THE UNIVERSITY
of EDINBURGH

Efficient, Scalable, and Fair Read-Modify-Writes

Bharghava Rajaram

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2014

Abstract

Read-Modify-Write (RMW) operations, or atomics, have widespread application in (a) synchronization, where they are used as building blocks of various synchronization constructs like locks, barriers, and lock-free data structures (b) supervised memory systems, where every memory operation is effectively an RMW that reads and modifies metadata associated with memory addresses and (c) profiling, where RMW instructions are used to increment shared counters to convey meaningful statistics about a program. In each of these scenarios, the RMWs pose a bottleneck to performance and scalability. We observed that the cost of RMWs is dependent on two major factors – the memory ordering enforced by the RMW, and contention amongst processors performing RMWs to the same memory address. In the case of both synchronization and supervised memory systems, the RMWs are expensive due to the memory ordering enforced due to the atomic RMW operation. Performance overhead due to contention is more prevalent in parallel programs which frequently make use of RMWs to update concurrent data structures in a non-blocking manner. Such programs also suffer from a degradation in fairness amongst concurrent processors. In this thesis, we study the cost of RMWs in the above applications, and present solutions to obtain better performance and scalability from RMW operations.

Firstly, this thesis tackles the large overhead of RMW instructions when used for synchronization in the widely used x86 processor architectures, like in Intel, AMD, and Sun processors. The x86 processor architecture implements a variation of the Total-Store-Order (TSO) memory consistency model. RMW instructions in existing TSO architectures (we call them *type-1* RMW) are ordered like memory fences, which makes them expensive. The strong fence-like ordering of type-1 RMWs is unnecessary for the memory ordering required by synchronization. We propose weaker RMW instructions for TSO consistency; we consider two weaker definitions: *type-2* and *type-3*, each causing subtle ordering differences. Type-2 and type-3 RMWs avoid the fence-like ordering of type-1 RMWs, thereby reducing their overhead. Recent work has shown that the new C/C++11 memory consistency model can be realized by generating type-1 RMWs for SC-atomic-writes and/or SC-atomic-reads. We formally prove that this is equally valid for the proposed type-2 RMWs, and partially for type-3 RMWs. We also propose efficient implementations for type-2 (type-3) RMWs. Simulation results

show that our implementation reduces the cost of an RMW by up to 58.9% (64.3%), which translates into an overall performance improvement of up to 9.0% (9.2%) for the programs considered.

Next, we argue the case for an efficient and correct supervised memory system for the TSO memory consistency model. Supervised memory systems make use of RMW-like supervised memory instructions (SMIs) to atomically update *metadata* associated with every memory address used by an application program. Such a system is used to help increase reliability, security and accuracy of parallel programs by offering debugging/monitoring features. Most existing supervised memory systems assume a sequentially consistent memory. For weaker consistency models, like TSO, correctness issues (like imprecise exceptions) arise if the ordering requirement of SMIs is neglected. In this thesis, we show that it is sufficient for supervised instructions to only *read and process their metadata in order to ensure correctness*. 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. Our experimental results show that *SuperCoP* performs better than the existing state-of-the-art correct supervision system by 16.8%.

Finally, we address the issue of contention and contention-based failure of RMWs in non-blocking synchronization mechanisms. We leverage the fact that most existing lock-free programs make use of compare-and-swap (CAS) loops to access the concurrent data structure. We propose *DyFCoM* (Dynamic Fairness and Contention Management), a holistic scheme which addresses both throughput and fairness under increased contention. DyFCoM monitors the number of successful and failed RMWs in each thread, and uses this information to implement a dynamic backoff scheme to optimize throughput. We also use this information to throttle faster threads and give slower threads a higher chance of performing their lock-free operations, to increase fairness among threads. Our experimental results show that our contention management scheme alone performs better than the existing state-of-the-art CAS contention management scheme by an average of 7.9%. When fairness management is included, our scheme provides an average of 3.4% performance improvement over the constant backoff scheme, while showing increased fairness values in all cases (up to 43.6%).

Acknowledgements

First and foremost, I wish to thank my advisor, Dr. Vijayanand Nagarajan for his wonderful guidance and support throughout my PhD studies. Being from a more hardware-oriented background, Vijay inuited my potential in working on higher level architecture when he accepted to guide me for my PhD thesis. His pleasant and often optimistic demeanour cheered me through highs and lows during my studies. His ability of thoroughly analyze problems and their different dimensions in order to arrive at pragmatic yet optimal solutions still amazes me. Whenever I encountered a problem, he always stood by me and boosted my morale, as a result of which, I felt more encouraged and motivated work towards solving those problems. He also gave me me the opportunity to be his teaching assistant for two semesters. This gave me a lot of confidence in my continuing pursuit of joing and excelling in the academia. I can never thank him enough for his efforts in making me a sound researcher.

Secondly, I would like to thank Christian Fencsch, who was my second supervior. All the discussions I have had with him served to expand my understanding of the subject. He is truly a vast reservoir of knowledge. In the same vein, I would like to extend my thanks to Bjorn Franke, Mike OBoyle, and Murray Cole for the support they gave me during my PhD. Their support was invaluable. A special mention to Muray Cole for being in my doctoral dissertation and viva panel. His criticism was extremely useful and also enlightening.

I would like to thank Marco Elver, and Andrew Mcpherson, who woorked with me in 2 of the major contributions of my thesis. Both of them possessed the ability to neatly organize things to the finest detail, which greatly influenced me in my approach to my research career.

I would also like to thank me fellow countrymen, Karthik and Manik, with whom I spent most of my free time. Karthik introduced me to the city of Edinburgh on my arriva, and was with me through thick and thin for my first 2 years here. Just when I was feeling sad about his departure to the USA, working for Synopsys, Manik came in and gave me an opportunity to help him as Karthik helped me. I especially enjoyed all my cooking sessions with them, and with Manik, our shared passion in gardening, and gaming brought us closer.

I would also like to thank my labmates, Elliot, Yuan, Thibault, Siddharth, Vasseilious, Konstantina, Murali, and Kiran for their valuable help in cultivating my thesis to the form it is now. Last but not the least, I would like to thank my parents, Rajaram and Uma Rajaram without whom, I wouldnt be successful in my endeavours.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

- Bharghava Rajaram, Vijay Nagarajan, Andrew J. McPherson, and Marcelo Cintra. 2012. SuperCoP: a general, correct, and performance-efficient supervised memory system. In Proceedings of the 9th conference on Computing Frontiers (CF '12). ACM, New York, NY, USA, 85-94.
- Bharghava Rajaram, Vijay Nagarajan, Susmit Sarkar, and Marco Elver. 2013. Fast RMWs for TSO: semantics and implementation. In Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation (PLDI '13). ACM, New York, NY, USA, 61-72.

(*Bharghava Rajaram*)

Dedicated to my late mother Uma Rajaram.
Life feels empty without you.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Era of Multicore Processors

High Performance Computing (HPC) has been at the forefront of computer science research as a driver of advancement in scientific research, by performing complex simulations in the fields of fluid dynamics, genetics, nuclear reactions, etc. HPC has also found place in the financial world where large scale computing is required to perform several complex computations including market simulations, and predictive analysis. Increase in HPC is to be largely attributed to the increased performance of microprocessors over the last few decades, which has been driven by *Moore's Law* [Moore, 1965]. Moore's law is the observation that, the number of transistors that can be made available on silicon doubles approximately every 2 years. This period has further been decreased to 18 months [Schaller, 1997].

This increase in the availability of transistors has had researchers working endlessly on how to utilize the additional transistors in order to provide increased computing power. Initial performance gains were achieved via *frequency scaling*. However, increasing the frequency of operation of a microprocessor also leads to increase in power consumption (and consequently heat generation). Since there is a limit to the power budget of any silicon device, a *power wall* is reached, and processors have to rely on alternate architectures to keep providing increased performance. Another bottleneck to performance is the disparity in the speed of the processor and memory, otherwise called the *memory wall*. Indeed, from 1986 to 2000, processor speed improved at an

annual rate of 55% while memory speed only improved at 10% [Wulf and McKee, 1995].

Due to the scalability issues of both frequency and memory speeds, the additional transistor bounty was utilized to add additional hardware logic to a processor to exploit *Instruction-level Parallelism (ILP)* [Hennessy and Patterson, 2003]. ILP involves performing several operations of a computer program simultaneously. The techniques used to achieve ILP includes :- *instruction pipelining* - execution of multiple instructions are partially overlapped; *out-of-order processing* - instructions are made to execute in any order as long as they do not violate data dependencies; *supercsalar execution* - multiple execution units are used; branch prediction, speculative execution etc.

However, the diminishing returns of ILP [Wall, 1991] has led to the use of *thread-level parallelism (TLP)* [Herlihy and Shavit, 2008]. In the case of TLP, multiple processes run in parallel, operating on the same or different data. To architecturally exploit TLP, mainstream processor vendors have focused on using the available transistors to add multiple processing cores on the same chip, otherwise called *chip multiprocessors* (CMPs). Vendors like Intel, AMD, and Sun have already been shipping out processors with multiple cores since the early 2000s. Starting with dual-core processors, the number of cores on a single chip has been increasing exponentially (e.g. Intel's Xeon Phi, Sun Niagara, etc).

1.1.1 Limitations of Chip Multiprocessors

While CMPs do provide increased performance, they suffer from limitations caused due to the factors including the coherence protocol [Sorin et al., 2011], the underlying consistency model [Adve and Gharachorloo, 1995], communication between processors, and efficient synchronization. In this thesis, we address two of the major problems facing multicore processors in terms of both performance and programmability. They are as follows.

Synchronization in Multicore processors: The increased computing power of the CMPs can only be exploited with the design of parallel software, which involves parallelizing existing algorithms to run on multiple cores simultaneously. Researchers

have adopted several parallel programming models to enable this, the most common of which is the shared memory model [Culler et al., 1997]. In the shared memory model, all processing cores share the same global memory space, to which they can read and write asynchronously. In such a scenario, however, there will be concurrent accesses to the same memory address from multiple processors which race with each other. Efficient *synchronization* mechanisms [Mellor-Crummey and Scott, 1991] are required to facilitate *control* of these concurrent accesses through co-operation amongst the processors, in order for correct execution of a parallel program. Synchronization, while critical in assuring the correctness of multi-threaded programs, is also a bottleneck to performance scaling in multicore processors, as evidenced by several research works [Anderson, 1990; Attiya et al., 2011, 2006; Cederman et al., 2013a; Graunke and Thakkar, 1990]. *Indeed, efficient synchronization has been one of the key problems in the parallel computing domain since its inception.*

Scalability: Another problem plaguing multicore processors is their scalability with regard to both performance and fairness [Cederman et al., 2013a]. Limitations in parallelization and the communication performance between processors affects the increase in performance that can be achieved by increasing the number of processing cores. Also, since the different processors share the same memory space, contention amongst processors limits scalability as well. *It is preferable to decrease contention and increase the concurrency of a parallel program for better performance.* It is worth noting that synchronization performance also does not scale well with the number of processors.

Since efficient synchronization and improved scalability are crucial to getting the most performance out of a multicore processor, we also need tools to enable programmers to create, debug, maintain, or otherwise support other programs and applications. The difficulty in writing and debugging parallel code is a well known problem in parallel computing. Two of the main classes of programming tools which are required in order to write efficient and correct parallel programs are the compiler/interpreter and the debugger/profiler. Examples of debugging/profiling tools includes Valgrind [Seward and Nethercote, 2005], Intel Pin [Luk et al., 2005], Memwatch, mtrace [LLC, 2010].

These tools help increase reliability, security and accuracy of parallel programs.

Firstly, reliability includes finding deadlocks, memory errors, and other factors that cause lockups & crashes. Secondly, the security aspect of debugging involves finding memory and threading vulnerabilities that can be exploited by hackers. And finally, identifying memory corruption, and race conditions help increase the accuracy of parallel programs by eliminating erroneous results. Memory systems which allow for such debugging during runtime, by maintaining auxiliary information about each memory address, are otherwise called *supervised memory systems*.

1.2 RMWs are important but expensive

We observed that **Read-Modify-Writes (RMWs) are indispensable** to efficient synchronization [Attiya et al., 2011], memory supervision/debugging, and profiling. RMW primitives form the basis for several **blocking** and **non-blocking** synchronization mechanisms. This makes RMWs seldom avoidable in writing efficient parallel programs. While used primarily in synchronization applications, RMWs also serve other important applications. First of these are **supervised memory systems** used for debugging parallel programs. Here RMW-like operations are used to update a memory address and some auxiliary data associated with that memory address in an *atomic* fashion. The auxiliary data is used to store information about that memory address, which is used to alter program execution during runtime. Examples of debugging/profiling tools using supervised memory include Valgrind [Seward and Nethercote, 2005], Intel Pin [Luk et al., 2005], mtrace [LLC, 2010], etc.

RMWs are also used in updating statistics counters which are shared amongst several threads. We introduce these applications in a little more detail in the background chapter.

In each of the above applications, we observed that the performance of RMWs is a severe bottleneck, affecting both efficiency, and scalability. In the following section, we discuss why RMWs pose a serious challenge to achieving high performance and scalability of parallel programs. We then identify the issues related to the performance overhead of RMW operations in each of the application classes and present our approach to tackling these issues.

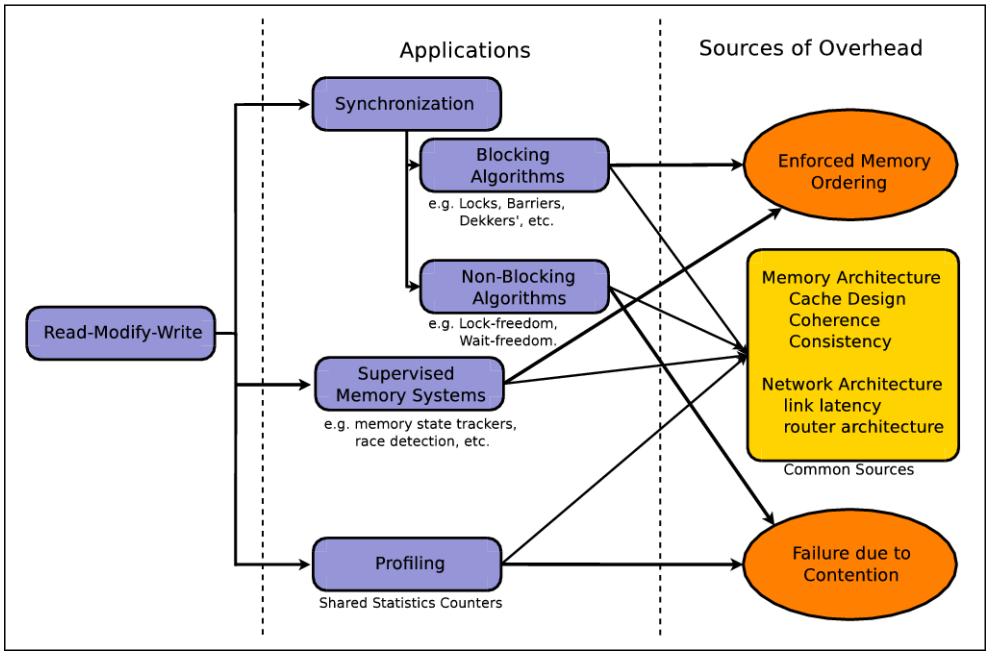


Figure 1.1: Applications of Read-Modify-Writes and performance issues associated with them. In this thesis, we address 2 sources of overhead, namely enforced memory ordering, and contention-based fails (in orange boxes)

The application of RMWs is summarized in Figure 1.1. It is apparent that RMWs are necessary in enabling effective and efficient parallel computing. We observe that RMWs are much more expensive then regular reads/writes in contemporary processor architectures which implement weaker consistency models. RMW operations in the Intel x86 architectures are particularly expensive and contribute significantly to the overhead of synchronization mechanisms. Measurements on a stock 8-core *Intel Sandybridge* processor, using the Splash-2 [Woo et al., 1995] benchmark suite showed that the **average latency of an RMW operation is 67 cycles**. In comparison, regular load and store instructions take around 4 cycles to complete [David Levinthal, 2009] (close to the L1 data cache latency). This latency is further hidden by the out-of-order execution model used by Intel processors. Similarly, on a 32-core Intel Xeon Processor, **the average latency of an RMW instruction is 74 cycles** across the SPLASH-2 benchmark suite. These measurements are further elaborated upon in chapter 3. In combination with a comparatively higher frequency of synchronization operations, the

large latency of RMW operations presents a performance bottleneck for parallel programs.

