Reactive SC: Speculatively relaxing memory consistency model contraints using dynamic classification of cache blocks

Shaizeen Aga
The University of Michigan, Ann Arbor
shaizeen@umich.edu

Cory Perry
The University of Michigan, Ann Arbor
coryperr@umich.edu

Aaron Tami
The University of Michigan, Ann Arbor
atami@umich.edu

Abstract

Making parallel machines programmable is paramount to exposing and eventual harnessing of the performance of multiple cores. Memory consistency model plays an important role is deciding how programmable a parallel machine is, for it affects the assumptions a programmer can make. Sequential consistency is arguably the most intuitive memory model but is rarely implemented in practice owing to the concomitant performance costs.

In this paper we build upon previous work[12] which relaxes constraints imposed by Sequential Consistency by dynamically classifying memory accesses based on the kind of data that is accessed. While in [12] the authors classify memory accesses at page level, we leverage existing cache coherence infrastructure and by adding some more state information to the directory in a cache coherence protocol, classify memory accesses at cache block level. We show with our simulations on PARSEC[3] and SPLASH2[13] benchmarks that *Reactive SC* is successful in closing the performance gap between SC and TSO. *Reactive SC* is within the vicinity of 1.1% to 6.5% of TSO and performs about 1.9% to 5.4% better than page level implementation.

General Terms Design, Language, Performance

Keywords Memory Consistency, Parallel Programming, Sequential Consistency

1. Introduction

The advent of chip multiprocessors and their pervasiveness has led more and more programmers to use parallel programming to harness underlying cores for performance. One of the biggest luxuries a programmer gives up on moving from a sequential to parallel programming is the ability to reason in clear terms about the order in which instructions by multiple processors get executed. This "*happens-before*" relationship is implicitly guaranteed in a sequential program by the program order that the programmer has specified. To help programmer reason about this order, a formal contract or memory consistency model is defined. This dictates what value 'a read' by a processor returns. A consistency model lays down constraints on the completion order for memory accesses by each processor.

Several consistency models have been proposed that vary in the constraints they enforce [2, 6, 9]. Most uniprocessor hardware and software optimizations which overlap and/or reorder memory operations have to be compromised in order to abide by these constraints [1]. Of the consistency models proposed in literature, Sequential consistency is considered the most intuitive given it is a logical extension from uniprocessor to multiprocessor. Unfortunately this model comes with an expensive performance cost, because most modern hardware optimizations such as reordering, overlapping, and coalescing violating this model. Because of this, manufacturers chose to support more relaxed memory consistency models [1] which allow these memory optimizations.

There has been considerable effort in the research community towards the design of efficient SC hardware [4, 5, 7, 11]. For example, in-window speculation proposed by Gharachorloo et al. [7] allows memory operations to be reordered before they are retired from the ROB. This technique effectively delineates coherence activity from the imposition of consistency constraints; however, given that the store buffer still needs to be drained for a load to retire, in window speculation alone does not suffice for an efficient SC design. Aggressive proposals which use out-of-window speculation have also been proposed [7], but have concomitant complex recovery mechanisms which have not been realized in practical hardware implementations.

In this paper we build upon previous work [12] which argues that we need not treat all memory locations the same and thus impose the same constraints on them uniformly. Memory operations to locations which are private to a processor can be freely reordered or overlapped without any constraints, while operations to locations which are not private need to be subject to the constraints imposed by the memory consistency model under consideration. While in [12] memory locations are classified at page level, we demonstrate that much performance increase is left untapped by this implementation and that we can further exploit this property at a finer granularity by classifying memory locations at the cache block level. To do this, we have augmented the directory in the cache coherence protocol to keep track of the classification of memory locations and we use this classification to effectively relax the consistency constraints. We call our proposal *Reactive SC*. Our results show that *Reactive SC* delivers performance close to TSO and is thus effective in closing the performance gap between SC hardware and TSO hardware.

The contributions of this work are as follows:

- We show that classification of memory operations at page level to relax memory consistency constraints leaves much room for improvement which can be further exploited by alternatively classifying memory operations at cache block level.
- We present in this paper a protocol to classify memory operations at cache block level and its implementation, which we call *Reactive SC*.
- Simulation results demonstrate that the proposed solution *Reactive SC* closes the performance gap between SC and TSO.

