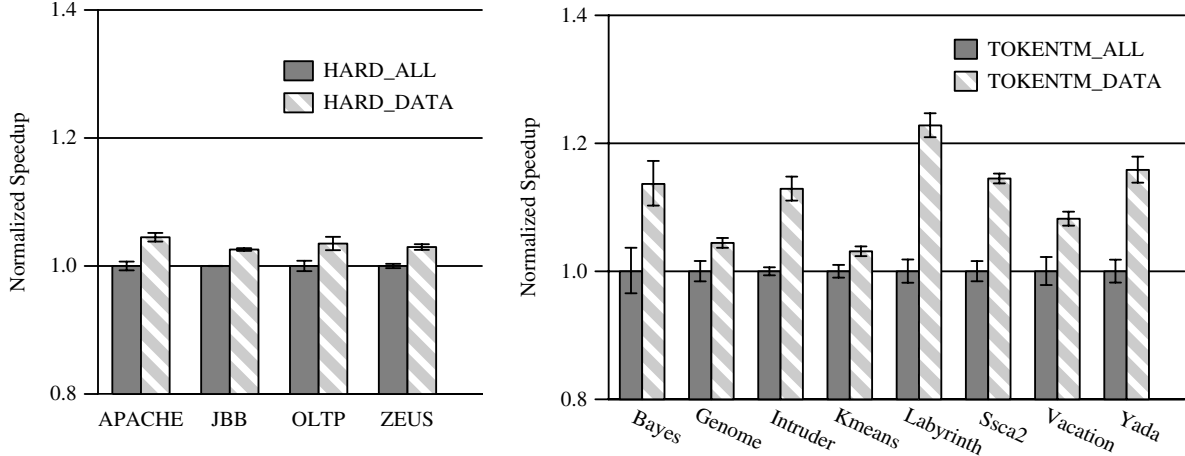
### 5.2. Hardware-Assisted Lockset-based Race Detection (HARD)

Zhou et al. [43] propose HARD to efficiently detect mutual exclusion bugs in multi-threaded software. It uses the lock-set algorithm to catch violations of common locking paradigms. For instance, it ensures that all accesses to a shared variable are protected by at least one common lock. It does so by tracking the locks currently held by a thread (*LockSet*) and comparing them against the set of locks used to protect the variable thus far (*CandidateSet*). HARD tracks these lock-sets at cache-block granularity and uses Bloom filters to efficiently represent them in hardware. In order to identify shared memory locations, it uses a simple finite state machine that initializes blocks in private states and transitions them to shared states when they are accessed by multiple threads.

To define HARD using supervised memory, we first assign semantics to metadata. In this case, metadata stores the sharing state of the location and its Candidate-

**Table 2. Key Supervised System Parameters**

	HARD	TokenTM
Core	8-core out-of-order, 4-wide, 32-entry instruction window, 64-entry ROB, Fully pipelined functional units, 6-stage integer pipeline, YAGS branch predictor, Store-set predictor	8-core 1 IPC in-order
L1 Cache	4-way 32 KB split/private, 1 cycle uncontended latency	
L2 Cache	8-way, 16MB unified/shared, 34 cycle uncontended latency	
Interconnection	Crossbar, 64 byte 14-cycle links	
Coherence	MESI directory protocol, Full-bit vector L2 Directory	
Memory Controllers	4 on-chip memory controllers	
Memory	4 GB 250-cycle response latency	



**Figure 7. Performance Benefit of  $TSO_{data}$  vs.  $TSO_{all}$ : HARD (left) and TokenTM (right)**

Set, if applicable. The NEXT function defines the transitions between the metadata states. For instance, it specifies that on a store to a ‘shared’ location with CandidateSet  $C$  (i.e.,  $cur_m = Shared \# C$ ) by a thread with LockSet  $L$ , the new metadata state,  $next_m = Modified \# C \cap L$ . Metadata states where the CandidateSet is empty are defined to be EXCEPTION states. The exception handler logs the memory location and the old CandidateSet for subsequent inspection by the programmer. It then resets the metadata to avoid recurring exceptions on this location. HARD is safely-supervised. The locksets and the finite state machine are specific to a memory location and the exception handler only logs the trapping memory location.

We build two versions of HARD— $HARD_{all}$  that operates with the  $TSO_{all}$  memory model and  $HARD_{data}$  that operates with  $TSO_{data}$  using a 32-entry store buffer. We model the full functionality of store buffers including prohibiting load bypassing for supervised loads. We do not model the cache overheads of metadata since the overheads are similar for both  $TSO_{data}$  and  $TSO_{all}$  and they are estimated to be small (0.1-2.6% by Zhou et al. [43]). We evaluate the two implementations using the Wisconsin commercial workload suite [4]. Since we are unable to modify the source code, we use published prediction techniques to identify lock acquisition and release in the simulator [30].