Indeed, Attiya et al. [Attiya et al., 2011] call for processor designers to design low cost RMWs, as concurrent programming is impossible without synchronization mechanisms, which primarily use RMWs. Several research works, like [Attiya et al., 2006; Michael, 2004; Natarajan and Mittal, 2014; Spear et al., 2008; Speziale et al., 2011], have proposed algorithms where they consciously reduce the number of RMWs in order to gain performance.

Also, the performance overhead of RMWs persists across its various applications. For example, in supervised memory systems, overhead of even over 100% can be expected for the purpose of maintaining metadata-data atomicity [Nagarajan and Gupta, 2009]. In profiling applications, per-thread statistics counters are preferred over shared counters (which need RMWs). Recent research by Dice et al. [Dice et al., 2013b] shows how RMWs make profiling expensive and also propose approximate counters to overcome limitations in performance and scalability.

Thus, in each of the above cases, there is a need to reduce the cost of RMWs. In this thesis, we investigate the causes for the increased overhead of RMW operations, and consequently propose solutions for the same. In the next section, we briefly discuss our observations on the reasons for why RMWs are expensive. We then discuss our approach to overcoming the bottleneck presented by RMW operations.

1.3 Why are RMW operations expensive

In this section we discuss why RMWs are expensive, contributing significantly to the overall program execution time. We discuss the 3 applications of RMWs, namely synchronization, supervised memory and profiling, separately, although the causes for the increased RMW cost may overlap with each other for every application.

1.3.1 RMWs in Synchronization

Similarly to non-atomic memory operations, the cost of RMWs depends on several design parameters. These include: the underlying memory architecture (cache size,

levels of cache, page size, etc.), the network architecture (like communication cost between processors, router architecture), cache coherence and consistency. While these factors are common to all memory operations, RMWs incur additional overheads due to its atomic nature and usage.

In this thesis, we focus on two of these *unique* sources of overhead with regard to RMWs used in synchronization. Firstly, we observe that there is a mismatch between the memory orderings required with an RMW and the orderings that are enforced by practical implementations of RMWs. We attribute this to how RMWs are defined with respect to their atomicity. Please note that this overhead occurs across all consistency models (which also enforce memory ordering), and thus can be classified as a separate source of overhead.

Secondly, we observe that contention amongst RMWs from different threads impedes the scalability of synchronization mechanisms. Since RMWs compete to update shared variables, some RMWs fail and have to be re-executed depending on the synchronization mechanism. We observe that these *contention-based fails* of RMWs contribute significantly to the overhead of concurrent programs.

We now discuss these causes in further detail. They are again described in an elaborate manner in the later chapters in this thesis (chapters 3,4,5), which address these issues with respect to the application of RMWs.

Overhead due to Memory Orderings enforced by RMWs: In the previous sections, we assumed a naive definition of atomicity in terms of non-interference from other processors. Further exploration into how atomicity is defined yielded the observation that an exact understanding of what kind of interference is allowed or disallowed greatly influences the cost of an RMW. This definition of atomicity, along with the underlying memory consistency model [Adve and Gharachorloo, 1995] restricts the ordering of memory operations around the RMW.

In this thesis, we specifically target the *Total-Store-Order* (TSO) consistency model. In the stronger *sequential consistency* (SC), all memory operations are strictly ordered, leading to poor performance in traditional implementations of SC. While recent efforts [Lin et al., 2010; Blundell et al., 2009] have strived to reduce this performance overhead, SC is still not existent in today's commercial processors. On the other hand, TSO is widely used in Intel's and AMD's x86 architectures, and also Sun's SPARC

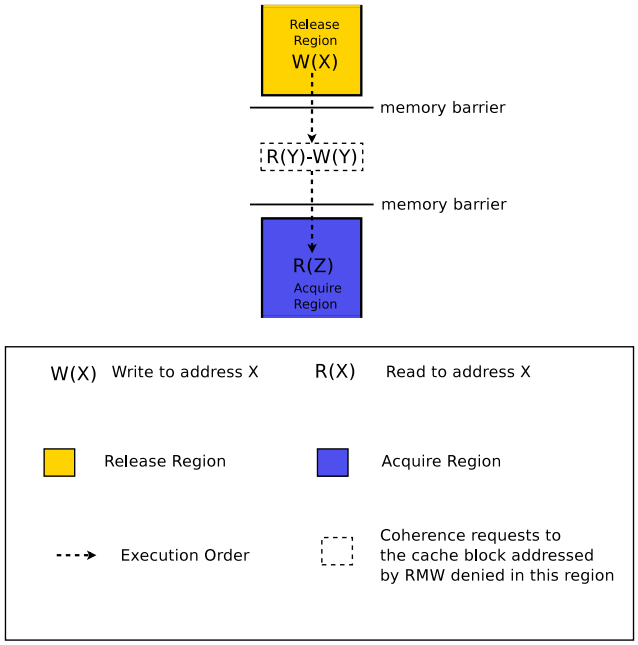


Figure 1.2: Existing RMW implementation in TSO - ordered like a memory fence.

processors. Also, as evidenced in the previous section, the cost of RMWs in TSO is manifold the cost of regular memory operations. Weaker consistency models, like release consistency (RC) [Mirapuri et al., 1992; Machine and Staff, 1995], do not enforce any memory ordering at hardware level, whilst relying on the programmer to insert appropriate memory barriers. While TSO is the focus of this thesis, we also show how to extend our work to weaker consistency models.

Existing RMW implementations in TSO require that the read and write of the RMW occur together in the global memory order [Intel Corporation, 2009; SPARC International, 1994; Gharachorloo, 1995; Sorin et al., 2011]. Since the underlying memory consistency model dictates the global memory order, the implementation of the RMW is also dependent on the memory model. TSO relaxes the $write \rightarrow read$ ordering and allows writes to be retired into the write-buffer; however, it requires writes to complete in program order. Therefore, TSO requires writes that are before the RMW in program order to be ordered before the write of the RMW. However, since read and write (of the RMW) should occur together in global memory order, writes before the RMW should also be ordered before the read of the RMW.

Thus existing RMW implementations first drain the write-buffer before performing

the RMW. To perform the RMW atomically, they read the cache block into the local cache with read-write permissions and deny coherence requests to the cache block until the write (of the RMW) completes. Only after the write completes, do they allow instructions following the RMW to be retired. In effect, existing RMW implementations implicitly insert memory barriers before and after the RMW as shown in Figure 1.2. This memory fence-like semantics of existing RMW implementations are the cause for RMWs to be expensive in architectures which implement TSO consistency. In this thesis, we investigate whether such a behavior is necessary for an RMW in order to provide the required ordering when used in synchronization.

Ordering Requirements of RMWs: While discussing the cost of RMWs, it is important to understand the functionalities that are required by an RMW, and the context in which it is used. For example, the RMWs used in synchronization are typically required to be strongly ordered [Boehm and Adve, 2008], i.e., they require acquire and/or a release semantics [Gharachorloo, 1995]. An *acquire* semantics requires that memory operations which follow the RMW should appear to perform after the RMW, while a *release* semantics requires that all memory operations which precede the RMW should appear to perform before the RMW. For instance, the RMW used in a lock operation requires acquire semantics since memory operations within the critical section should only be performed after the lock is obtained; conversely, an RMW when used in an unlock operation requires release semantics. In some situations, for instance in barrier synchronization, an RMW requires both acquire and release semantics.

In particular, our approach in this thesis is guided by the requirements of general programs, in particular by just what properties are needed for the C/C++11 implementation [Becker, 2011; C, 2011]. Recent studies [Terekhov, 2008; Batty et al., 2011] show how the new C/C++11 concurrency model can be implemented on a TSO architecture. They introduce synchronization reads and writes of various flavors; these reads and writes are referred to as atomics (*SC-atomic-reads* and *SC-atomic-writes*). Batty et al. [Batty et al., 2011] have shown that the C/C++11 model is correctly implementable on TSO by replacing C/C++11 SC-atomic-writes and/or SC-atomic-reads by RMWs, leaving other language constructs (reads, writes, fences) to be implemented by plain TSO reads, writes and barriers. The ordering that an RMW provides should

be sufficient in being able to replace the SC-atomic-reads and SC-atomic-writes.

Contention-based Failures: Apart from the increased cost of RMWs due to memory ordering constraints, we observed that contention also contributes significantly to the overall program execution time. Avoiding *contended hot spots* is a fundamental principle in the design of concurrent data structures [Herlihy and Shavit, 2008]. Both blocking and non-blocking algorithms suffer limitation to their scalability due to contention. In blocking algorithms, the amount of time a thread holds a given lock, and the granularity of the lock itself, have a significant impact on the overhead due to contention. In contrast, for non-blocking algorithms, we observe that the overhead due to contention is largely caused by the contention of the RMWs themselves; more specifically, the work wasted due to RMW failures [Dice et al., 2013a; Hendler et al., 2010; Morrison and Afek, 2013]. Consider lock-free algorithms for example. Most lock-free algorithms like [Cederman et al., 2013b; Michael and Scott, 1996; Herlihy, 1991] synchronize using read-modify-write (RMW) loops (particularly compare-and-swap (CAS) loops). In a CAS loop, a thread observes the shared state, performs a computation, and uses a CAS to update the shared state (called the *linearizing CAS*). If the linearizing CAS succeeds, this read-compute-update sequence appears to be atomic; otherwise the thread must retry the loop. The requirement of atomicity of the CAS loop implies that only one thread at a time can successfully complete a CAS loop. The remaining CAS loops will result in failures of the CAS. With increased contention, the number of CAS failures will increase resulting in poor throughput. This increased contention can be in the form of increased number of concurrent threads, or reduced amount of work done between successive synchronization operations. This is the main limiting factor in the scalability of non-blocking algorithms. Thus, instead of contention in itself, addressing the failures caused due to contention is an important challenge in increasing the performance of RMWs in synchronization.

Fairness issues due to contention-based failures: Apart from the performance overhead incurred due to contention-based failures, fairness is also of significant concern in concurrent programs especially under high contention. Fairness indicates how equal a chance do all threads have when accessing a shared resource. This can be

of concern particularly in NUMA architectures where the latency of memory access varies across different processing nodes. Also, low fairness indicates potential starvation of certain threads, or an unequal distribution of work. High fairness means that all threads have an *equal* opportunity to progress by accessing the shared resource.

Due to the progress guarantees of non-blocking algorithms, fairness is often compromised, where certain threads have more access to the shared resource (here a memory address accessed by an RMW) than others. In particular, the lock-free progress guarantee may lead to a thread failing all its CAS loops and never getting an opportunity to update the shared data structure. In fact, an extreme case of this is when all but one thread fail their CAS loops i.e. one thread continues to perform successful CAS loops while all other threads fail. This drastically affects fairness and even the overall performance of the program. In order to enforce a level of fairness in the program, it is necessary to ensure the fairness of RMWs performed by the concurrent threads. Thus, in non-blocking algorithms, it is important to address the issue of fairness along with performance in order to provide efficient synchronization.

1.3.2 RMWs in Supervised Memory Systems

Supervised memory systems make use of *supervised memory instructions* (SMIs) to access some (or all) of the memory addresses. As mentioned earlier, SMIs are similar to RMW operations, where they read the metadata associated with a memory address, and update it atomically if required. An SMI can also potentially raise an exception on reading metadata, if certain debugging criteria are met. Supervised memory systems support both supervised and non-supervised instructions.

Similarly to synchronization, supervised memory systems order SMIs to ensure correctness of supervision [Venkataramani et al., 2009]. Since any supervised instruction can cause an exception, all instructions should be ordered with respect to supervised instructions to guarantee precise exceptions [Bobba et al., 2011]. This ordering requirement leads to heavy performance degradation in weaker consistency models. The correctness problem in supervised memory systems and our approach to ensuring correctness without compromising on performance is elaborated upon in chapter 4.

1.3.3 RMWs in Profiling

While RMWs used in profiling applications have no specific ordering requirements, they are still afflicted by the issue of contention. In this vein, we address RMWs used in profiling in a similar manner as RMWs used in non-blocking algorithms. Please refer to chapter 5 for further discussion on this.

In summary, the two main factors which influence the performance overhead of RMWs are the memory ordering constraints and the contention amongst processors in accessing shared memory addresses via RMWs. While the issue of memory ordering is important in both blocking and non-blocking algorithms, the issue of contention-based RMW fails is more prevalent in non-blocking algorithms. This is due to increased contention as evidenced by existing literature [Cederman et al., 2013a]. Additionally, contention also degrades the fairness amongst threads in performing RMWs. Similarly supervised memory systems suffer a larger overhead due to the ordering requirement. Contention based fails do not occur in supervised memory systems, as the RMWs used always succeed. Contention however does play a role in deciding the communication cost involved in performing supervised instructions.

1.4 Thesis Contributions

This section describes the research contributions of the dissertation. Each contribution addresses the performance issues of atomic RMWs in both blocking and non-blocking synchronization mechanisms, by targeting the ordering constraints of RMWs or the bottleneck due to contention based fails. As discussed in the previous section, the performance overhead of RMWs due to their ordering constraints occurs universally in all applications of RMWs. However, ordering requirements vary for different applications. In this regard, we target RMWs used for synchronization purposes and RMWs used for supervised memory systems separately. RMW contention, however, is more prevalent in non-blocking algorithms (lock-free programs) and thus, we address the contention issue in this context alone. We also address contention in the context of

profiling applications

We briefly state the contributions in this thesis before discussing them in further detail:

- **Fast RMWs for TSO**: In this work we argue for weaker TSO RMWs, where atomicity is relaxed in order to avoid the memory barrier-like behavior of existing TSO RMW implementations (we call this *type-1 RMWs*). We observed that strong barrier-like semantics of RMWs is not often required in concurrent programs. We considered two weaker definitions: *type-2* and *type-3* each causing subtle ordering differences. Type-2 RMWs is a weakening of atomicity which only prevents *reads and writes of the same address as the RMW* from appearing between read and write (of the RMW) in the global memory order. Type-3 RMWs further weakens the atomicity by preventing only *writes of the same address as the RMW* from appearing between read and write (of the RMW) in the global memory order.

  We formally specify how such weaker RMWs would be ordered, and show that type-2 RMWs, in particular, can seamlessly replace existing type-1 RMWs in common synchronization idioms – except in situations where a type-1 RMW is used as a memory barrier. Recent work has shown that the new C/C++11 concurrency model can be realized by generating conventional (type-1) RMWs for C/C++11 SC-atomic-writes and/or SC-atomic-reads. We formally prove that this is equally valid using the proposed type-2 RMWs; type-3 RMWs, on the other hand, could be used for SC-atomic-reads (and optionally SC-atomic-writes). We further propose efficient microarchitectural implementations for type-2 (type-3) RMWs – simulation results show that our implementation reduces the cost of an RMW by up to 58.9% (64.3%), which translates into an overall performance improvement of up to 9.0% (9.2%) on a set of parallel programs, including those from the SPLASH-2, PARSEC, and STAMP benchmarks.

- **SuperCoP - Supervision for Correctness and Performance**: This work discusses the implementation of memory supervision with both correctness and performance i.e. a supervised memory system which maintains the memory ordering required for correctness, while not compromising on performance. The

current state-of-the-art solution to ensuring correctness of supervised memory systems is Bobba et al.'s *Safe and efficient supervised memory systems* [Bobba et al., 2011]. Bobba proposed *TSOall*, which orders all SMIs and is universally correct but inefficient. For better performance Bobba proposed *TSOdata*, which does not order non-supervised with respect to supervised instructions and vice versa. This solution is limited in its usage ,as discussed in chapter 4, despite its higher efficiency.

While TSOall requires all SMIs to perform (as a whole) *in-order* with respect to supervised and non-supervised instructions, 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.

We compare the performance of SuperCoP, TSOdata, and TSOall, using the HARD [Zhou et al., 2007] 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 though TSOdata is not applicable to all supervised systems.