This paper is organized as follows. Section 2 presents background and describes the key insight along with the page level implementation as proposed in [12]. Section 3 presents *Reactive SC* and describes complete architecture for it. Section 4 evaluates it. Section 5 presents our conclusions.

2. Background and Motivation

In this section we describe the technique employed in [12] and highlight the untapped opportunity to obtain better performance for the proposed SC design by decreasing the granularity of classification from page level to cache block level.

2.1 Sequential Consistency: Basics

Multicore chips that support shared memory accesses allow multiple threads to concurrently read and write to a single shared address space. Most out-of-order processors employ optimizations like non-fifo store buffers and non-blocking reads, which allow loads and stores to complete out of order with respect to each other, effectively hiding memory latency. These techniques work well for uniprocessors, but for multicore processors that support shared memory accesses and local caches an inconsistent view of memory can be propagated across cores. Memory models specify the allowed behavior of multithreaded programs with shared memory by imposing restrictions on the order of shared memory accesses initiated by each processor.

Sequential consistency, which is by far the most intuitive memory model, specifies that the system must appear to execute all loads and stores to all memory locations in a global order that respects the program order of each thread. This mandates following four ordering constraints:

- *Load* $\Rightarrow$ *Load order*: Loads should complete before letting any subsequent loads complete. A load is considered complete when the return value is bound and cannot be modified by any other write operation.
- *Store* $\Rightarrow$ *Store order*: Stores should complete before letting any subsequent stores complete. A store is considered complete when the value written is visible to all processors.
- *Load* $\Rightarrow$ *Store order*: Preserve ordering between a load and a subsequent store.
- *Store* $\Rightarrow$ *Load order*: Preserve ordering between a store and a subsequent load.

It is the imposing of these constraints that limit the performance of SC as compared to other memory models which relax one or more of these constraints.

2.2 Key insight: Classification of Memory Accesses

The key insight of the authors in [12] is that a memory access to a location which is private to a given thread can be successfully reordered with respect to a memory access to a location which is shared amongst multiple threads without violating SC; it is only for memory operations to shared data that ordering constraints need to be imposed. Most hardware designs to date have ignored this opportunity and have uniformly enforced memory model constraints upon all memory operations. If the runtime system can successfully classify memory accesses as to either private or shared data we could relax ordering constraints accordingly.

We can let the processor reorder an access if we can guarantee that there will be no conflicting access by another thread that could observe or alter the result of this access. Such an access is classified as *safe* while the remaining accesses are classified as *unsafe*. It is clear that all accesses to thread local data which is never accessed by any other thread are safe accesses.

Memory accesses can be classified as follows to ascertain at run time if an access is safe or not. A memory operation by a thread is:

- *Private* if it accesses a memory location that has only been accessed by that thread up to that point in the execution.
- *Shared read-only* if it accesses a memory location that has been read by multiple threads but not has been written since initialization.
- *Shared read-write* if it is accesses a location that has been accessed by multiple threads and least one of those accesses is a write.

It is clear that private and shared read only accesses are safe while shared read write accesses are *unsafe*. Thus, from the processors perspective, we now have four types of memory accesses:

Safe loads, Unsafe loads, Safe stores, Unsafe stores.

Along with these types, we will have sixteen different memory orderings (instead of four) but we only need to impose memory model ordering constraints between an *unsafe* [Load, Store] access and another *unsafe* [Load, Store] access and can relax the remaining twelve ordering constraints which have any access classified as safe [Load, Store]. If the proportion of safe accesses in an application is high, our design results in the relaxation of ordering constraints on a very large amount of the memory operations.