### 5.3. TokenTM: Token-based Transactional Memory

TokenTM is an unbounded hardware TM system that uses memory tokens to enforce a per-location single-writer/multiple-reader invariant. As a result, it detects memory conflicts among concurrent speculative transactions. Each memory location has  $T$  logical tokens and TokenTM requires a thread to acquire one ( $T$ ) token(s) before it can read (write) a location. Memory metabits carry information about which threads have

acquired tokens for the location thus far. Metabits are also overloaded to perform TM version management, wherein the thread acquiring  $T$  tokens also logs the location’s old data value in a thread-private log.

To define TokenTM using supervised memory, we first assign semantics to metadata. In this case, metadata carries the acquired token information for the location. The NEXT function enforces the TokenTM token acquisition rules. It checks that a thread either has sufficient tokens or can acquire the tokens to perform the memory operation. For instance, on a store by thread  $t1$  to a location with all tokens available (i.e.,  $cur_m = None \text{ Taken}$ ), the new metadata state  $next_m = T \text{ tokens} @ t1$ . However, if the thread cannot acquire the required number of tokens, the new metadata state would be an EXCEPTION state. This would invoke the TokenTM software handlers that perform conflict resolution. TokenTM is safely-supervised. It uses metadata to detect a conflicting access on the current location. This information is not used to infer the presence or absence of conflicts on prior or later accesses.

We build two versions of TokenTM— $TokenTM_{all}$  that implements the  $TSO_{all}$  memory model and  $TokenTM_{data}$  that implements  $TSO_{data}$  with a 32-entry store buffer. We modify the source code obtained from the authors of the original work to incorporate store buffers. Our implementation drains the store buffers on a transaction commit. We evaluate the performance of the two TokenTM implementations using the STAMP benchmark suite [24].

### 5.4. Results

Figure 7 shows the overall performance comparison between the  $TSO_{data}$  and  $TSO_{all}$  systems. Table 3 presents some behavioral characteristics of supervised system programs. The first three rows give the percentage

**Table 3. Supervised Operations Characteristics**

	HARD <sub>data</sub>				TokenTM <sub>data</sub>							
	Apache	JBB	OLTP	Zeus	Bayes	Genome	Intruder	Kmeans	Labyrinth	Ssca2	Vacation	Yada
%Updates	1.5	0.7	0.4	2.2	9.7	20.6	6.0	1.6	2.4	2.0	10.5	5.8
%Exceptions	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1	0.2	<0.1	<0.1	<0.1	<0.1	<0.1
%SupLoads	66.1	65.8	69.2	62.9	75.9	87.3	90.4	93.7	82.8	87.7	83.2	70.2
%SupLoadSBHit	4.1	1.7	4.3	4.9	0.1	0.1	0.2	<0.1	0.1	0.2	<0.1	0.3

of supervised operations that update metadata, that take an exception and that are supervised loads respectively.

We focus on the three key issues—reordered metabit-reads, load bypass and late exceptions—that arise in supervised systems on relaxed memory systems.

**Incorrect Metabit-read reordering (I1).** While HARD<sub>all</sub> and TokenTM<sub>all</sub> do not reorder metabit-reads, the safe supervision property enables HARD<sub>data</sub> and TokenTM<sub>data</sub> to safely do so using store buffers. Since TokenTM and HARD treat every memory operation as supervised, this optimization is important for performance. HARD<sub>data</sub> exhibits a speedup ranging from 3% in JBB to 5% in Apache over HARD<sub>all</sub> on the commercial benchmarks (Figure 7 (left)). On the other hand, TokenTM<sub>data</sub> exhibits much higher speedups over TokenTM<sub>all</sub> ranging from 3% in Kmeans to 22% in Labyrinth (Figure 7 (right)). TokenTM<sub>data</sub> achieves better overall speedups than HARD<sub>data</sub> for two major reasons. First, in-order cores used by TokenTM systems are more sensitive to reordering issues than out-of-order cores. Second, the use of store buffers reduces the conflict window for transactions in TokenTM<sub>data</sub>.

Overall, **our results show that there exist supervised systems and core designs where TSO<sub>data</sub> provides significant performance over TSO<sub>all</sub>.**

**Load Bypass (I2).** The final row of Table 3 gives the percentage of supervised loads that find a matching store in the store buffers. Our results show that only a small percentage of loads are stopped from bypassing their values from the store buffers. Hence, their impact on performance is likely to be small.