- **DyFCoM - Dynamic Fairness and Contention Management**: We tackle the drawback in the existing RMW contention management schemes and the fairness degradation under contention by adopting a dynamic scheme which monitors the progress of the different threads through their RMWs, and uses this information to (a) tweak the contention-backoff parameters for optimal performance, and (b) throttle threads in order to ensure a high level of fairness. In this

regard, we present DyFCoM (Dynamic Fairness and Contention Management) – a holistic scheme which addresses both throughput and fairness in non-blocking lock-free programs, by monitoring RMWs used in such programs and controlling their execution. With regard to throughput, we apply a backoff whenever a CAS fails – similar to existing contention management schemes. However, unlike the existing scheme, DyFCoM dynamically determines backoff values, completely agnostic of the level of contention, concurrency, and the processor architecture. We do this by periodically monitoring the throughput of individual threads, and gradually change the backoff value based on this throughput.

Similarly, for fairness, we use the periodical throughput values to identify potentially starving threads and deliberately fail the linearizing CAS instructions of non-starving threads, increasing the chances for a starving thread to progress. We provide a simple implementation of both these features by augmenting the CAS instructions with the required monitoring, force-fail and backoff functions. Our experimental results show that DyFCoM performs better than the existing state-of-the-art CAS contention management scheme by an average of 7.93%. DyFCoM shows increased fairness values in all cases (up to 43.64%).

1.5 Organization of the thesis

This thesis is organized into six chapters and are as follows:

The current chapter (**chapter 1**) introduces the problem tackled by this thesis, namely high cost of RMW operations when used in synchronization, debugging/monitoring, and/or profiling. This chapter also provides an outline of the thesis.

Chapter 2 presents a an elaborate background on RMW operations, and discusses how they are used in synchronization mechanisms, both blocking and non-blocking. Chapter 2 also discusses the fundamentals of a supervised memory system. Also, as all the work presented in this thesis assumes TSO consistency, we provide a theoretical model of the memory orderings guaranteed by TSO consistency as part of the global memory order.

Chapter 3 discusses the weaker RMW instructions that we propose for TSO in order to reduce their performance overhead. We also present related experimental

results that validate our approach.

Chapter 4 presents our work on supervised memory systems. We first discuss the problems with correctness in supervised memory systems and then present our approach to ensuring correctness.

In **chapter 5**, we discuss our proposed dynamic fairness and contention management scheme that provides both increased performance and fairness for lock-free programs.

Finally, **Chapter 6** concludes and offers reflections on the research work presented in this thesis. We also discuss potential future directions for research in this area.

Chapter 2

Background

This chapter provides a bare bones introduction to synchronization, and Read-Modify-Write (RMW) instructions, followed by an overview of the memory consistency models in use today. The Total-Store-Order (TSO) model is further elaborated upon as all the work presented in this thesis targets the TSO consistency model. We then discuss the progress guarantees provided by the various synchronization procedures, focusing on lock-free programming. From the memory supervision perspective, we discuss how atomic operations form the basis of a generalized *supervised memory instruction* (SMI), and why atomicity is required for this case.

2.1 Synchronization in Multicore Processors

Attiya et al. [Attiya et al., 2011] identify two frequently used patterns of synchronization - *read-after-write* (RAW) and *atomic-write-after-read* (AWAR). The RAW pattern involves a process (or thread) writing to a shared memory address (say x), followed by the same process reading from a different shared address (y), without writing to y in between. Such a pattern checks for writes to y from other threads in order to implement important synchronization mechanisms like mutual exclusion. Dekker's algorithm is the most significant example of RAW synchronization.

The AWAR pattern consists of a process (or thread) reading from a shared memory address followed by the process writing to the same or another shared memory address, where the read and write operations are *atomic*. In this thesis, we focus on the AWAR

synchronization pattern which make use of read-modify-writes (RMWs). An example of the AWAR synchronization pattern is shown in Listings 2.1 and 2.2. The mechanism is explained further in §1.3. It is worth noting that the RAW synchronization pattern can also be transformed into AWAR and vice versa.

Another way to classify synchronization mechanisms is based on the thread *progress guarantees* that the mechanism can provide. In this regard, synchronization mechanisms can either be *blocking*, like locks, semaphores, barriers etc. or *non-blocking*, like lock-free and wait-free algorithms [Herlihy and Shavit, 2008]. An algorithm is non-blocking if the suspension of one or more threads will not stop the potential progress of the remaining threads. On the other hand, in blocking algorithms, if a thread holding a lock is suspended, other threads may have to wait indefinitely for this lock to be released. Newer synchronization mechanisms like transactional memory [Hammond et al., 2004] borrows from both blocking and non-blocking algorithms.

2.1.1 Synchronization is Expensive

Effective synchronization is crucial in assuring the correctness of multi-threaded programs. However, synchronization primitives also pose a bottleneck to performance scaling in multicore processors. This has been shown by several research works [Anderson, 1990; Attiya et al., 2011, 2006; Cederman et al., 2013a; Graunke and Thakkar, 1990]. Attiya et al. [Attiya et al., 2011] showed that *expensive* synchronization cannot be avoided in concurrent algorithms using RAW or AWAR synchronization patterns, especially in *weaker consistency models*. A consistency model dictates the order of memory operations from concurrent threads as it is observed globally. A recent study by Cederman et al. [Cederman et al., 2013a] showed the limitations in the scalability of concurrent algorithms using different synchronization mechanisms.

Synchronization performance is affected by the communication patterns exhibited by threads, the cost of instructions used in the synchronization process, and the cache coherence overhead. While the cache coherence overhead is not specific to synchronization (it affects all memory operations in general), and the communication patterns of synchronization are specific to the synchronization mechanism used, the cost of the instructions used in synchronization is a common cause for poor performance of syn-

chronization across different mechanisms. The RAW synchronization patterns make use of expensive memory fences or barriers, while AWAR makes use of *read-modify-write* (RMW) instructions. We observe that RMWs used in synchronization are expensive in weaker consistency models and significantly contribute to the overhead of synchronization concurrent programs.

In this thesis, **w**e propose to achieve efficient synchronization by optimizing the performance of RMWs in weaker consistency models. We focus on the *total-store-order* (TSO) consistency model which is prevalent in the popular x86 processor architectures designed by Intel and AMD. Sun's SPARC processors also make use of the TSO consistency model.

2.2 Read-Modify-Write Instructions

An RMW is as a special machine instruction which reads a value from an address, and replaces it with a modified value *atomically* [Intel Corporation, 2009; Herlihy and Shavit, 2008; Michael and Scott, 1995a; SPARC International, 1994]. Examples of RMW instructions in SPARC/x86 processor architectures include atomic swap (SWAP/XCHG), test-and-set (TAS), fetch-and-add (FAA), and compare-and-swap (CAS). Initial RMW implementations in bus based multiprocessors locked the system bus [Intel Corporation, 2009] during the execution of an RMW, thereby explicitly ensuring atomicity. Subsequent RMW implementations leverage the insight that it is sufficient for an RMW to be logically atomic [Michael and Scott, 1995a]. To this end, the RMW obtains exclusive permissions to the cache line addressed and locks it in the local cache. The cache coherence protocol ensures that coherence requests from other processors to this cache line are denied until the RMW completes and the lock is released. This is adopted in the present Intel[Intel Corporation, 2009], AMD[Advanced Micro Devices, 2009], and SPARC[SPARC International, 1994] processor architectures. Commercial processor architectures which have a relaxed memory ordering [Mirapuri et al., 1992; Machine and Staff, 1995] often use a Load-Linked / Store-Conditional (LL/SC) pair to implement an atomic primitive. The read and write of the RMW (R_a and W_a) [1]

[1]Please note that we use the terms R_a for the read part of an RMW, and W_a for the write part of the RMW throughout this thesis.

are implemented as two separate instructions, LL and SC, which loop indefinitely till the write is successful without interruption. All RMW primitives can be implemented using LL/SC primitives.

2.2.1 RMWs used in Synchronization

An RMW used in synchronization is typically a machine-level instruction where a processor reads and writes to the same memory address *atomically* i.e. without the interference of other processors. Typical RMW instructions used for synchronization in commercial processor architectures include *compare-and-swap* (CAS), *test-and-set* (TAS), *fetch-and-add* (FAA), and *swap*. Irrespective of whether the synchronization mechanism is blocking or non-blocking, RMW operations (otherwise called *atomic* operations), are predominantly used to achieve consensus in synchronization. Indeed, Herlihy [Herlihy, 1991] showed that the ability to read and write to an address atomically is critical to solve the *consensus* problem, which abstracts important synchronization problems. It is not only difficult, but also impossible to implement some of the synchronization constructs without RMW operations.

Now, RMWs are costly in mainstream processor architectures such as x86 and SPARC, which support (variants of) *Total-Store-Order* (TSO). RMWs in these architectures are ordered similarly to a memory barrier [Intel Corporation, 2009; SPARC International, 1994], incurring the cost of a write-buffer drain in the critical path. Even in weaker consistency models, such as release consistency [Machine and Staff, 1995], the RMWs are generally associated with memory barriers [Boehm, 2011]. Increased cost of RMWs impacts the synchronization cost negatively.

Listing 2.1: Acquiring a Spinlock - xchg is an atomic swap instruction provided by x86

```
1 locked:   ; Lock variable − locked = 1, unlocked = 0
2       dd 0
3 spin_lock:
4       mov eax, 1 ; EAX register set to 1
5       xchg eax, [locked] ; Atomically swap EAX and the lock variable
6       test eax, eax ; Self−test EAX − Sets zero flag if EAX = 0
7       jnz spin_lock ; Lock acquired if EAX = 0, If not redo loop
8       ret
```

Listing 2.2: Releasing a Spinlock

```
1 spin_unlock:
2       mov eax, 0 ; EAX register set to 0
3       xchg eax, [locked] ; Atomically swap EAX and lock variable
4       ret
```

Almost all practical implementations of locking mechanisms, including spinlocks, reader-writer locks, ticket locks make use of RMWs. For example, the acquire part of a spinlock for an Intel x86 processor architecture [Intel Corporation, 2009] is shown in Listing 2.1. Here, the EAX register is set to 1 and is atomically swapped with the *lock variable* using the *xchg* instruction. This stores 1 in the lock variable, whose previous value is stored in EAX. If this previous value was 0, the lock was free and has now been acquired. If the previous value was 1, the lock is held by another thread, and this thread retries to acquire the lock. Listing 2.2 shows how a lock is released. The EAX register is set to 0, and is atomically swapped with the lock variable, thereby resetting the lock. Once a thread acquires a lock, it can execute its critical section. The locking mechanism ensures that only one thread at a time can enter its critical section. If the atomic *xchg* instruction is replaced with a regular load and store, then other processors can potentially write to the locked variable between the load and store. Thus, the value returned by the load instruction can be stale, resulting in multiple processors potentially acquiring the lock. It is worth noting that the RMW used in releasing of a lock, can be replaced by a normal write operation in consistency models that ensure write ordering (with respect to reads and writes). This includes the *total-store-order* (TSO) consistency model that is implemented by x86 processor architectures. However, locking mechanisms used in popular libraries, like pThreads, used RMWs in both the acquire and release portions of a lock.

Listing 2.3: Pseudocode for enqueue operation

```
1 enqueue(queue_t *Q, data* value) {
```

```
2   node = new_node();
3   node->value = value;
4   node->next.ptr = NULL;
5   while(1) {
6     tail = Q->Tail;
7     next = tail.ptr->next;
8     if(tail = = Q->Tail) {
9       if(next.ptr = = NULL) {
10  if(CAS(&tail.ptr->next, next, node))
11    break;
12  else
13    CAS(&Q->Tail, tail, next.ptr);
14      }
15    }
16  }
17  CAS(&Q->Tail, tail, node);
18 }
```

Similarly to lock-based synchronization, lock-free and wait-free implementations of standard data structures like queues, stacks, hash tables etc. also use RMWs to complete their synchronization process. The pseudocode for the *enqueue* operation of a lock-free algorithm for a queue [Michael and Scott, 1996] is shown in Listing 2.3. Here, the node to be inserted into the queue data structure is created in Lines 2-4. Lines 5-16 form a *CAS loop*, which repeats indefinitely until the enqueue operation is successful. Inside the loop, first the tail node and the address of the node pointed to by the tail node (Lines 6,7) are read. Then, the validity of the tail node (Lines 8,9), and whether it has changed since the enqueue operation began are checked. Then the tail node is updated atomically using a CAS (Line 10). If the update is successful, the enqueue is complete. If not, the CAS loop repeats. The CAS is necessary here so that only one processor at a time can update the queue. All other processors which attempt to update the queue fail, can retry the operation.

2.2.2 RMWs used in Profiling Parallel Programs

RMWs are used in tools [Levon and Elie, 2004; Seward and Nethercote, 2005] and applications [Afek et al., 2012; Lev and Moir, 2011] which make use of shared counters to maintain certain statistics related to parallel programs. Statistics counters are used to count events that may occur with high frequency while the value of the counter is read infrequently, as is common for performance monitoring and diagnostics.

Simply incrementing a shared counter without ”synchronization” does not work for multiple threads, because one threads update can overwrite anothers, thereby losing the effects of one or more increments. Such counters can be made thread-safe by protecting them with a lock, but in most modern shared memory multiprocessors, it is preferred to increment the counter using RMW instructions like CAS, and FAA.

2.2.3 Historical Definition of RMW instructions

Gharachorloo [Gharachorloo, 1995], in his research on memory consistency models, defined RMW instructions as *aggressive* and *conservative*. The *aggressive* definition says that for an RMW to be atomic, there should be no writes to the location addressed by the RMW. For the *conservative* definition, no writes should be allowed between the read and write of the RMW i.e. R_a and W_a. We observe that these definitions are not consistent with the implementation of RMW in various commercial processors.

Sorin and Mark Hill [Sorin et al., 2011], in their primer on consistency models, define that an RMW is atomic as long as the read and write occur in the global memory order [Lamport, 1978]. We adopt a similar definition methodology for an RMW, with which we then explain the implementation of RMWs in different consistency models.

2.3 Overview of Memory Consistency Models

2.3.1 Sequential Consistency

Arguably the most intuitive memory consistency model is sequential consistency. Sequential consistency was first formalized by Lamport [Lamport, 1979]. Lamport first called a single processor (core) sequential if the result of an execution is the same as

if the memory operations had been executed in the order specified by the program. A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all processors (cores) were executed in some sequential order, and the operations of each individual processor (core) appear in this sequence in the order specified by its program. Due to the strict ordering constraint, traditional sequentially consistent systems [Adve and Gharachorloo, 1995] do not make use of a write-buffer. There are no practical implementations of SC ordering due to the large overhead due to strict ordering. Although recent proposals [Blundell et al., 2009; Lin et al., 2010] show that sequential consistency can be implemented with minimal overhead, these have not been considered for practical implementation as of yet. We assume that sequentially consistent systems have strict intra-processor ordering between all memory operations.

RMW instructions in Sequential Consistency: Since sequential consistency does not make use of a write-buffer, and strictly enforces all memory ordering, W_a is executed right after the R_a completes. any memory operation before the RMW has to be completed before the R is performed. Any memory operation following the RMW in program order, has to be retired only after the W completes.

2.3.2 Total Store Order

Total-Store-Order (TSO) is used in SPARC implementations and, more importantly, approximately matches the memory consistency model of the widely used x86 architecture. TSO systems retire writes to a write-buffer, where all writes complete in FIFO order. Subsequent reads can be retired before the write completes. Thus TSO sacrifices read-write ordering. Also, reads which are to an address for which a write exists in the write-buffer, obtains its value from the corresponding entry in the write-buffer. The TSO model is further elaborated upon later in this chapter. The behavior of RMWs in TSO is also discussed.

2.3.3 Relaxed Memory Order

In a yet weaker or relaxed memory order (RMO), the architecture in itself does not enforce any ordering between any memory instructions. RMO systems preserve only the ordering that is explicitly specified by the programmer with the use of fences.

Writes retire to the write-buffer, but are not constrained to complete in-order. Thus, writes complete out of order. Also, reads can be issued out of order, and can also be completed out of order with respect to writes. This creates the need for fences

RMW instructions in RMO: RMO systems does not impose any ordering constraint on RMWs. Practical RMO systems, like PowerPC and MIPS, typically use LL/SC primitives, which allows reads/writes from the same processor to complete between LL and SC. However, due to the mechanism of LL/SC, reads following the RMW cannot retire till the SC completes, as success of the LL/SC depend on the success of the SC. The SC can, however, retire before the writes in the write-buffer if there are no explicit fences inserted by the programmer. In the atomic_ops library built by HP labs [Boehm, 2011], the LL/SC pair in PowerPC, uses the *lwsync* (lightweight sync) instruction whenever ordering is required.