2.3 Untapped Opportunity

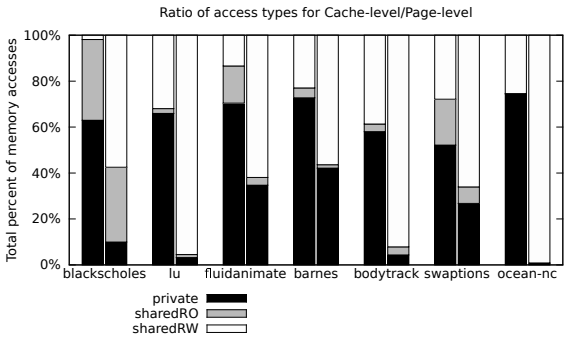

Figure 1: **Memory access type classification. Left bar is for cache-level, while the right for page-level**

Figure 1 shows the access type distribution of memory accesses in the executions of several programs. The classification is based on observing the behavior of the parallel section of the program (i.e. after main thread spawns the first thread) and tracking the types of memory accesses at page and cache block level. These results were obtained using PIN [10]. It can be seen from this graph that the amount of safe memory accesses when classified at cache block level is considerably greater than when memory accesses are classified at page level. This suggests an untapped opportunity at page level classification, wherein classification done at cache block level can achieve more safe accesses and could get better performance. It was this opportunity that we explored in our work.

3. Proposed Solution: *Reactive SC*

In this section, we describe our proposed solution of *Reactive SC*, the design and implementation of protocols employed, along with hardware modifications that are warranted.

3.1 Overview

Reactive SC classifies memory accesses on cache block granularity into safe or unsafe. While in [12] the authors employ the page table and TLB to classify accesses at page granularity, we leverage the directory in the cache coherence protocol to keep track of classification information. Since the coherence state machine uses the directory to keep track of coherence state for all cache blocks actively used by the processors, we add a few more state bits to each directory entry to hold the current classification of cache block.

3.2 Directory Protocol

Figure 3 shows the directory entry augmented to keep track of classification information. We keep the classification state machine separate from coherence state machine and they act as separate state

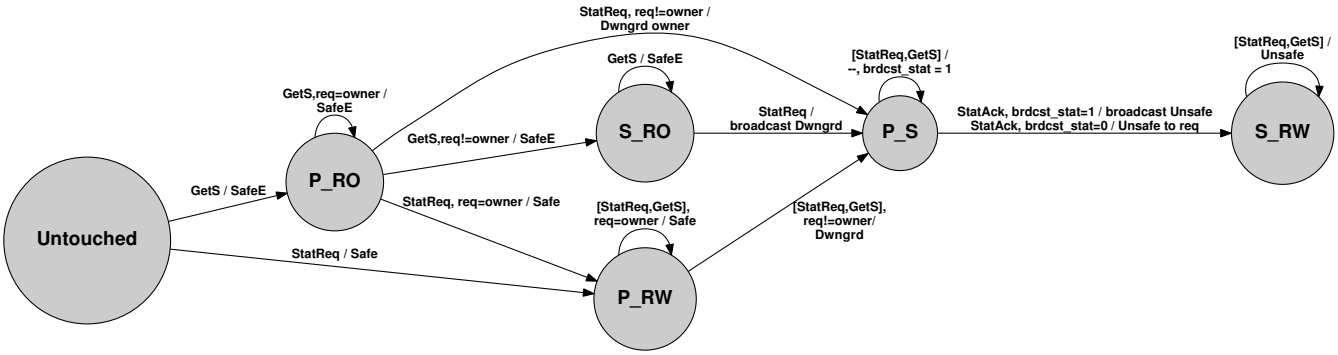

Figure 2: **Directory controller state machine**

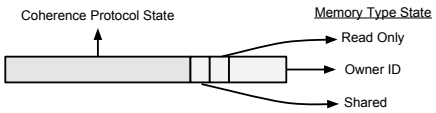

Figure 3: **Augmented Directory Entry**

machines running in parallel. Thus apart from coherence protocol state we add the following state to each directory entry:

- Read only: Single bit classifying entry as being only read so far.
- Shared: Single bit classifying entry as being shared by multiple processors. If the read only bit is set then cache block is Shared read only, otherwise the cache block is shared read write
- Owner ID: Bits keep track of current owner of the cache block. An owner of the cache block is the first processor to every access the cache block. Any subsequent access by any other processor will nullify this field. Thus owner of the cache block in the classification state machine is different from the concept of owner in a coherence state machine.