**Late Exceptions (I3).** A few supervised operations take an exception (Table 3, row 2). Thus, while it is important to handle late exceptions correctly, the performance of exception handlers is not critical.

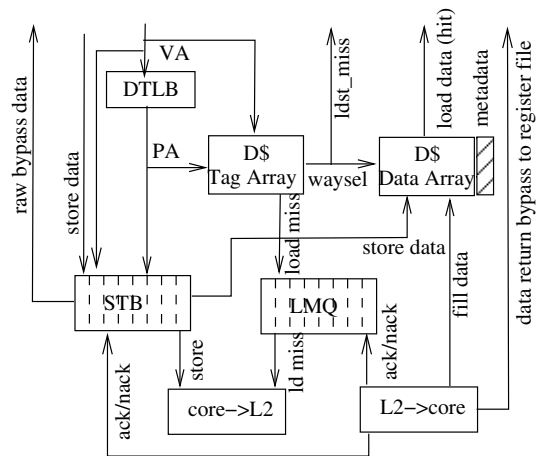
## 6. Implementation Case Study

In order to study supervised memory “end to end” and to ensure that the above simulation studies did not gloss over real implementation issues, we examined the RTL on an industrial design, added key hardware com-

ponents and developed associated low-level trap handlers. Here we highlight key findings of the study.

We choose the OpenSPARC T2 multi-core system because it has a publicly available RTL model and already implements TSO for data accesses. Each of eight 8-thread cores has two integer pipelines, a floating-point pipeline and a memory pipeline and can issue instructions from at most two threads per cycle. It buffers stores in a per-thread 8-entry store buffer (STB) and issues them to the shared L2 cache in order. Loads bypass values from the store buffer, if present.

Figure 8 illustrates the datapath of the load-store unit (LSU) that executes and retires supervised operations. Two issues deserve particular mention. First, we handle supervised loads that match an address in the store buffer as a partial hit to avoid bypassing (Issue I2). The load is stalled until the store completes. Second, we handle metadata exceptions on supervised stores (issue I3) with a deferred trap and *sparse restart* [25] (similar to Qiu *et al.* [29]). On a late exception, we architecturally expose the incomplete stores in the store buffer to a hypervisor-level trap handler. We re-use the existing store buffer diagnostic interface (ASI\_STB\_ACCESS) to read them out. To flexibly complete these operations later, software needs the entire context of the store oper-



**Figure 8.** OpenSPARC Load-Store Unit (LSU)

```

ENTRY (dump_store_buffer)
...
/* %g1 contains HEAD of dump buffer */
GET_ERR_DFESR(%g4, %g3)
srlx %g4, DFESR_STB_INDEX_SHIFT, %g4
and %g4, DFESR_STB_INDEX_MASK, %g3
/* g3 contains TAIL of store buffer */
or %g0, ASI_STB_FIELD_CURR_PTR, %g5
ldxa [%g5]ASI_STB_ACCESS, %g4
/* g4 contains HEAD of store buffer */
loop:
/* Check for wrap-around */
cmp %g3, 7
bg, pn %xcc, wrap_around
nop
dump_entry:
...
or %g3, ASI_STB_FIELD_DATA, %g5
ldxa [%g5]ASI_STB_ACCESS, %g2
/* Store Buffer data */
stx %g2, [%g1+ERR_STB_DATA]
or %g3, ASI_STB_FIELD_MARKS, %g5
ldxa [%g5]ASI_STB_ACCESS, %g2
/* Store Buffer address and byte marks */
stx %g2, [%g1+ERR_STB_MARKS]
...
cmp %g3, %g4 ! If (TAIL==HEAD) break;
bne %xcc, loop
inc %g3

```

**Figure 9. Store Buffer Dump Handler**

ation (e.g., virtual address and address space identifier). Hence, we expand the store buffers to carry this additional information. Figure 9 shows an extract of the handler code for reading out the contents of a store buffer. The handler begins from the head of the buffer and walks down to the tail; reading out and saving the contents for subsequent execution. A subtle deadlock issue arises if the stores in the handler use the same store buffer entries (like normal ASI stores in SPARC). Hence, we allocate a separate logical single-entry store buffer for the handler code.

We make coarse estimates of implementation cost using Synopsys Design Compiler [35] in conjunction with its 90nm technology library (saed90nm) and Cacti 5.3 [32] for relative (not absolute) memory array comparisons, as Cacti is known to be pessimistic for small memories. This study asks and answers the three following questions.