2.4 Total Store Order Consistency Model

We elaborate on the TSO consistency model as used by x86 processor architectures as this forms the basis for our proposal on weakening RMW operations in order to provide better performance.

2.4.1 Base TSO

The base TSO model presented here follows Alglave [Alglave, 2010]. We present here only a brief introduction to Alglave's formulation published previously. Readers, particularly those familiar with alternative TSO formulations, should refer to Alglave's thesis for more details. The thesis has a proof of equivalence with the SPARC definition of TSO [SPARC International, 1992], which is separately shown by Owens et al [Owens et al., 2009] to resemble the x86 multiprocessor model.

As usual in axiomatic memory models, we first derive a set of candidate executions from a program. Each candidate execution contains a set of events and relations over them, and represents a conceivable execution path (with control-flow unfolded, and values for each read in the program). In the next step, the memory model will carve out (via conditions on those relations) which of these candidate executions are allowed

by the model.

The events (memory reads, writes, and barriers) are annotated with their thread, type, and for memory accesses the associated address and value. From the program we derive the *program order* (po) relation, a local (per-thread) total order over events from the same thread as they appear in the program. We also consider two relations which are existentially quantified over: a *reads-from* map (rf) and *write-serialization* (ws), both relations over events. The relation rf maps, for each read, the write that the read takes its value from to the read. The relation ws is a linear order per location relating all (and only) the writes to the same location, and represents the coherence order of the system (in prior work, this relation is also called *coherence* co).

For ease of stating the memory model, we derive various additional relations from the above. The *from-reads* relation (fr) relates a read to all writes to the same location that come after (in ws) the write it reads from (given by rf). The *external-reads-from* relation (rfe) is the subrelation of rf which is restricted to reads which read from a different-thread write. The *communication* relation com is the union of ws, rfe, and fr.

A *preserved-program-order* relation (ppo) relates all memory operations from the same thread in program order, according to TSO ordering rules. Thus it relates all memory operations, except writes to program order-subsequent reads: In other words, $W \xrightarrow{\text{po}} W$, $R \xrightarrow{\text{po}} W$, $R \xrightarrow{\text{po}} R$ all belong to ppo also.

A *barrier-separated* relation (bar) relates memory operations (on the same thread) separated in program order by a memory barrier.

The behavior of a program is the set of corresponding execution witnesses which are valid. A valid execution witness is one where the union of com, ppo, and bar is acyclic, and satisfies the uniproc condition. The uniproc condition states that the relation com is consistent with the per-thread order of memory operations to the same location. The first condition says that a happens-before-like relation is acyclic. In this case we call a linear extension of com, ppo, and bar the global-happens-before relation (ghb). Informally, it is the global memory order (also known as execution order) in which memory operations appear to perform.

Finally, the *global-happens-before* relation ghb is a total order which is a linear extension of tso. Informally, it is the global memory order in which memory operations appear to perform. It is worth noting that every execution witness can have multiple

legal global memory orders.

2.5 Progress Guarantees in Synchronization

Understanding progress guarantees provided by synchronization mechanisms is important when optimizing the mechanism for performance or other parameters. Progress of a thread is indicated by a thread successfully performing its synchronization operations. For example, a thread which successfully obtains a lock and then releases it is deemed to have progressed. Any modification that is made to the synchronization primitives should not compromise on the progress guarantees as it is vital to the pattern of synchronization.

Traditionally, synchronization mechanisms have been classified as either *blocking* or *non-blocking* based on their progress conditions. We now elaborate on each of these progress conditions while also providing practical implementations of the same.

2.5.1 Blocking Progress condition

A blocking progress guarantee , such as locks, semaphores or barriers, are simple to define in that when a thread which is in the middle of performing its synchronization crashes or is suspended indefinitely, then other threads may have to wait indefinitely in order to complete their own synchronization primitive. For example, consider the case of a lock as shown in Listings 2.1 and 2.2. Assume that a thread t_0 obtains the lock by setting the lock variable. Any other thread attempting to obtain this lock will be unsuccessful and keeps retrying to obtain the lock. Now, if t_0 crashes without releasing the lock i.e. resetting the lock variable, then other threads which try to get the lock will see that the lock variable is set. This will cause these threads to assume that the lock is still being held by another thread (t_0) and will continue to retry acquiring the lock. However, since t_0 has crashed, the lock variable is never reset thus causing the concurrent threads that are attempting to obtain the lock to spin indefinitely trying to get the lock. In effect, such threads will never progress and thus are blocked by crashed thread.

Blocking a thread is undesirable for many reasons. An obvious reason is that while

the thread is blocked, it does not perform any meaningful task. If the blocked thread was performing a high-priority task, halting its progress is not preferable. Also, certain interactions between locks can lead to error conditions such as deadlocks or livelock.

2.5.2 Non-blocking progress condition

Non-blocking progress conditions overcoming the main issue in blocking synchronization by ensuring that a suspended or crashed thread does not affect the progress of other threads. There are three non-blocking progress guarantees which vary in the strictness with which they allow other threads to proceed. They are (a) wait-freedom, (b) lock-freedom, and (c) obstruction-freedom.

2.5.2.1 Wait-freedom

Wait-freedom is the strongest non-blocking guarantee of progress, which ensures individual thread progress. This combines system-wide throughput with starvation-freedom. An algorithm is wait-free if every synchronization operation completes in a bounded manner i.e. the number of steps in the algorithm to complete an operation is bounded. This property is critical for real-time systems in order to guarantee bounded completion.

The flip side to wait-freedom is that its implementation is quite complex and induces a large performance overhead [Herlihy, 1991]. This is because a system-wide progress guarantee requires that each of the threads collaborates with remaining threads and help each other progress instead of performing its own operations.

2.5.2.2 Lock-freedom

Lock-freedom allows individual threads to starve but guarantees system-wide progress. An algorithm is lock-free if it satisfies that when the program threads are run sufficiently long at least one of the threads makes progress i.e. can complete its synchronization operation. All wait-free algorithms are lock-free. In fact, a lock-free algorithm which is free of starvation is wait-free.

Lock-free algorithms are more prevalent in literature as compared to wait-free algorithms. This is due to their less complicated structure, and lesser performance overhead

as compared to wait-free algorithms. A Lock-free algorithm typically implements its synchronization operations as part of a compare-and-swap (CAS) loop where a thread reads the state of the shared data structure, and updates the data structure if no other thread has updated it in the meantime. This structure of a lock-free program is presented in greater detail in chapter 5.

2.5.2.3 Obstruction-freedom

Obstruction-freedom is the weakest non-blocking progress guarantee, which ensures progress of a thread when all other concurrent competing threads are suspended. This means that an algorithm is obstruction-free if at any point, a single thread executed in isolation for a bounded number of steps will complete its synchronization operation. All lock-free algorithms are also obstruction-free.

2.6 Fair Synchronization

The fair synchronization problem is to design an algorithm that guarantees fair access to a shared resource among a number of concurrent threads. *Fair access* means that a thread cannot be allowed to starve i.e. not progress, or to progress rapidly by having a relatively higher chance of performing its synchronization operations. Fairness can be a measured quantity, or a progress guarantee of its own. Taubenfeld's work on fair synchronization [Taubenfeld, 2013] states the case for *strict fairness*, where no process can access a shared resource twice while some other process is kept waiting. There is no limit on the number of processes that can access a resource simultaneously. With such a requirement, threads which successfully perform their CAS loops should be made to wait until all threads which failed their CAS loops are allowed to access the shared object at least once. Taubenfeld attacks this problem in a seemingly blocking manner. However, he assumes a progress condition which says that in the absence of process failures, if a process is trying to complete its CAS loop, then some process, not necessarily the same one, eventually completes its CAS loop. Apart from lock-freedom, this progress condition also requires *deadlock-freedom* and *livelock-freedom*. Indeed, Taubenfield's approach to solving fairness works under the assumption of a benevolent scheduler where lock-free algorithms behave in a wait-free manner.

Unlike the strict fairness mentioned above, weaker alternatives are also important practically, especially in NUMA architectures [Cederman et al., 2013a]. As NUMA architectures are becoming the standard in the microprocessor industry, fairness of synchronization constructs is becoming more important. Possible differences in the access latencies of competing threads for a memory location may even lead some of them to starvation. Also, some threads can dominate others, by obtaining access to the shared object several times consecutively, not allowing other threads to progress. Although such an execution still guarantees lock-freedom, fairness is compromised. A relevant definition of fairness was introduced into this context by Ha et al. [Ha et al., 2007] comparing the minimum number of operations a thread had with the average number of operations of all threads. This helps distinguishing cases of starving or less served threads. For identifying the opposite cases we can compare the average number of operations with the maximum ones among the threads. This helps to identify faster or *run-ahead* threads. Since our goal is to address any unfair behavior, we use the minimum of the two fairness measures (for starving and run-ahead threads) as the overall fairness. Formally:

$$fairness(\Delta t) = min\left\{\frac{N.min(n_i(\Delta t))}{\sum_i n_i((\Delta t))}, \frac{\sum_i n_i((\Delta t))}{N.max(n_i(\Delta t))}\right\}$$

where $n_i(t)$ is the number of successfully performed operations by the thread i, in the time interval t. Fairness index values close to 1 indicate fair behavior, while lower values imply the existence of a set of threads being treated differently from the rest. The fairness index achieves value 1 when all the threads perform equal number of operations, i.e. perfect fairness. The fairness index is 0 when at least one thread completely starves. Several other fairness measures can be used, as mentioned by Jain et al. [Jain et al., 1998]. We chose to go with the fairness measure described above due to its simplicity and that it captures both extremes of starving and run-ahead threads.

2.7 Supervised Memory Systems

Memory supervision or Supervised Memory Systems refers to those techniques which utilise metadata associated with every memory address, in order to provide support for applications including race detection [Zhou et al., 2007], memory checking [Venkataramani et al., 2009], deterministic processing [Devietti et al., 2009], empty/full bits [Agarwal et al., 1995; Alverson et al., 1990], record/replay [Narayanasamy et al., 2005], information flow tracking [Qin et al., 2006; Suh et al., 2004; Venkataramani et al., 2008] and transactional memory [Bobba et al., 2008]. In recent years, there has been renewed interest in such memory systems. This additional data, or *metadata*, which is used to store auxiliary information about the program memory, 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 2.1. 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 [Alverson et al., 1990; Suh et al., 2004; Venkataramani et al., 2009; Zhou et al., 2004, 2007], transactional memory [Bobba et al., 2008], fine-grained synchronization [Zhu et al., 2007], and deterministic processing [Devietti et al., 2009]. Supervised memory systems have become increasingly attractive with the emergence of multicore and manycore architectures which pose challenges in programmability and reliability.

Metadata is accessed and processed atomically with program data as shown in Figure 2.1. 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 sup-

Supervised Memory Read (SMR)	Supervised Memory Write (SMW)

Atomic
{
Read Data (R_d)
 	
/* Read & process Metadata */
Read Metadata (R_m)
if (Metadata == ...)
exception()
Metadata = f(Metadata)
 	
Write Metadata (W_m)
}
	Atomic
	{
	/* Read & process Metadata */
	
	Read Metadata (R_m)
	if (Metadata == ...)
	exception()
	Metadata = f(Metadata)
	
	Write Metadata (W_m)
	Write Data (W_d)
	}

Figure 2.1: Supervised Memory Instructions

port such supervised memory instructions (SMIs) are known as *supervised memory systems*. As shown in Figure 2.1, each SMI is an RMW.

2.7.1 Metadata and SMIs

For simplicity, we use the full-empty bits supervision scheme [Agarwal et al., 1995; Alverson et al., 1990] 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.

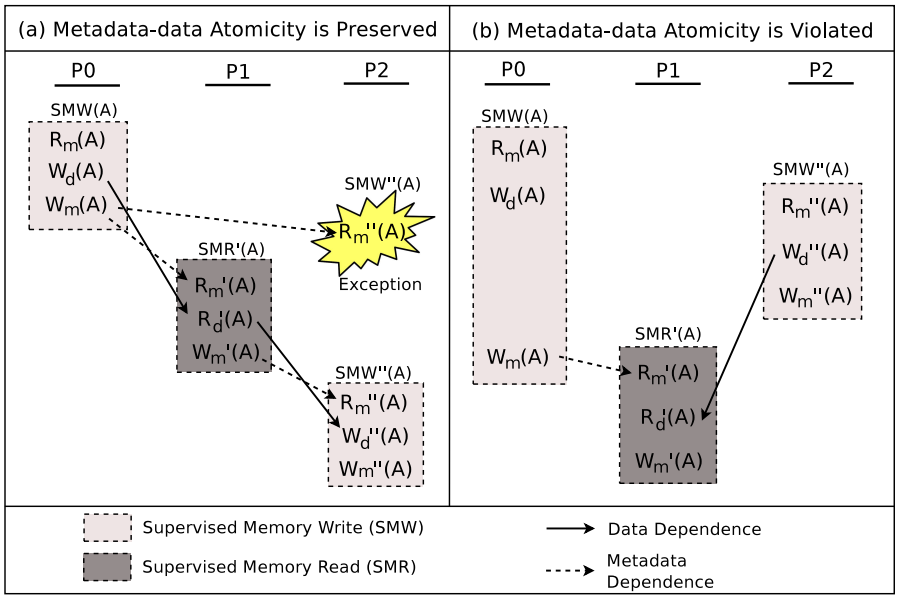


Figure 2.2: This figure shows the need for metadata-data atomicity for supervised memory systems. Metadata of A is initially *empty*. If metadata-data atomicity is preserved, SMR′(A) reads the value written by SMW(A). This is followed by SMW′′(A), which updates A. *The exception raised by SMW′′(A) prevents 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).

2.7.2 Atomic operations in Supervised Memory Systems

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. This means that a supervised memory operation has to be completed in an atomic fashion as there may be concurrent metadata accesses from other processors. This is done to ensure that any supervised instruction gets the most recent value of metadata for its processing. The formulation of a supervised memory operation (Figure 2.1), along with the requirement for atomicity, shows that all supervised accesses to memory are treated similarly to RMW instructions. There are, however, supervision schemes which do not require RMWs for metadata access (e.g. DIFT).

Consider Figure 2.2, where both T_0 and T_2 perform an SMW to address A, while T_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 of SMW(A) and SMW$''$(A) cannot interleave with each other. This is illustrated in Figure 2.2, where SMW(A) completes first, following which T_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 T_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. 2.2, both SMW(A) (from T_0) and SMW$''$(A) (from T_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.

2.7.3 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[Nethercote and Seward, 2007a; Newsome and Song, 2005; Nethercote and Seward, 2007b], 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 [Nagarajan and Gupta, 2009]. Other atom-

icity schemes include the use of transactional memory [Chung et al., 2008], where supervised instructions occur 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% [Nethercote and Seward, 2007a]

(Decoupled supervised memory systems) In decoupled systems described in [Chen et al., 2006; Kannan, 2009; Vlachos et al., 2010], 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 [Vlachos et al., 2010].

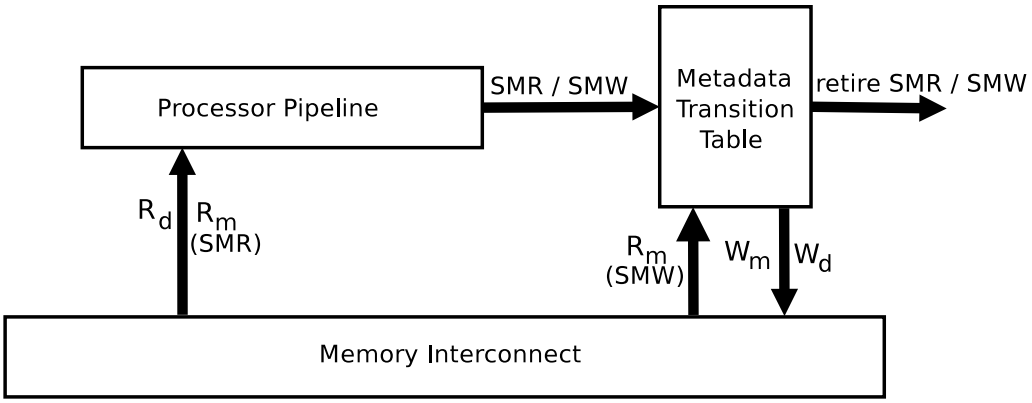


Figure 2.3: Memtracker implementation of a supervised memory system. An SMI (SMR/SMW) can be retired only after its metadata and data operations are completed.