Figure 2 shows the directory controller state machine for classification. Our design tries to minimize the network messages injected by observing cache coherence messages and initiating state transitions. A cache block not being used by any processor is not present in the directory and is thus untouched. A Load by any processor will lead to a GetS cache coherence message sent to the directory. On observing this, the classification state transitions to P_RO (or Private read-only) state and the Owner ID field is set accordingly. The directory replies to the processor that the current classification for cache block is safe_exclusive. We differentiate a safe block from safe_exclusive so that processors intending to perform a Store are forced to handshake with the directory as explained in section 3.4. Any subsequent GetS from a processor which is not the owner leads to a transition to S_RO (or shared read-only) state. Since stores can silently happen in a MESI like protocol if the cache block is in exclusive state at the processor side, we could not leverage GetX coherence message in our state machine. Instead we have introduced StatReq message which is send by a processor intending to do a Store. Status-Req message in UNTOUCHED state leads to a transition to P_RW or private read write state and the directory replies to the processor that the classification is safe.

The state P_S is a transient state in our state machine that is used when a cache block transitions from safe to unsafe state ([P_RO, P_RW, S_RO] → S_RW). When a Store is being performed on a cache block which is in read only state (P_RO or S_RO) by a non-owner processor or a Load or Store is being performed by non-owner on cache block which is marked as P_RW, it implies that this cache block is shared amongst different processors. It is in this case that the directory should inform the processors actively using the cache block to drain their store buffers so as to hide any memory re-orderings they have performed under the assumption that the cache block was safe. The directory accomplishes this by sending a Status-Downgrade message to the concerned processors (owner for P_RO and P_RW and broadcast for S_RO). The processors receiving the downgrade message drain their store buffers and send a Status-Ack message to the directory. On receiving all the Status-Ack messages, the directory transitions to S_RW or shared read write state and replies to the original requestor that the classification is unsafe. Any subsequent GetS or Status-Req will be replied by the Directory as being unsafe.

While there is a cost associated with sending downgrades, draining of store buffers, and collecting acknowledgements, the important point to note is these transitions are rare, thus rendering this cost is negligible. Furthermore, it is cheaper to get the classification state information than to get coherence state information as the directory is the single centralized place to query classification information and thus armed with classification information we can relax ordering constraints imposed by memory model.

3.3 Cache Controller Protocol

To avoid always querying the directory for classification, we cache the classification state on to private caches of processors. This leads to addition of two bits per cache block as shown in Figure 6.

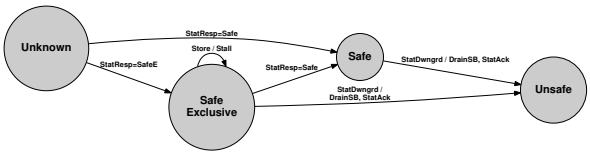

Figure 4: **Cache controller state machine**

Figure 4 shows the cache controller state machine. Responses from the Directory lead to transitions from one state to another. Loads get hits in either safe/safe_exclusive/unsafe state for classification state. Stores on the other hand get hits only in

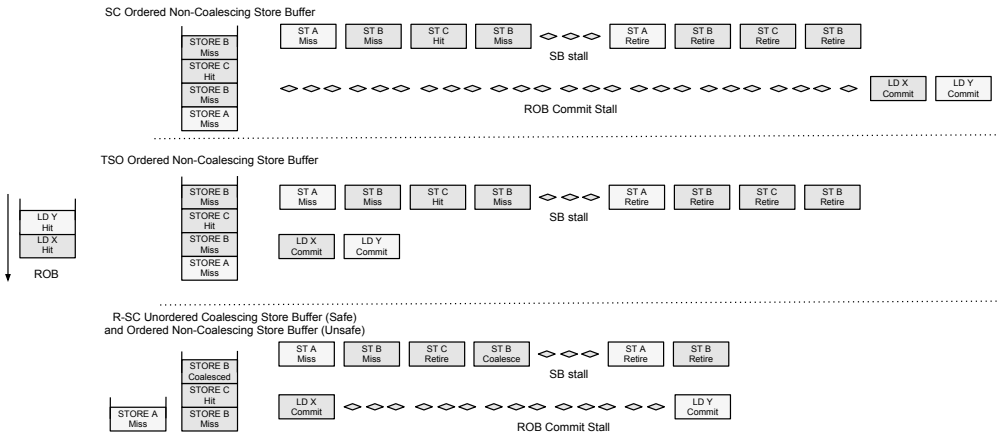

Figure 5: **An execution example**

safe/unsafe state and have to send a Status-Req message when cache block is in either unknown/safe_exclusive state.