**Can we correctly handle supervised store exceptions? Yes.** As shown in Figure 9, the exception handler uses ASI loads to access the store buffer entries on an exception. Each access takes about 21 cycles. Overall, the handler takes 203 cycles to read out half the store buffer (4 entries).

**Is it easy to add metabits to the existing design? Possible.** We widened the data paths and did not have to change the control paths in the DRAM and L2 controllers. We modified 6 (of 50) design files for the DRAM controller and 3 (of 8) files for the L2 controller.

**Table 4. Memory Area (in mm<sup>2</sup>) in OpenSPARC T2**

Module	Base	TSO <sub>data</sub>	%Overhead
DRAM Controller	0.19	0.22	15.8%
L2 Bank	28.83	31.71	10.0%
L2 Controllers	0.21	0.21	<1%
L1 Data Bank	1.20	1.32	10%
Store Buffer RAM	0.06	0.17	83%
Other	52.93	52.93	0%
Total	305.69	331.65	8.5%

### Do metabits impose a significant area overhead? No.

In terms of area, the overhead mainly arises in the memory arrays. Table 4 presents their area for various modules in the base and TSO<sub>data</sub> designs. The biggest memory array (the L2 data banks) incurs an overhead of 10%.

## 7. Conclusions and Future Work

This work builds a firm foundation for future supervised systems by addressing the two drawbacks of many existing proposals: *SC-centricity* and *no formalism*. To this end, we develop supervised systems that (1) operate with the TSO (and thus x86) memory model, and (2) are formally specified.

We also introduce a program property, safe supervision, whose intent is similar to data-race-freedom and simplifies programmer reasoning about supervised systems. As one astute reviewer observed, safe supervision restricts metadata use to be coherence-like (i.e., per-address) rather than consistency-like (across address).

Future work could explore programmer-centric and weaker hardware-centric models for supervised memory, tackle more supervised systems, and expand safe supervision to cover a wider range of programs.

## 8. Acknowledgements

This work is supported in part by the U.S. National Science Foundation (CNS-0551401, CNS-0720565 and CNS-0916725), Sandia/DOE (#MSN123960 - DOE890426), and University of Wisconsin (Kellett award to Hill). Lupon’s visit to Wisconsin was supported by the Generalitat de Catalunya under grant 2009-BE2-00306. The views expressed herein are not necessarily those of the NSF, Sandia, DOE, or GdC. Hill and Wood have a significant financial interest in Microsoft. Bobba was a PhD student at the University of Wisconsin-Madison when this work was performed.

We thank Siddharth Barman, Arkaprava Basu, Dan Gibson, Michael Swift, Amit Kumar, Greg Wright, and the anonymous reviewers for their comments.