(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* [Qin et al., 2006; Suh et al., 2004; Venkataramani et al.,

2008, 2009; Zhou et al., 2007], 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**[Venkataramani et al., 2009], 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 2.3. *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 **inorder**. 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.

In terms of metadata storage model, supervised memory systems are categorized as either shared, or interleaved. In a shared metadata memory model, metadata and data share the same memory space. For every data address, there is a corresponding metadata address. Also, multiple data addresses can share a single metadata address when metadata for a number of addresses fit into the data word length of each memory location. In an interleaved metadata memory model, the memory word length is extended to fit the metadata for every address along with the data. Another possible metadata storage model stores metadata in a separate cache. Since this model is wasteful of space and energy, it is not considered for practical deployment.

2.7.4 Supervised Memory Systems for Weaker Consistency Models

Typically, memory supervision proposals, particularly hardware-assisted supervision like memtracker, assume sequential consistency, or rather the absence of a write-buffer. A few proposals address TSO/RMO, but fail to address the correctness issues defined in [Bobba et al., 2011]. These correctness issues arise due to the violation of the memory ordering semantics of supervised instructions. In this regard, we explain the correctness issues that arise in memory supervision for weaker consistency models, and define the ordering requirement for correctness. This is further elaborated upon in chapter 4.

Decoupled memory supervision on the other hand does address weaker memory consistency models, like in [Chen et al., 2006; Kannan, 2009; Vlachos et al., 2010], but as mentioned earlier, decoupled supervision makes use of an additional processing core to perform metadata computations. This can be seen as a 100% overhead as the additional cores are used only to process metadata and not to execute the actual application program. In this thesis, we focus on hardware-assisted supervision as compared with decoupled supervision.

2.8 Summary

In this chapter, we first discussed the importance of synchronization in parallel computing, and then elaborated on the significance of RMWs operations in performing efficient synchronization. Since the cost of an RMW is dependent on the memory consistency model (or the ordering imposed by the model), we also briefly describe the various consistency models and how RMWs order themselves in each model. We elaborated on the TSO model as it is necessary to understand how the global memory order in TSO is established. We then discuss the progress guarantees provided by different synchronization mechanisms, including fairness guarantees. Finally, we discussed the operation of a supervised memory system, stressing on the requirement of metadata-data atomicity. We also describe the basic structure of a supervised memory system under sequential consistency.

Chapter 3

Fast RMWs for TSO

In this chapter, we discuss the behavior of read-modify-writes (RMWs) in processors which adhere to the Total-Store-Order (TSO) consistency model, such as the popular x86 architectures [Advanced Micro Devices, 2009; Intel Corporation, 2009] and Sun's SPARC [SPARC International, 1994]. As mentioned in earlier chapters, *Read-Modify-Write* (RMW) instructions are primitive synchronization operations used to solve a variety of concurrency problems. Herlihy [Herlihy, 1991] showed that the ability to read and write to an address atomically is critical to solve the *consensus* problem, which abstracts important synchronization problems. Most modern processor architectures have support for such RMW instructions – examples include test-and-set (TAS), fetch-and-add (FAA), compare-and-swap (CAS), and load-linked/store conditional (ll/sc).

Unfortunately, RMWs in TSO are expensive as compared to regular reads and writes. Our study sheds light on why RMWs are expensive on TSO architectures. Our observation led us to the conclusion that the memory ordering around an RMW and how atomicity is defined plays a significant role in increasing RMW cost. In the present TSO architectures, RMWs are ordered like a memory barrier, which results in the write-buffer being drained before performing an RMW. This write-buffer drain makes an RMW expensive. The reason why an RMW is ordered like a barrier is because of the confluence of the definition of atomicity it adheres to, and the consistency model. Current TSO RMWs do not allow any memory operation to occur between the read and write of the RMW [1] in the global memory order as specified by

[1] R_a denotes the read part, W_a the write part of the RMW

the TSO model. As a result, when a thread performs an RMW, the writes that are in the write-buffer of the corresponding processor must be completed before the RMW is performed. Allowing any of these writes to complete between R_a and W_a may order other read and write instructions between them.

We then study the ordering requirements of synchronization (typically *acquire* and *release*), and see that the strict ordering that current TSO RMWs (we call it Type-1 RMW) enforce is unnecessary for synchronization ordering. Also, we discuss the use of RMWs in implementing synchronization constructs for TSO as per the C/C++11 concurrency model [Becker, 2011; C, 2011]. Batty et al. [Batty et al., 2011] have shown that the C/C++11 model is correctly implementable on TSO by replacing C/C++11 SC-atomic-writes and/or SC-atomic-reads [Terekhov, 2008] [2] by Type-1 RMWs, leaving other language constructs (reads, writes, fences) to be implemented by plain TSO reads, writes and barriers.

As part of our approach, we propose to weaken the atomicity ordering of RMWs by formulating two weaker RMWs: type2, and type-3. In both cases, we allow the write-buffer to drain between the R_a and W_a. We still maintain atomicity by the existing cache-locking scheme. Type-2 and Type-3 RMWs differ in the type of memory operations whose coherence requests are served by the processor performing the RMW. We show that such weaker RMWs still provide acquire and release ordering under the TSO consistency model. We also prove that, similar to type-1 RMWs, we can replace SC-atomic-reads and SC-atomic-writes with the weaker RMWs in order to implement the C/C++11 consistency model.

Finally, we evaluate the weaker RMWs and compare their performance with the stricter type-1 RMW. Our experimental results show increased efficacy of weaker RMWs over stronger RMWs. With this evidence, we discuss how such weaker RMWs can be provided in commercial architectures and how they should be used at the program level.

[2] Please note that SC here is short for *sequential consistency*.

3.1 Introducing the Problem

As mentioned earlier, RMWs are costly in current TSO architectures, where they are ordered similarly to a memory barrier [Intel Corporation, 2009; SPARC International, 1994], incurring the cost of a write-buffer drain in the critical path. When an RMW is issued, the write-buffer is first drained; then the read and the write (of the RMW) are performed atomically – typically by locking the cache-line locally and denying coherence requests to the locked cache-line until the write completes. Thus, instructions following the RMW are allowed to complete only after the write (of the RMW) and the pending writes prior to it complete [Sorin et al., 2011]. As a quick illustration, we measured an average latency of 67 cycles for an RMW on an 8-core *Intel Sandybridge* processor, using the Splash-2 Woo et al. [1995] benchmark suite. Similarly, on a 32-core Intel Xeon Processor, the average latency of an RMW instruction is 74 cycles across the SPLASH-2 benchmark suite. This latency does not significantly change if we insert a memory barrier (`mfence` instruction) after each RMW, strengthening the hypothesis of a forced write-buffer drain. Since efficient synchronization is important to effectively harness the power of multicores, it is highly desirable that RMWs are efficient. Nevertheless, the optimization of RMWs has historically received little attention [Attiya et al., 2011].

Semantically speaking, why are TSO RMWs ordered like a memory barrier? We observe that the ordering of RMWs with other memory accesses in TSO depends on the precise semantics of how *atomic* they have to be with respect to those other accesses. TSO can be defined in terms of a global memory order, a relation over memory accesses in the program. Existing TSO RMWs are defined to prevent *writes to any address* from appearing between the read and the write in this global memory order [Intel Corporation, 2009; SPARC International, 1994]. We call this strict definition *type-1* atomicity. We show that this strict atomicity definition, combined with the other TSO ordering rules, results in type-1 RMWs being strongly ordered with respect to memory operations before and after it, just like a memory barrier.

This strong ordering is exploited by programmers in various synchronization primitives. Figure 3.1(a) shows the key steps involved in the implementation of Dekker's algorithm for achieving mutual exclusion and Figure 3.1(b) shows the same code in

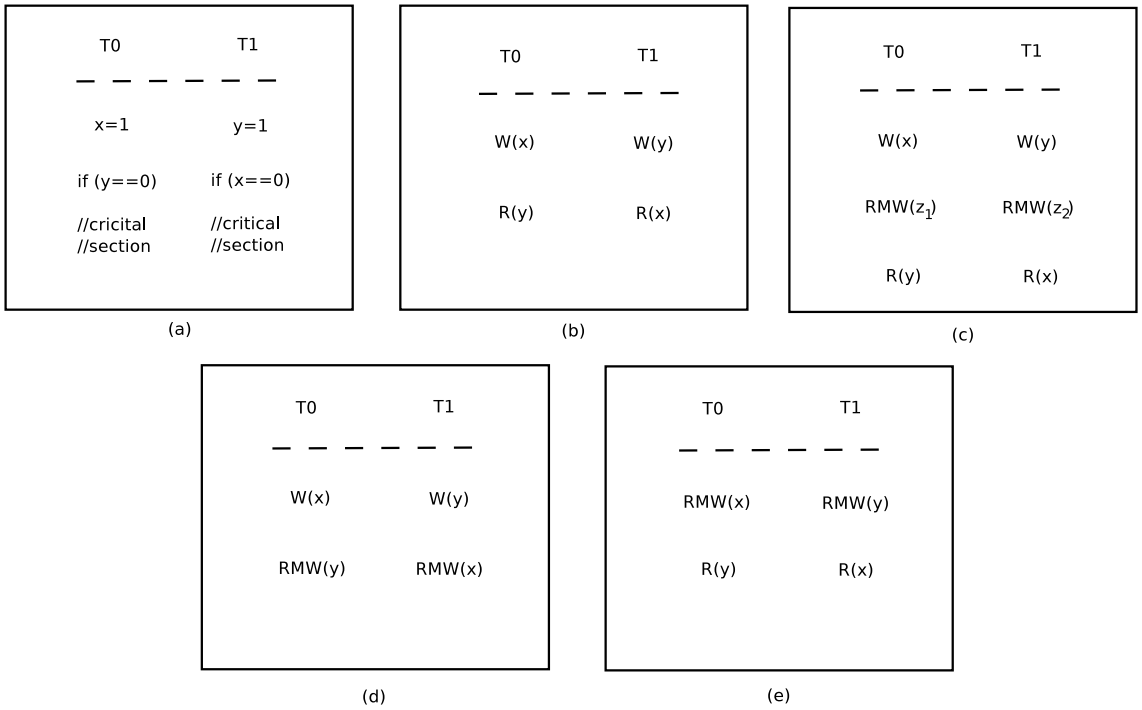


Figure 3.1: Dekker's Algorithm: (a) code snippet. (b) reads and writes involved: $W(x)$ denotes a write to address x, $R(x)$ denotes a read from address x. (c) using RMWs as memory barriers. (d) replacing reads with RMWs. (e) replacing writes with RMWs. In all subfigures, initially, x=y=0.

terms of reads and writes. For correctness, at least one of the reads should return a value of 1; otherwise both of the threads can enter the critical section simultaneously. One way to ensure this is by inserting memory barriers between the writes and the reads. In fact, since type-1 RMWs behave like memory barriers, they can be used instead of memory barriers as shown in Figure 3.1(c). Alternatively, as shown in Figure 3.1d (Figure 3.1e), correctness can also be ensured by replacing reads (and/or writes) with RMWs, since type-1 RMWs are strongly ordered with respect to memory operations before and after it in program order. For the same reason, the C/C++11 concurrency model can be implemented on TSO by replacing SC-atomic-reads (and/or SC-atomic-writes) with RMWs [Batty et al., 2011].

The goal of this work is to examine whether the ordering of TSO RMWs can be weakened in ways that enable a more efficient implementation, while remaining strong enough for it to replace existing RMWs in synchronization idioms. In other words, can

we design fast yet portable RMWs for TSO?

Our approach here is guided by the requirements of general programs, in particular by just what properties are needed for the C/C++11 implementation. Thus, this is hardware design exploiting the freedom provided by language-level concurrency models, and sufficing for those requirements.

Since the ordering semantics of an RMW depends on its atomicity semantics, our approach to weakening the ordering semantics is through weakening the atomicity semantics. In contrast to the strict *type-1* atomicity which disallows writes of *any* address between the read and the write, we consider two weaker atomicity definitions: the *type-2* atomicity which disallows only reads and writes of the same address as the RMW; and the even weaker *type-3* atomicity, which disallows only writes to the same address as the RMW.

Our key contribution is to derive the ordering semantics of the proposed weaker RMWs, and examine if the ordering is strong enough to replace existing RMWs in synchronization idioms. Unlike a type-1 RMW, a type-2 RMW is not explicitly ordered with respect to memory operations before and after it. Thus, a type-2 RMW cannot be used as a memory barrier like in Figure 3.1(c). However, we show that a type-2 RMW appears strongly ordered with respect to any memory operation that synchronizes with the RMW i.e. any memory operation from another thread that is to the same address as the RMW. Indeed, like before, Dekker's algorithm can be ported to TSO by replacing reads (and/or writes) with type-2 RMWs. It is worth noting that in the scenario shown in Figure 3.1c (Figure 3.1d), each of the RMWs appear to be strongly ordered with respect to the writes (reads) from the other thread which synchronize with the RMW; this strong ordering is again able to guarantee correctness. Consequently, type-2 RMWs are able to replace existing RMWs in synchronization idioms. Consider Figure 3.1(b) with the assumption that $RMW(y)$ is ordered before $W(y)$. Following from the definition of type-2 RMWs, $W(y)$ cannot be ordered between the read and write of $RMW(y)$ ($R_a(y)$ and $W_a(y)$). Therefore, $W_a(y)$ is also ordered before $W(y)$ i.e. $W_a(y) \rightarrow W(y)$. The underlying TSO ordering ensures the following orderings: $W(x)$ $\rightarrow W_a(y)$, and $W(y) \rightarrow W_a(x)$. This results in the ordering – $W(x) \rightarrow W_a(y) \rightarrow W(y)$ $\rightarrow W_a(x)$. Again, from type-2 atomicity definition, $W(x) \rightarrow W_a(X)$ implies that $W(x)$ $\rightarrow R_a(X)$ Thus, we still observe that $W(x) \rightarrow RMW(x)$, although type-2 RMWs are

not strongly ordered. For similar reasons, C/C++11 can be ported to TSO by replacing SC-atomic-writes (and/or SC-atomic-reads) with type-2 RMWs. Thus, type-2 RMWs are able to replace existing type-1 RMWs in all synchronization idioms, except when used as a memory barrier.

Table 3.1: Conventional RMW (type-1) vs proposed RMWs (type-2, type-3)

Atomicity Definition	Dekker's with reads replaced by RMWs?	Dekker's with writes replaced by RMWs?	Dekker's with RMWs as barriers?	C/C++11 by replacing SC-atomic-reads with RMWs?	C/C++11 by replacing SC-atomic-writes with RMWs?
type-1	✓	✓	✓	✓	✓
type-2	✓	✓	✗	✓	✓
type-3	✓	✗	✗	✓	✗

A type-3 RMW is also not explicitly ordered with respect to memory operations before and after it, and hence cannot be used a memory barrier (like a type-2 RMW). However, unlike a type-2 RMW, it appears strongly ordered only with respect to a write/RMW (but not a read) that synchronizes with the RMW. Therefore, Dekker's algorithm can be ported to TSO by replacing reads (but not writes) with type-3 RMWs. Similarly, C/C++11 can be ported to TSO by replacing SC-atomic-reads (but not SC-atomic-writes) with type-3 RMWs. Table 3.1 lists the different scenarios where type-1, type-2, and type-3 RMWs can be used interchangeably. Since, type-2 and type-3 RMWs are not strongly ordered with respect to memory operations before and after it, they do not provide barrier-like semantics provided by type-1 RMWs. Also, as mentioned above, type-3 RMWs cannot replace writes in Dekker' algorithm (or C/C++11 SC-atomic-writes).

In our final contribution, we propose efficient microarchitectural implementations of the weaker RMWs, which, in contrast to existing implementations, do not incur the cost of a write-buffer drain. Our implementation of a type-2 RMW allows instructions following it to retire as soon as the read obtains exclusive ownership of the cache-line and locks it locally. The write simply retires into the tail of the write-buffer – thus the write-buffer drain is moved out of the critical path. To guarantee atomicity, coherence requests to the locked cache-line are denied until the write (of the RMW) and the pending writes prior to it complete. However, to prevent a potential deadlock we need to ensure that the above pending writes will eventually complete, and not be blocked by an RMW from another processor. We ensure this by tracking the list of unique RMW addresses in per-processor bloom filters. When a pending write (before the RMW) is found to conflict with the list of maintained RMW addresses, we revert to draining the write-buffer, thus avoiding the possibility of a deadlock.