3.4 Processor Modifications

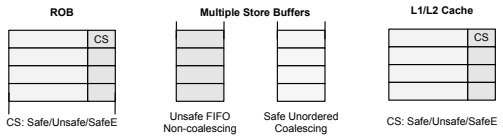

Figure 6: **Processor & Cache modifications**

We assume in window speculation as proposed in [7] which is used to optimize SC hardware. This lets stores to be retired from the ROB and placed in the store buffer which have to be retired in strict FIFO order. This helps SC hardware hide write miss latency to some extent. Figure 6 shows processor hardware changes required to support *Reactive SC*. We keep track of classification state in the ROB. *Reactive SC* hardware employs an additional un-ordered, coalescing store buffer for safe stores as in [12]. As pointed out in section 2.2 safe accesses can be reordered without violating SC and this unordered store buffer allows us to retire stores out of order and to coalesce stores that are to the same address.

The key invariant we need to ensure to guarantee correctness is our design is the following: on a classification state transition (safe to unsafe) any memory reorderings we have done should be committed before letting the memory operation causing the state transition to complete. We adhere to this invariant by making the following design decisions:

- A load/store is not retired from ROB till its classification state is known.
- On receipt of Status-Downgrade message, store buffers are drained and lsq is snooped to mark matching loads/stores as unsafe.

The above two design decisions, along with directory and cache controller state machines ensure that directory is always aware of loads/stores done by a processor to a cache block and thus has correct classification information and draining store buffers on state transitions ensures that any memory reorderings are committed safely before a conflicting access (the operation causing the state transition) is allowed to proceed.

The above design adds following features to baseline SC design:

```
STORE A (A: Unsafe)
STORE B (B: Safe)
STORE C (C: Safe)
STORE B
LOAD X  (X: Safe)
LOAD Y  (Y:Safe)
```

Figure 7: **Program Listing for sample execution in Figure 5**

- Store buffers with bypassing capabilities for safe loads.
- Store buffers with bypassing capabilities for unsafe loads (when FIFO buffer is empty).
- Overlapping and coalescing store buffer for safe stores.

In *Reactive SC* we are then left with following ordering constraints:

- Unsafe loads cause FIFO store buffer to drain.
- Unsafe stores retire out of store buffer in order.

In essence the orderings between unsafe accesses need to be enforced as pointed out in section2.2.

4. Execution Example

Consider an execution example in figure 5. The order of instructions is as follows:

We demonstrate three different executions: a SC execution, a TSO execution and *Reactive SC* execution. TSO (total store order) [8] memory model is used in SPARC implementations and appears to match the memory consistency model of the widely used x86 architecture. TSO essentially relaxes Load to Store ordering and thus a load behind a store in program order is allowed to complete while the store is pending. The figure presents the state of the execution when I4 is at the head of ROB ready to retire.

For the SC execution, Loads in ROB cannot retire till all pending stores are retired from the store buffer. The two stores to location B cannot be coalesced due to an interleaving store and Store C cannot retire even though it is a hit as stores are retired in order.

For TSO execution, except that both loads can retire as soon as they are at the head of ROB we still neither retire stores out of order nor coalesce stores.

Finally, in *Reactive SC* we employ two different store buffers with safe stores retiring to unordered coalescing store buffer and unsafe

stores retiring to ordered non-coalescing store buffer. Thus Store C can retire even though Store a is waiting on a cache miss and stores to B can be coalesced. LOAD X can retire as it is a safe load without requiring any store buffer drain. Since Load Y is unsafe we need to wait for Store A to retire but not the other stores as we need to preserve ordering only amongst unsafe accesses.