## 9. References

- [1] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, December 1996.
- [2] Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition. In *Proc. of the 17th Annual Intl. Symp. on Computer Architecture*, pages 2–14, May 1990.
- [3] Agarwal et al. The MIT Alewife machine: Architecture and Performance. In *Proc. of the 22nd Annual Intl. Symp. on Computer Architecture*, pages 2–13, June 1995.
- [4] Alameldeen et al. Evaluating Non-deterministic Multi-threaded Commercial Workloads. In *Proc. of the 5th Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 30–38, February 2002.
- [5] Alverson et al. The Tera computer system. In *Proc. of the 4th Intl. Conf. on Supercomputing*, pages 1–6, June 1990.
- [6] Baugh et al. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In *Proc. of the 35th Annual Intl. Symp. on Computer Architecture*, June 2008.
- [7] Blundell et al. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proc. of the 34th Annual Intl. Symp. on Computer Architecture*, June 2007.
- [8] Jayaram Bobba. *Hardware Support for Efficient Supervised and Transactional Memory Systems*. PhD thesis, University of Wisconsin-Madison, 2010. [http://www.cs.wisc.edu/multifacet/theses/jayaram\\_bobba\\_phd.pdf](http://www.cs.wisc.edu/multifacet/theses/jayaram_bobba_phd.pdf).
- [9] Bobba et al. TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In *Proc. of the 35th Annual Intl. Symp. on Computer Architecture*, June 2008.
- [10] Hans-J Boehm and Sarita V. Adve. Foundations of the C++ Concurrency Memory Model. In *Proc. of the SIGPLAN 2008 Conf. on Programming Language Design and Implementation*, pages 68–78, June 2008.
- [11] Chen et al. Log-based architectures for general-purpose monitoring of deployed code. In *ASID '06: Proc. of the 1st workshop on Architectural and System Support for improving software dependability*, pages 63–65, 2006.
- [12] Chen et al. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In *Proc. of the 35th Annual Intl. Symp. on Computer Architecture*, pages 377–388, June 2008.
- [13] Click et al. The Pauseless GC Algorithm. In *VEE '05: Proc. of the 1st Intl. Conference on Virtual Execution Environments*, pages 46–56, 2005.
- [14] Dalton et al. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proc. of the 34th Annual Intl. Symp. on Computer Architecture*, June 2007.
- [15] Devietti et al. DMP: Deterministic Shared Memory Multiprocessing. In *Proc. of the 14th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, March 2009.
- [16] Edward A. Fesutel. On the advantages of tagged architecture. *IEEE Transactions on Computers*, 22(7):644–656, 1973.
- [17] Gharachorloo et al. Memory Consistency and Event Ordering in Scalable Shared-Memory. In *Proc. of the 17th Annual Intl. Symp. on Computer Architecture*, pages 15–26, May 1990.
- [18] Hangal et al. TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model. In *Proc. of the 31st Annual Intl. Symp. on Computer Architecture*, June 2004.
- [19] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, Part 1*, September 2009.
- [20] Hari Kannan. Ordering Decoupled Metadata Accesses in Multiprocessors. In *Proc. of the 42nd Annual IEEE/ACM International Symp. on Microarchitecture*, December 2009.
- [21] Alvin R. Lebeck and David A. Wood. Active Memory: A New Abstraction for Memory System Simulation. *ACM Transactions on Modeling and Computer Simulation*, 7(1):42–77, January 1997.
- [22] Manson et al. The Java Memory Model. In *POPL '05: Proc. of the 32nd Symposium on Principles of Programming Languages*, pages 378–391, 2005.
- [23] Martin et al. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, pages 92–99, September 2005.
- [24] Minh et al. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.
- [25] Mayan Moudgill and Stamatis Vassiliadis. Precise Interrupts. *IEEE Micro*, 16(1):58–67, 1996.
- [26] Vijay Nagarajan and Rajiv Gupta. Architectural Support for Shadow Memory in Multiprocessors. In *VEE '09: Proc. of the 2009 Intl. Conference on Virtual Execution Environments*, pages 1–10, 2009.
- [27] Gregory M. Papadopoulos and David E. Culler. Monsoon: An explicit token-store architecture. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 398–407, 1998.
- [28] Qin et al. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *Proc. of the 11th IEEE Symp. on High-Performance Computer Architecture*, February 2005.
- [29] Xiaogang Qiu and Michel Dubois. Tolerating Late Memory Traps in ILP Processors. In *Proc. of the 26th Annual Intl. Symp. on Computer Architecture*, May 1999.
- [30] Ravi Rajwar and James R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proc. of the 34th Annual IEEE/ACM International Symp. on Microarchitecture*, December 2001.
- [31] Sewell et al. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.
- [32] Shyamkumar et al. CACTI 5.1. Technical Report HPL-2008-20, Hewlett Packard Labs, 2008.
- [33] Burton J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *Society of Photo-optical Instrumentation Engineers*, 298:241–248, 1981.
- [34] Suh et al. Secure Program Execution via Dynamic Information Flow Tracking. pages 85–96.
- [35] Synopsys. Synopsys Design Compiler Ultra. <http://www.synopsys.com/Tools/Implementation/RTLsynthesis/Pages/DCUltra.aspx>, 2009.
- [36] Venkataramani et al. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In *Proc. of the 13th IEEE Symp. on High-Performance Computer Architecture*, pages 273–284, February 2007.
- [37] Venkataramani et al. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Proc. of the 14th IEEE Symp. on High-Performance Computer Architecture*, pages 173–184, February 2008.
- [38] Virtutech Inc. Simics Full System Simulator. <http://www.simics.com/>.
- [39] Witchel et al. Mondrian memory protection. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 304–316, October 2002.
- [40] Xu et al. A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording. In *Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 49–60, October 2006.
- [41] Jie Yu and Satish Narayansamy. A Case for an interleaving constrained shared-memory multi-processor. In *Proc. of the 36th Annual Intl. Symp. on Computer Architecture*, pages 325–336, June 2009.
- [42] Zhou et al. iWatcher: Efficient Architectural Support for Software Debugging. In *Proc. of the 31st Annual Intl. Symp. on Computer Architecture*, pages 224–237, June 2004.
- [43] Zhou et al. HARD: Hardware-Assisted Lockset-based Race Detection. In *Proc. of the 13th IEEE Symp. on High-Performance Computer Architecture*, pages 121–132, February 2007.