The type-3 RMW implementation is almost identical, with one difference. Since type-3 atomicity permits reads to the same address as the RMW between the read and the write, the read need not obtain exclusive ownership of the cache-line – leading to a potentially more efficient implementation. Our experimental results from benchmarks chosen from Splash-2, PARSEC, STAMP, and lock-free data structures show that in comparison with the existing type-1 RMW, our proposed type-2 RMW (type-3 RMW) is up to 58.9% (64.3%) cheaper, which translates into an overall performance improvement of upto 9.0%(9.2%)

We are not the first to propose weaker atomicity semantics for RMWs in general. In fact, Gharachorloo et al. [Gharachorloo et al., 1993] have already observed that it is sufficient for RMWs to use a type-3 definition for atomicity. However, in order for their TSO specification to be compliant with the original TSO specification, additional program order edges are added to RMWs, making the RMWs strongly ordered. In other words, by explicitly adding additional program order edges, the RMWs in their specification are effectively made equivalent to type-1 RMWs. In this work, we consider the case in which the atomicity definitions are weakened, but additional program order edges are not added to the RMW. Besides, our proposed type-2 atomicity definition, to the best of our knowledge has not been considered before. More on related work in.

3.2 Semantics of TSO RMWs

In this section we will propose definitions of atomicity weaker than the standard strong definition for RMWs in TSO, and derive the ordering properties that apply. We will then use those ordering properties to demonstrate the use of weakened RMWs in synchronization – in particular, we will demonstrate when they are sufficient to implement the C/C++11 concurrency model.

We begin with recalling the base TSO model (without RMWs) as described in Chapter 2, and then add our new formulations of atomicity. As mentioned earlier, our base TSO model follows Alglave [Alglave, 2010], where our atomicity definitions fit most naturally. We add RMWs to the base TSO model, and derive the memory orderings enforced by an RMW. We then show how the orderings enforced by the RMW behave in different synchronization scenarios.

3.2.1 Adding RMWs to the Base TSO model

We now consider events coming from RMWs. These correspond to one read and one write to the same location – we denote the read part of the RMW as R_a and the write part of the RMW as W_a. In an RMW, the read part comes before the write in program order – consequently, the read R_a reads an earlier value and not the value written by W_a. In addition to this, R_a and W_a need to be performed atomically, where atomicity is one of the following three definitions:

- **Type-1 Atomicity.** This is a strict definition of atomicity, used by existing TSO RMWs [Intel Corporation, 2009; SPARC International, 1994], that prevents *writes of any address* from appearing between the read and the write in the global memory order. More formally, with type-1 RMWs added to the TSO model, valid execution witnesses are ones which further impose that there is no event in ghb between R_a and W_a.

- **Type-2 Atomicity.** This is a weakening which only prevents *reads and writes of the same address as the RMW* from appearing between R_a and W_a in the global memory order. More formally: $\{\forall M(x) : M(x) \xrightarrow{ghb} R_a(x) \vee W_a(x) \xrightarrow{ghb} M(x)\}$.

- **Type-3 Atomicity.** This is a further weakening which merely prevents *writes of the same address as the RMW* from appearing between R_a and W_a in the global memory order. More formally: $\{\forall W(x) : W(x) \xrightarrow{ghb} R_a(x) \vee W_a(x) \xrightarrow{ghb} W(x)\}$.

It is important to note that even type-3 atomicity, the weakest of the atomicity definitions, satisfies the notion of atomicity required for solving the *consensus* problem [Herlihy, 1991] – consensus being the abstract problem that models synchronization idioms. Nonetheless, this does not imply that the three types of RMWs can be used interchangeably. In fact, we shall see that each of the three atomicities gives rise to RMWs that are ordered differently.

Atomicity-induced orderings. Each atomicity definition, by disallowing a specific set of memory operations between R_a and W_a in the global memory order – effectively requires both R_a and W_a of the RMW to be ordered identically with such disallowed memory operations. For example, if just R_a (and not W_a) is originally ordered before a disallowed memory operation M in the ghb ($R_a \xrightarrow{ghb} M$), then atomicity requires W_a to also be ordered before M ($W_a \xrightarrow{ghb} M$) – otherwise M could end up between R_a and W_a in the ghb. In other words, the atomicity constraint induces additional memory orderings – the atomicity relation ato is used to refer to such atomicity-induced orderings. In the above example, the ordering $W_a \xrightarrow{ato} M$ would be an atomicity-induced ordering. Accounting for such atomicity-induced orderings, the global memory order (ghb) is the linear extension of the union of com, ppo, bar, and ato. A valid execution witness, like before, is one which has an acyclic union of the above relations (including ato), and satisfies the uniproc condition. Next, we will derive the atomicity-induced memory ordering constraints for each of the atomicity definitions.

3.2.2 Type-1 RMWs

The strict type-1 definition of atomicity combined with TSO's preserved program order ensures that a type-1 RMW is strongly ordered with respect to memory operations before and after it.

Lemma 1. *An RMW placed between a write W_1 and a read R_2, results in the enforcement of $W_1 \xrightarrow{ato} R_a$, $W_a \xrightarrow{ato} R_2$ and consequently, $W_1 \xrightarrow{ato} R_2$.*

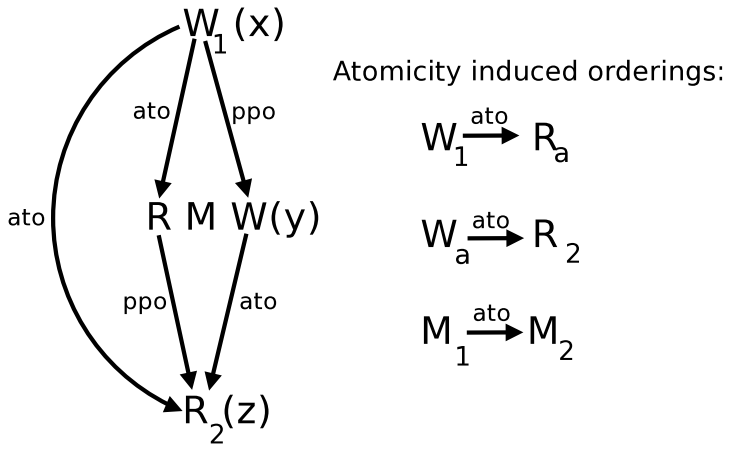


Figure 3.2: Additional memory orderings induced by type-1 RMW

Proof. Type-1 atomicity mandates that either $W_a \xrightarrow{ghb} W_1$ or $W_1 \xrightarrow{ghb} R_a$. As shown in Figure 3.2, $W_1 \xrightarrow{ppo} W_a$. This implies $W_1 \xrightarrow{ato} R_a$. Next, we prove the second part: $W_a \xrightarrow{ato} R_2$. As shown in Figure 3.2, $R_a \xrightarrow{ppo} R_2$. This implies that either R_2 occurs after W_a in the ghb or R_2 is between R_a and W_a. Meanwhile, type-1 atomicity mandates that there cannot be any writes between R_a and W_a in the ghb; in particular there cannot be any writes to location z. This implies that even if R_2 were to occur between R_a and W_a, it can be safely be moved after W_a. This in turn implies $W_a \xrightarrow{ato} R_2$. Finally, $W_1 \xrightarrow{ato} R_2$, because of transitivity ($W_1 \xrightarrow{ato} R_a$ and $R_a \xrightarrow{ppo} R_2$). □

Such strongly ordered type-1 RMWs result in costly implementations that involve a write-buffer drain; however, they can be used to port synchronization idioms to TSO without requiring additional memory barriers. Below, we demonstrate how type-1 RMWs are used in various synchronization idioms:

Dekker's: write-replacement. One way to ensure that Dekker's algorithm works on TSO architectures is to replace the writes with type-1 RMWs as shown in Figure 3.3 [Intel Corporation, 2009; SPARC International, 1994] In the above example, we assume that the read $R(y)$ from thread 0 reads the initial value of 0. For Dekker's algorithm to work the read $R'(x)$ should read a value of 1. The following sequence of orderings ensure this: $W_a(x) \xrightarrow{ato} R(y) \xrightarrow{fr} W'_a(y) \xrightarrow{ato} R'(x)$ – where ato denotes the additional orderings induced by atomicity.

Dekker's: read-replacement. Using similar reasoning, it is easy to see that replac-

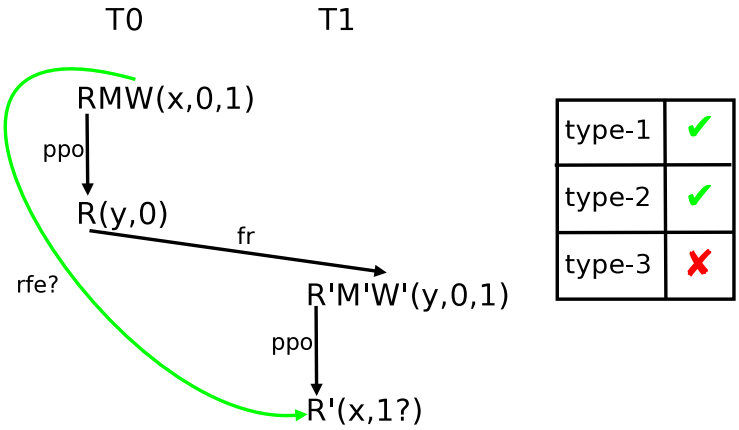
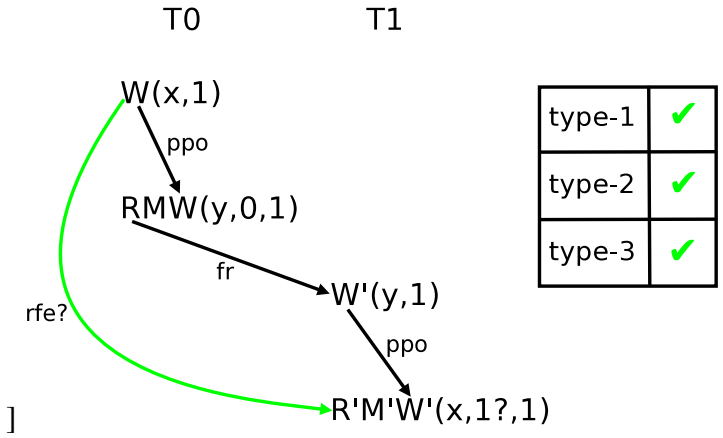


Figure 3.3: Dekker's with writes replaced by RMWs. In this and other examples that follow, $RMW(x,0,1)$ means that the RMW reads a value of 0 from location x and updates it to 1



]

Figure 3.4: Dekker's with reads replaced by RMWs.

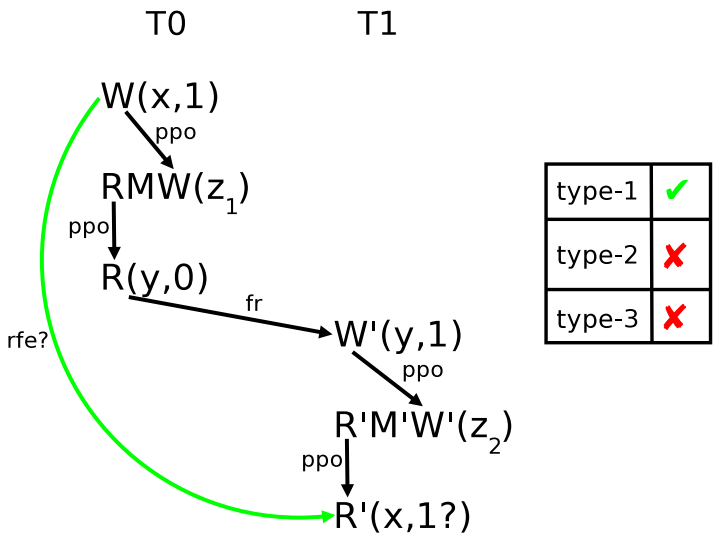


Figure 3.5: Dekker's with RMWs used as memory barriers. The two RMWs access different addresses z_1 and z_2.

ing reads with type-1 RMWs will also ensure that Dekker's algorithm works on TSO (Figure 3.4).

Dekker's: RMWs as barriers. One simple way to make Dekker's algorithm work on TSO is to insert memory barriers between the writes and the reads, as the $W \rightarrow R$ ordering enforced by the memory barriers would ensure correctness. Since type-1 RMWs order memory operations before and after it, they can very well be used instead of the barriers. As shown in Figure 3.5, the following sequence of ordering ensures correctness: $W(x) \xrightarrow{ato} R(y) \xrightarrow{fr} W'(y) \xrightarrow{ato} R'(x)$.

Implementing C/C++11 using type-1 RMWs. The C/C++11 concurrency model [Becker, 2011; C, 2011] is an adaptation of *data-race-free-0* [Adve, 1993] which guarantees SC for data race free programs. It introduces a variety of atomic memory operations parameterized by different *memory order* parameters. Correct compilation depends (among other things) on mapping these atomic memory operations to hardware primitives. Batty et al. [Batty et al., 2011] recently proved that C/C++11 can be implemented on X86-TSO by mapping C/C++11 SC-atomic-reads and SC-atomic-writes to type-1 RMWs supported by x86 architectures (non-SC atomic reads and writes and non-atomic accesses can simply be mapped to ordinary TSO reads and writes). In fact, it is easy to adapt this proof and show that it is sufficient to map at least one of the

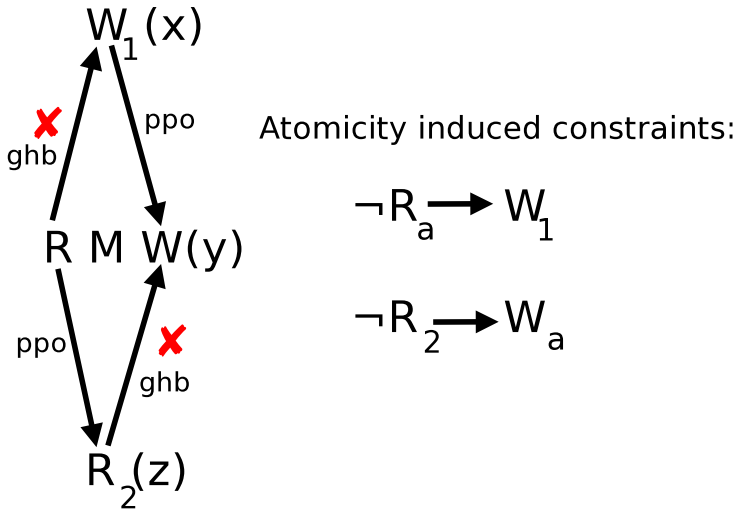


Figure 3.6: Memory ordering disallowed by a type-2 RMW

SC-atomic-writes or the SC atomic reads to type-1 RMWs. This is elaborated upon in the next section which deals with the proofs detailing the implementation of the C/C++11 model using different RMWs. Informally, since TSO already preserves all program orders except the $W \rightarrow R$ order, we only need to ensure SC-atomic-writes are ordered with subsequent SC-atomic-reads; similarly to Dekker's algorithm, this can be accomplished by replacing either the reads or writes with type-1 RMWs.