This example demonstrates that *Reactive SC* can relax more orderings than TSO but is still limited with store buffer drains for unsafe loads.

5. Experimental Setup

Processor core @ 2GHz	
Fetch/Exec/Commit	4 instructions (max 2 loads or 1 store) per cycle per core
FIFO Store Buffer	64 entry FIFO buffer
Unordered Store Buffer	8 entry unordered, coalescing store buffer
L1 Cache	64 KB per-core, 4-way set associative, 64B block size, 1-cycle hit latency, write-back, write up to 8 bytes per cycle
L2 Cache	512KB per-core, 4-way set associative, 64B block size, 10 cycle hit latency
Coherence	MOESI directory protocol
Interconnection	Torus-2D topology, 512-but link width, 8-cycle link latency
Memory	80-cycle DRAM lookup latency

Table 1: **Processor Configuration**

6. Experimental Results

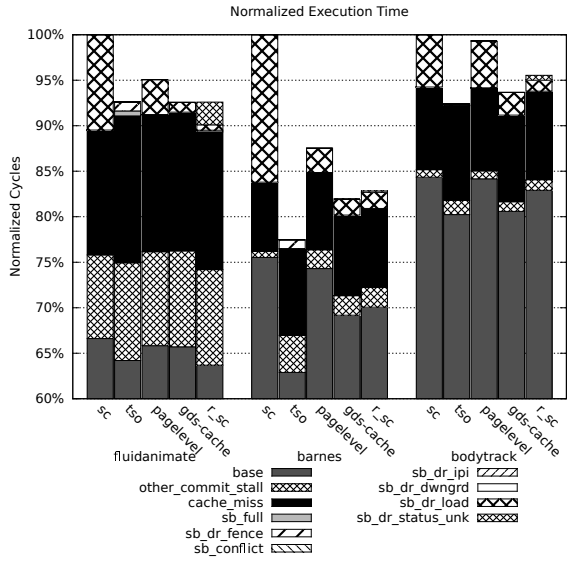

Figure 8: **Execution cycle stack with commit stage stall cycles**

Graph 1 shows normalized execution cycle stacks composed of stall cycles in the commit stage due to various reasons. By relaxing memory constraints on safe loads, *Reactive SC* is able to significantly decrease the number stall cycles due to loads waiting for store buffer drains. We were also able to achieve near TSO overall performance

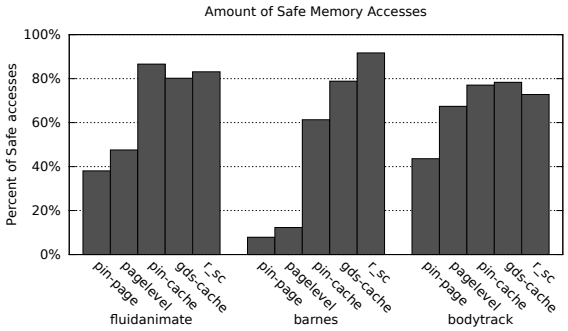

Figure 9: **Memory access type classification observed in pintool and simulations**

numbers while maintaining the programmability and intuitiveness of SC.

Our access type classification data obtained from FeS2 simulations was able to come reasonably close to the data observed from our initial pintool analysis. *Reactive SC* is able to classify a much greater number of safe accesses by using a finer granularity than the page level implementation.

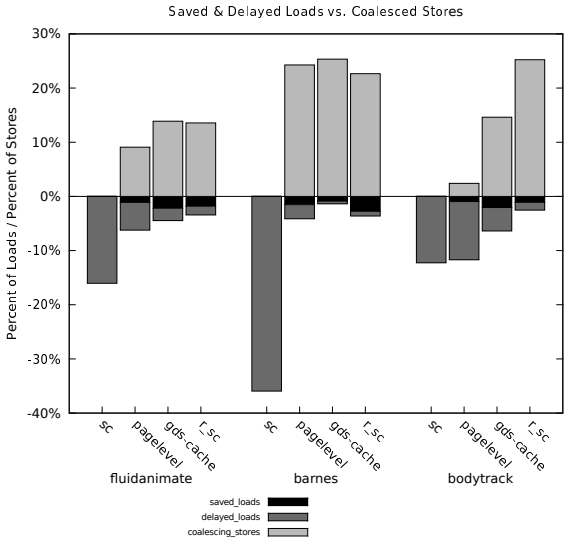

Figure 10: **Percent coalesced stores in upper half. Percent saved/delayed loads in lower half**