3.2.3 Type-2 RMWs

We show that, unlike a type-1 RMW, a type-2 RMW placed between a write W_1 and a read R_2 does not explicitly enforce any of $W_1 \xrightarrow{ghb} R_a$, $W_a \xrightarrow{ghb} R_2$, or $W_1 \xrightarrow{ghb} R_2$. However, as shown in Figure 3.6, it disallows $R_a \xrightarrow{ghb} W_1$ and $R_2 \xrightarrow{ghb} W_a$ from being enforced [3] – in effect, a type-2 RMW is *implicitly ordered with respect to memory operations before and after it.*

Lemma 2. *A type-2 RMW placed between two memory operations W_1 and R_2, disallows the enforcement of the following two orderings: $R_a \xrightarrow{ghb} W_1$ and $R_2 \xrightarrow{ghb} W_a$.*

[3] Disallowing an ordering $M_1 \xrightarrow{ghb} M_2$ (say) is not the same as enforcing $M_2 \xrightarrow{ghb} M_1$. The latter implies that M_2 will occur before M_1 in every valid global memory order, while the former implies that it is not necessary for M_1 to occur before M_2 in every valid global memory order

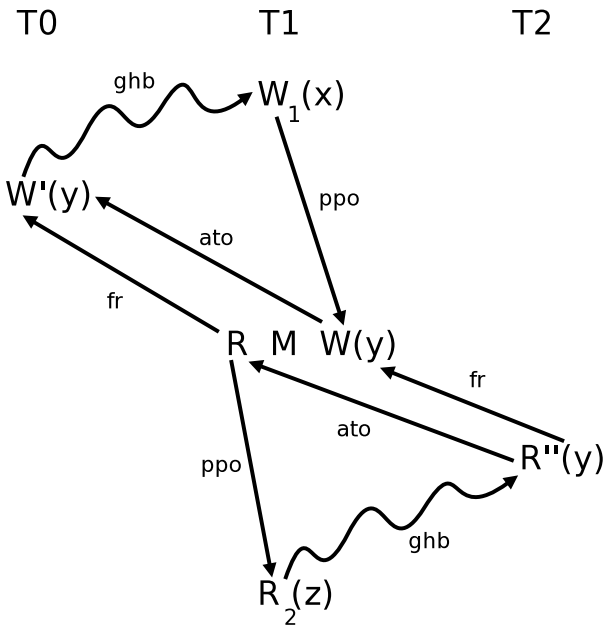


Figure 3.7: Scenario for proof of lemma 2.

Proof. Let us attempt to prove by contradiction by assuming the ordering $R_a \xrightarrow{ghb} W_1$ is enforced. Since there is no ppo edge directly connecting R_a and W_1, $R_a \xrightarrow{ghb} W_1$ we will need to be enforced via a sequence of edges as shown in Figure 3.7. More specifically, there has to be a write $W'(y)$ which conflicts with $R_a(y)$ such that: $R_a(y) \xrightarrow{fr} W'(y) \xrightarrow{ghb} W_1(x)$. But, $R_a(y) \xrightarrow{fr} W'(y)$ implies $W_a(y) \xrightarrow{ato} W'(y)$, due to type-2 atomicity. This leads to a cycle: $W_a(y) \xrightarrow{ato} W'(y) \xrightarrow{ghb} W_1(x) \xrightarrow{ppo} W_a(y)$.

Similarly for the other part let us assume $R_2 \xrightarrow{ghb} W_a$. As shown in Figure 3.7, this implies that there has to be a read $R''(y)$ which conflicts with $W_a(y)$ such that: $R_2(z) \xrightarrow{ghb} R''(y) \xrightarrow{fr} W_a(y)$. But, $R''(y) \xrightarrow{fr} W_a(y)$ implies $R''(y) \xrightarrow{ato} R_a(y)$, due to type-2 atomicity. This leads to a cycle: $R_a(y) \xrightarrow{ppo} R_2(z) \xrightarrow{ghb} R''(y) \xrightarrow{ato} R_a(y)$. □

Effect of implicitly ordered type-2 RMWs. Since a type-2 RMW neither enforces $W_1 \rightarrow R_a$ nor $W_a \rightarrow R_2$, it also does not transitively enforce $W_1 \rightarrow R_2$. Consequently, a type-2 RMW is not ordered like a memory barrier; in the next section we will propose an efficient implementation that does not incur the cost of a write-buffer drain. At the same time, a type-2 RMW appears to be strongly ordered with respect to any memory operation that synchronizes with the RMW i.e any memory operation from another thread that is to the same address as the RMW. As shown in Figure 3.7, with respect

to $W'(y)$ which synchronizes with R_a, W_1 appears to be ordered before the RMW. This is because, type-2 atomicity induces the ordering $W_a(y) \xrightarrow{ato} W'(y)$, which results in the sequence of orderings: $W_1(x) \xrightarrow{ppo} W_a(y) \xrightarrow{ato} W'(y)$, thereby ensuring $W_1(x) \rightarrow W'(y)$. Likewise, with respect to $R''(y)$ which synchronizes with W_a, $R_2(z)$ appears to perform after the RMW – the sequence of orderings $R_{''}(y) \xrightarrow{ato} R_a(y) \xrightarrow{ppo} R_2(z)$ ensures this. Consequently, type-2 RMWs can seamlessly replace existing RMWs in synchronization idioms, as we will demonstrate next.

Dekker's: write-replacement. Similarly to type-1 RMWs, Dekker's algorithm will continue to work with writes replaced by type-2 RMWs as shown in Figure 3.3. Since $R(y) \xrightarrow{fr} W'_a(y)$, $R(y) \xrightarrow{ato} R'_a(y)$ (due to type-2 atomicity). Now, the sequence of orderings $R_a(x) \xrightarrow{ppo} R(y) \xrightarrow{ato} R'_a(y) \xrightarrow{ppo} R'(x)$ ensures that $R_a(x) \xrightarrow{ghb} R'(x)$. This in turn implies that $W_a(x) \xrightarrow{ato} R'(x)$, again due to type-2 atomicity.

Dekker's: read-replacement. Using a similar reasoning, replacing reads with type-2 RMWs will also ensure that Dekker's algorithm works on TSO (Figure 3.4). Since $R_a(y) \xrightarrow{fr} W'(y)$, $W_a(y) \xrightarrow{ato} W'(y)$ (due to type-2 atomicity). Now, the sequence of orderings $W(x) \xrightarrow{ppo} W_a(y) \xrightarrow{ato} W'(y) \xrightarrow{ppo} W'_a(x)$ ensures that $W(x) \xrightarrow{ghb} W'_a(x)$. This in turn implies that $W(x) \xrightarrow{ato} R'_a(x)$, again due to type-2 atomicity.

Dekker's: RMWs as barriers (different addresses). A type-2 RMW cannot be used as a memory barrier in Dekker's algorithm if the RMWs used to replace the barriers access different addresses, since they would not appear strongly ordered with one another. As shown in Figure 3.5, it can potentially allow the following sequence of operations – $R_a(z_1), R(y), R'_a(z_2), R'(x), W(x), W_a(z_1), W'(x), W'_a(z_1)$ – which would lead to $R'(x)$ to read a value of 0.

Dekker's: RMWs as barriers (same address). A type-2 RMW, however, can be used as a memory barrier in Dekker's algorithm if the inserted RMWs access the same address, since this ensures that the RMWs appear strongly ordered to one another. As shown in Figure 3.8, type-2 RMWs used in the above fashion ensure that $R'(x)$ will read the correct value of 1. To see why, first recall that based on our assumption $R(y) \xrightarrow{fr} W'(y)$. This implies that $W_a(z) \xrightarrow{rfe} R'_a(z)$ (as the other possibility $W'_a(z) \xrightarrow{rfe} R_a(z)$ would result in the following cycle: $W'_a(z) \xrightarrow{rfe} R_a(z) \xrightarrow{ppo} R(y) \xrightarrow{fr} W'(y \xrightarrow{ppo} W'_a(z))$. This in turn leads to the sequence $W(x) \xrightarrow{ppo} W_a(z) \xrightarrow{rfe} R'_a(z) \xrightarrow{ppo} R'(x)$, ensuring that

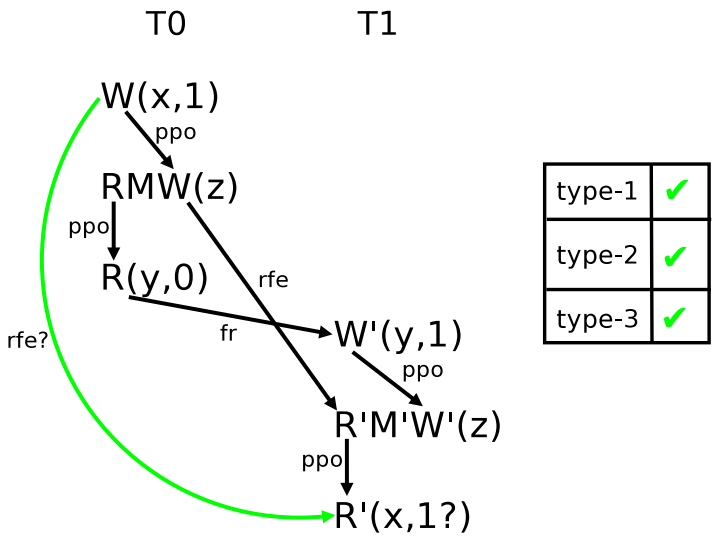


Figure 3.8: Dekker's with RMWs used as a memory barrier. The two RMWs access the same addresses z.

$R'(x)$ reads the correct value.

Implementing C/C++11 using type-2 RMWs. We formally show that, similarly to type-1 RMWs, C/C++11 can be implemented by mapping at least one of SC-atomic-writes or SC-atomic-reads to type-2 RMWs. Recall that, since TSO already preserves all program orders except the $W \rightarrow R$ order, we only need to ensure SC-atomic-writes are ordered with subsequent SC-atomic-reads. Intuitively, since type-2 RMWs appear strongly ordered when used in synchronization idioms, this can be accomplished by replacing either the SC-atomic-reads or SC-atomic-writes with RMWs, as shown in the next section.

3.2.4 Type-3 RMWs

We show that, similarly to a type-2 RMW, a type-3 RMW placed between W_1 and R_2 does not explicitly enforce any of $W_1 \xrightarrow{ghb} R_a$, $W_a \xrightarrow{R}_2 ghb$, or $W_1 \xrightarrow{ghb} R_2$. However, unlike a type-2 RMW it disallows only $R_a \rightarrow W_1$ (but could allow $R_2 \rightarrow W_a$) – in effect, a type-3 RMW is implicitly ordered with respect to memory operations before it, but not with those after it.

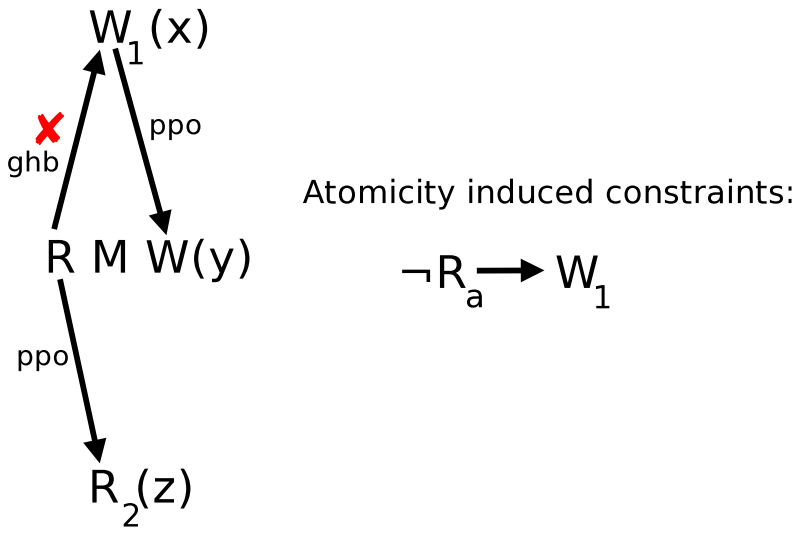


Figure 3.9: Memory ordering disallowed by a type-3 RMW

Lemma 3. *A type-3 RMW placed between two memory operations W_1 and R_2, disallows $R_a \xrightarrow{ghb} W_1$ (but could allow $R_2 \xrightarrow{ghb} W_a$ to be enforced).*

Proof. Proof of $\neg R_a \xrightarrow{ghb} W_1$ is identical to the first part of the proof of lemma 2. To understand why $R_2 \xrightarrow{ghb} W_a$ is not disallowed, let us consider the second part of the proof of lemma 2, where we assumed $R_2 \xrightarrow{ghb} W_a$. As shown in Figure 3.7, this implies that there has to be a read $R''(y)$ which conflicts with $W_a(y)$ such that: $R_2(z) \xrightarrow{ghb} R''(y) \xrightarrow{fr} W_a(y)$. Recall that type-2 atomicity induced the ordering: $R''(y) \xrightarrow{ato} R_a(y)$, which led to a cycle. However, such an ordering is not induced by type-3 atomicity, which allows for reads to happen between the $R_a(y)$ and $W_a(y)$, and so there is no cycle. □

Effect of implicitly ordered type-3 RMWs. Since a type-3 RMW enforces neither $W_1 \rightarrow R_a$ nor $W_a \rightarrow R_2$, it also does not transitively enforce $W_1 \rightarrow R_2$. Consequently, a type-3 RMW is not ordered like a memory barrier. At the same time, a type-3 RMW appears to be strongly ordered with respect to any write/RMW that synchronizes with the RMW. As shown in Figure 3.7, with respect to $W'(y)$ which synchronizes with R_a, W_1 appears to be ordered before the RMW. This is because type-3 atomicity induces $W_a(y) \xrightarrow{ato} W'(y)$, which in turn results in the sequence of orderings: $W_1(x) \xrightarrow{ppo} W_a(y) \xrightarrow{ato} W'(y)$ which ensures this. On the other hand, with respect to the read $R''(y)$ which

synchronizes with W_a, $R_2(z)$ does not appear to be ordered after the RMW, since type-3 atomicity allows $R''(y)$ to occur between $R_a(y)$ and $W_a(y)$. Consequently, type-3 RMWs cannot seamlessly replace existing RMWs in synchronization idioms, as we will demonstrate next.

Dekker's: write-replacement. Unlike type-1 or type-2 RMWs, replacing writes with type-3 RMWs cannot guarantee correctness (Figure 3.3). This is because type-3 atomicity is not able to induce $R(y) \xrightarrow{ato} R'_a(y)$. Hence, the following sequence is allowed: $R_a(x), R(y), R'_a(y), R'(x), W_a(x), W'_a(y)$ – which would lead to $R'(x)$ to read 0.

Other Dekker's scenarios. For the other Dekker's algorithm scenarios (Figure 3.4, Figure 3.5, and Figure 3.8) a type-3 RMW behaves identically to a type-2 RMW.

Implementing C/C++11 using type-3 RMWs. We formally show that C/C++11 can be implemented by mapping SC-atomic-reads (and optionally SC-atomic-writes) to type-3 RMWs. However, it is not sufficient (unlike type-1 and type-2 RMWs) for only the SC-atomic-writes to be so mapped. Intuitively, since type-3 RMWs appear strongly ordered only when synchronizing with writes or RMWs, but not reads, all SC-atomic-reads need to replaced with RMWs. How this can be proven to implement the C/C++11 concurrency model is discussed in the next section.

3.2.5 Summary

We show that type-2 RMWs can seamlessly replace type-1 RMWs in various synchronization idioms, except when a type-1 RMW is used purely as a memory barrier. Given that all modern TSO(-like) architectures have a dedicated memory barrier instruction, there is no need to use an RMW as a barrier. Furthermore, type-2 RMWs can still be used as a memory barrier provided such RMWs are forced to synchronize with each other (by forcing them to access the same address). Similarly to type-2 RMWs, type-3 RMWs also do not behave like memory barriers. However, unlike type-2 RMWs, type-3 RMWs only appear ordered with respect to writes/RMWs (but not reads) that synchronize with the RMW; thus type-3 RMWs cannot seamlessly replace type-1 RMWs. Nevertheless, we show that by replacing synchronization reads with type-3 RMWs, the above synchronization idioms can still be implemented using type-3 RMWs.

Table 3.2: Mapping from C/C++11 to X86

(a) read-write-mapping

Operation	x86 Impl.
non-SC read	mov
SC read	lock xadd(0)
non-SC write	mov
SC write	lock xchg

(b) read-mapping

Operation	x86 Impl.
non-SC read	mov
SC read	lock xadd(0)
non-SC write	mov
SC write	mov

(c) write-mapping

Operation	x86 Impl.
non-SC read	mov
SC read	mov
non-SC write	mov
SC write	lock xchg

3.3 C/C++11 implementation proofs

In this section, we show how type-2 and type-3 RMWs can be used to implement the C/C++11 concurrency model. [4] Recall that the C/C++11 concurrency model [Becker, 2011; C, 2011] has marked memory accesses of various kinds (only SC is important on TSO, the properties of the others are automatically satisfied by normal reads and writes on TSO). We work with the formal description in Batty et. al [Batty et al., 2011]. For a particular execution of a program, various relations among the actions corresponding to these operations are defined, including a happens-before relation; modification order mo, a total order per atomic location on writes to that location; and SC order sc, a total order on all SC atomic actions in the execution. There are several consistency conditions which these relations must satisfy for the execution to be consistent (briefly, both mo and sc must be consistent with happens-before; the ithb part of happens-before must be acyclic (C/C++11 ithb refers to the inter-thread-happens-

[4] Please note that the C/C++11 implementation proofs were formulated by Dr. Susmit Sarkar, University of St. Andrews.

before ordering, which involves the ordering from a release write to an acquire read of another thread); certain shapes contradicting coherence must not occur within happens-before; and reads must read from a happens-before consistent write). Furthermore, if any consistent execution in the sense above has a data race, then the program as a whole has no defined semantics.