Sequential consistency has a large number of delayed loads due to store buffer drains and TSO does not allow for any coalescing of stores. Not only were we able to reduce the number of delayed loads from SC, we were also able to coalesce a significant number of stores. By doing this we are able to achieve better utilization of our store buffer. Our experimental results were able to demonstrate a substantial performance gain over both SC and the page level implementation.

7. Conclusions and Future Work

Having a clean memory consistency model which follows the intuition of the programmer is paramount to truly harnessing the power of multiprocessors. There is a pressing need to make multiprocessors more programmable to greater majority of

programmers. We see our work as a positive step in this direction. We have demonstrated that by dynamically classifying the memory accesses into different classes we can not only reduce the performance overhead of SC but also close the gap between SC and the TSO model. Our simulation results prove that we are within 1.1%-6.5% of TSO performance for the tested benchmarks.

Currently, once a cache block is classified as unsafe it will always be classified as unsafe until it is no longer present in directory. In future work we intend to explore reverse classification transitions i.e. unsafe to safe to see if certain applications can benefit from it.

8. Acknowledgements

We would like to thank Professor Thomas Wenisch and Joseph Greathouse for having an excellently organized class which helped bring out the best in us. Special thanks to Professor Wenisch for helping us improve the robustness of our experimental methodology. We also would like to thank Professor Satish Narayanasamy for his helpful discussions and contributions throughout the course of the project. Finally, we sincerely would like to thank Abhayendra Singh for helping us with infrastructure preparations and for his constructive feedback.

References

[1] S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. volume 29, pages 66 –76, dec 1996. doi: 10.1109/2.546611.

[2] S. V. Adve and M. D. Hill. Weak orderingà new definition. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 2–14, New York, NY, USA, 1990. ACM. ISBN 0-89791-366-3. doi: 10.1145/325164.325100. URL `http://doi.acm.org/10.1145/325164.325100`.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-282-5. doi: 10.1145/1454115.1454128. URL `http://doi.acm.org/10.1145/1454115.1454128`.

[4] C. Blundell, M. M. Martin, and T. F. Wenisch. Invisifence: performance-transparent memory ordering in conventional multiprocessors. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 233–244, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0. doi: 10.1145/1555754.1555785. URL `http://doi.acm.org/10.1145/1555754.1555785`.

[5] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. Bulksc: bulk enforcement of sequential consistency. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 278–289, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. doi: 10.1145/1250662.1250697. URL `http://doi.acm.org/10.1145/1250662.1250697`.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 15–26, New York, NY, USA, 1990. ACM. ISBN 0-89791-366-3. doi: 10.1145/325164.325102. URL `http://doi.acm.org/10.1145/325164.325102`.

[7] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *In Proceedings of the 1991 International Conference on Parallel Processing*, pages 355–364, 1991.

[8] S. International. *The SPARC architecture manual: version 8*. Prentice Hall, 1992. URL `http://books.google.com/books?id=Qo1QAAAAMAAJ`.

[9] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. volume C-28, pages 690 –691, sept. 1979. doi: 10.1109/TC.1979.1675439.

[10] 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. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065034. URL `http://doi.acm.org/10.1145/1065010.1065034`.

[11] P. Ranganathan, V. S. Pai, and S. V. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, SPAA '97, pages 199–210, New York, NY, USA, 1997. ACM. ISBN 0-89791-890-8. doi: 10.1145/258492.258512. URL `http://doi.acm.org/10.1145/258492.258512`.

[12] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Madanlal. End-to-end sequential consistency. ISCA '12, 2012.

[13] 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. ISBN 0-89791-698-0. doi: 10.1145/223982.223990. URL `http://doi.acm.org/10.1145/223982.223990`.