Correct compilation to TSO depends (among other things) on mapping the atomic accesses to TSO hardware primitives. Batty et al [Batty et al., 2011] prove correctness for a few variant mappings on X86-TSO; specifically, the *read-write-mapping* of Table 3.2(a) (from a prototype by Terekhov [Terekhov, 2008]), which maps SC-atomic-reads and SC-atomic-writes to X86-TSO RMWs. It is easy to adapt their proof and weaken the mapping, making only the SC-atomic-reads RMW's as in Table 3.2(b): *read-mapping*, or only the SC-atomic-writes RMW's as in Table 3.2(c): *write-mapping*. We now show that each mapping above would suffice for correctly implementing C/C++11 using type-2 RMWs (and reprove for type-1), while for type-3 RMWs, the read-write mapping and the read-mapping work. The write-mapping would not work for type-3 RMWs, by Dekker's counterexample in the work (Figure 3.3).

3.3.1 A generic outline of the proof strategy

The proof is fairly standard, following the proofs in [Batty et al., 2012, 2011]. In particular, the way of constructing SC orders is derived from the earlier work.

Mapping read-from maps, and mo. First, the events occurring in the hardware models are related to the C/C++ actions from the corresponding program. For everything except the C/C++11 SC atomics, this is straightforward, as ordinary reads and writes correspond to C/C++11 reads and writes. For the SC actions, we assume that there is a unique mapping that can be derived. Then the hardware rf relation corresponds to the reads-from map of C/C++11, and the hardware ws relation (restricted to *atomic* locations) corresponds to mo of C/C++11.

ghb **contains the C/C++11 inter-thread-happens-before ordering** *ithb*. We notice that under any mapping (and any kind of RMW), each of the components of C/C++11 ithb are part of ghb, by the construction via release sequences. Thus the ghb is a greater

relation than the C/C++11 ithb. The key notion in C/C++11 ithb is the release sequence, which crosses threads from a write (release write) to a read (acquire read) that reads from it. There are possibly intervening writes to the same location on the release thread, and program order subsequent accesses on the acquire thread. Since external reads-from (rfe), coherence (ws) and reads to other accesses on the same thread (part of ppo) are all parts of ghb, ithb is contained within ghb.

Constructing the C/C++11 SC order. This part of the proof crucially depends on the mapping, so we will have to parameterize the proof by the mapping. We consider, as in the proof of SC actions on Power [Batty et al., 2012], an arbitrary linearization of the union of $\mathsf{po}_{\mathsf{SC}}$, program-order on SC actions; $\mathsf{ws}_{\mathsf{SC}}$, ws restricted to SC actions; $\mathsf{fr}_{\mathsf{SC}}$, which relates SC reads to all SC writes to the same location coherence-after the write the read reads from; and $\mathsf{erf}_{\mathsf{SC}}$, which relates a SC read and the last SC write in coherence before the write, or that write if a SC write, that the read reads-from.

We will then show that these relations are included in the ghb relation, and thus their union is consistent with ghb. As a corollary, by the acyclicity of ghb, we get that the union is acyclic and thus can be extended to a linear SC order.

C/C++11 concurrency. Assuming we can construct the SC order as above, we are now in a position to verify the consistency in C/C++11 of all behaviors permitted by TSO (with the variant RMWs) for race-free C/C++11 programs:

- Acyclicity of ithb: First, the ithb is contained within ghb, which is acyclic.
- Consistency of happens-before and mo: Second, mo should be consistent with C/C++ happens-before (which we get by ws being included in ghb, and the uniproc condition).
- Coherence diagrams: Third, the coherence diagrams [Batty et al., 2011] CoRR, CoRW, CoWR, and CoWW, must not be contradicted by the happens-before, which we get by the construction of ghb.
- Consistency of SC order: Fourth, sc should be consistent with happens-before and mo, which we get by our construction of sc.
- Reads read from a consistent write: Fifth, SC reads must read-from a write not happens-after the sc-last SC write, which we get by construction of sc. Other reads

must read from a happens-before consistent write, where we note that all reads read from the last write to the same location in ghb. It is possible, however, that there is no C/C++11 happens-before relating the read and write (hb is smaller than ghb). Then, we find a race in the original C/C++ program, contradicting the race-free assumption.

- Constructing a race: Suppose we have found a read and a write that it reads-from that are not C/C++11 happens-before related. We find the minimal such pair in ghb (we know ghb is acyclic, so this is well-founded). Cut off the program without this read, and anything program-order after that write. Now we add back the read, but read from a C/C++11 allowed write; and it races with the original write. We complete the program execution in any consistent way, to get a racy consistent execution. Note that without speculative execution as in Power, this proof is much simpler than the corresponding proof for Power [Batty et al., 2012].

3.3.2 Instantiating the generic proof

Now we fill in the pieces above for each atomicity definition and each mapping. The remaining obligation is finding events in the TSO execution corresponding to the C/C++11 SC atomics, and proving that $\mathsf{po}_{\mathsf{SC}}$, $\mathsf{ws}_{\mathsf{SC}}$, $\mathsf{fr}_{\mathsf{SC}}$, and $\mathsf{erf}_{\mathsf{SC}}$ are contained within ghb.

Read-write-mapping and read-mapping. For these mappings, we consider the write W_a of the RMW for the SC read, and the write (either by itself in the read-mapping, or from the RMW for the read-write-mapping) for the SC write. Then $\mathsf{po}_{\mathsf{SC}}$is a part of ghb (they are same-thread writes). $\mathsf{ws}_{\mathsf{SC}}$is a part of ghb by definition of write-serialization. Every $\mathsf{fr}_{\mathsf{SC}}$edge must be consistent with ghb, since the subsequent write cannot be in ghb between R_a and W_a of the RMW, using any atomicity definition. Every $\mathsf{erf}_{\mathsf{SC}}$edge must be consistent with ghb, since the write read-from must be coherence-before the W_a of the SC read, and cannot come between R_a and W_a in any atomicity definition.

Write-mapping Here SC reads are mapped to plain reads, and thus there is no write to use as above. Instead, we use the read as is for SC reads, and the read R_a of the RMW for the SC write. Using this mapping, $\mathsf{po}_{\mathsf{SC}}$is a part of ghb (they are same-thread reads). For write-serialization, $\mathsf{ws}_{\mathsf{SC}}$is a part of the ghb, since R_a of each write must be before that write in fr. Likewise, $\mathsf{erf}_{\mathsf{SC}}$is a part of ghb, but the proof has two

cases. For same threads, R_a of the write is ghb-before the read (same-thread reads). For different threads, R_a from the write is ghb before W_a in fr, and W_a is before the SC read in rfe. The last piece required is $\mathsf{fr}_{\mathsf{SC}}$. The SC read is certainly before in fr W_a of the RMW, but we are now considering R_a as representing the SC action. For Type-1 and Type-2 RMWs, it is consistent to impose that the SC read is before R_a, since they are to the same location, and no same-location actions can be in ghb between R_a and W_a. Then we get the required result.

For Type-3 RMWs, since a *read* can be in between R_a and W_a of a RMW, this strategy will not work. This is the point where the proof fails for Type-3 RMWs.

3.4 TSO RMWs: Implementation

In this section we first discuss how existing type-1 RMWs are implemented. We then describe our proposed type-2 and type-3 RMW implementations. For the following discussion we assume a chip multiprocessor with local L1 caches and a shared L2 cache; the local caches are kept coherent at the L2 cache level using a distributed directory based coherence protocol.

3.4.1 Type-1 RMW

Recall that a type-1 RMW is strongly ordered with respect to memory operations before and after it: a type-1 RMW placed between write W_1 and read R_2 results in the enforcement of $W_1 \rightarrow R_a$ and $W_a \rightarrow R_2$, where R_a/W_a are the read/write of the RMW respectively. To enforce $W_1 \rightarrow R_a$, pending writes in the write-buffer (if any) must complete before R_a can retire.

Furthermore, type-1 atomicity mandates that there should not be any conflicting reads or writes (to the same address as the RMW) between R_a and W_a. To ensure this, existing RMW implementations use a cache-line locking mechanism [Intel Corporation, 2009; Michael and Scott, 1995b; SPARC International, 1994]. The R_a obtains read/write permissions for the cache-line, and locks it before it retires, thereby denying coherence requests to the cache-line. Once W_a completes, the cache-line is unlocked.

To ensure that $W_a \rightarrow R_2$ is enforced, R_2 is allowed to retire only after W_a com-

pletes. In other words, reads that follow the RMW have to wait until: (a) all writes prior to the RMW are performed (the write-buffer is drained) and (b) R_a and W_a are performed. Thus, the type-1 RMW incurs the cost of a write-buffer drain and the cost of performing R_a and W_a.

Gharachorloo et al. [Gharachorloo et al., 1991] proposed two techniques to provide efficient memory ordering. Both these techniques can be used to improve the performance of type-1 RMWs. The first one involves issuing the read-exclusive request for all pending writes in parallel, to efficiently enforce the write-buffer drain. The actual writes, however, are completed in-order, keeping with TSO. Parallel issue of the read-exclusives will be serialized at the local L1 cache and at the directory, but will make full use of the interconnect and overlap invalidation and acknowledgement messages for all the pending writes. The second technique is to hide part of the write-buffer drain latency through in-window speculation. Here, the instructions following the RMW are speculatively executed, but are allowed to complete only after the RMW and all the pending writes before it complete.

3.4.2 Type-2 RMW

Recall that a type-2 RMW is not explicitly ordered with respect to memory operations before and after it in the program order. Since a type-2 RMW that is placed between memory operations W_1 and R_2, does not enforce $W_1 \rightarrow R_a$, R_a need not wait for the write-buffer to be drained. However, type-2 atomicity still disallows conflicting reads or writes from appearing between R_a and W_a in the global memory order. Similarly to a type-1 RMW, this is ensured using the cache-line locking mechanism. Like before, R_a obtains read/write permissions for the cache-line, locks the cache-line, and then retires. After this, W_a simply retires into the tail of the write-buffer. At this point the RMW effectively retires, and allows memory operations following it (e.g. R_2) to retire (since $W_a \rightarrow R_2$ is not enforced). Finally, when W_a reaches the head of the write-buffer and completes, the cache-line is unlocked.

Write-deadlocks. The above implementation, while simple, can potentially result in a deadlock. To guarantee type-2 atomicity, coherence requests to the cache-line locked by an RMW are denied until W_a and the pending writes prior to it complete. If such a

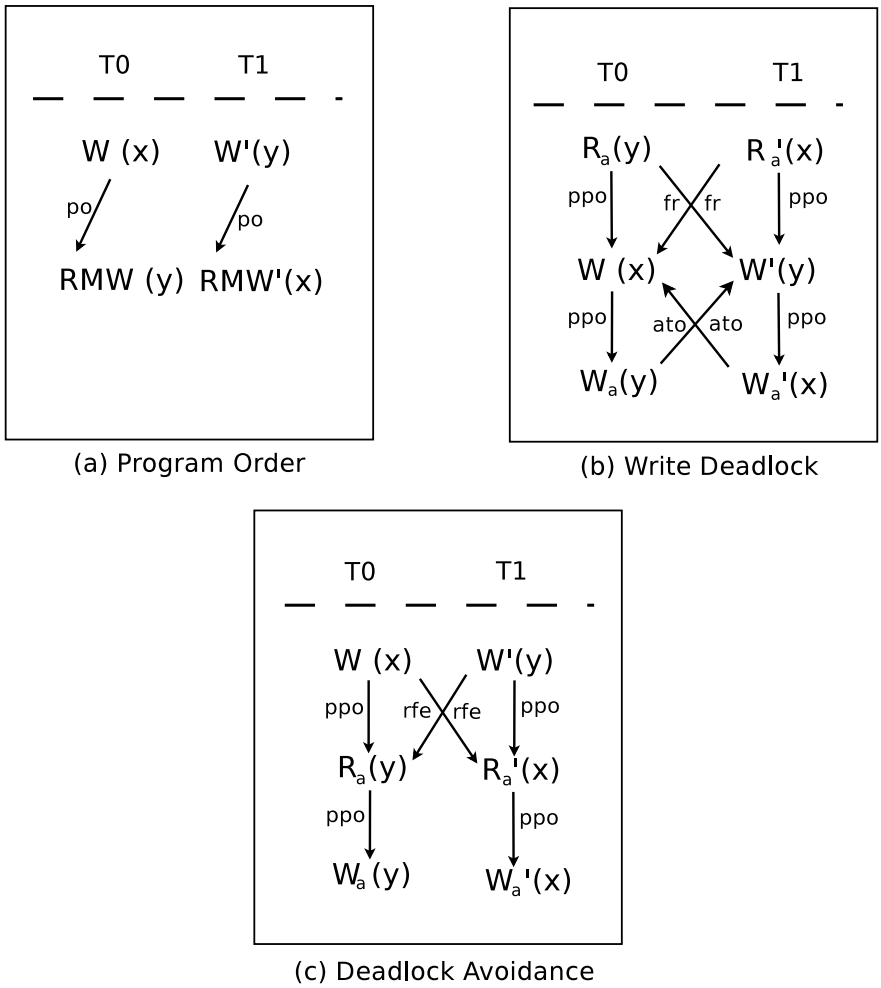


Figure 3.10: (a) shows a code segment that can cause a write-deadlock. (b) shows an execution order with a cyclic dependency of non-occurring events resulting in a write-deadlock. (c) shows how the cyclic dependency is removed by forcing a write-buffer drain for both RMWs. The fr orderings are converted to rfe orderings

pending write W_1 is to a cache-line which has already been locked by another RMW$'$ from a different processor, then W_1 (and hence W_a) will have to wait until W'_a completes. If W'_a itself is stalled because of a similar write in its write-buffer, a deadlock manifests.

This is illustrated in the code segment shown in Figure 3.10(a), where $W(x)$ occurs before RMW(y), and $W'(y)$ occurs before RMW$'(x)$ in program order. As shown in Figure 3.10(b), let us assume that $R_a(y)$ and $R'_a(x)$ have retired after locking their respective cache-lines, while the writes ($W(x)$ and $W'(y)$) have retired into the write-buffer and are yet to complete. Cache-line locking ensures that $W(x)$ cannot complete until $W'_a(x)$ has completed, and $W'(y)$ cannot complete until $W_a(y)$ has completed. However, since writes are ordered in TSO, $W_a(y)$ cannot complete until $W(x)$ completes, and $W'_a(x)$ cannot complete until $W'(y)$ completes. This leads to a write-deadlock.

More formally, our assumptions can be represented by the two fr orderings: $R'_a(x) \xrightarrow{fr} W(x)$ and $R_a(y) \xrightarrow{fr} W'(y)$. Now, type-2 atomicity induces the two orderings: $W'_a(x) \xrightarrow{ato} W(x)$ and $W_a(y) \xrightarrow{ato} W'(y)$. This in turn results in a cycle: $W(x) \xrightarrow{ppo} W_a(y) \xrightarrow{ato} W'(y) \xrightarrow{ppo} W'_a(x) \xrightarrow{ato} W(x)$. Since each of the memory operations which are part of the cycle have not yet performed, a deadlock ensues.

Deadlock avoidance. In order to ensure that the deadlock scenario discussed above never occurs, we should guarantee that none of the pending writes before an RMW, are to cache-lines locked by other RMWs – the deadlock safety property. To ensure this, we propose a mechanism to dynamically maintain the set of unique RMW addresses accessed by RMWs from all processors – the *addr-list*. Furthermore, we make this addr-list available locally to each of the processors.

Now, when an RMW is performed, if none of the pending writes in the write-buffer conflict with the addr-list, we can safely say that these writes are not to locked cache-lines. On the other hand, if any of the pending writes conflicts with the addr-list, the deadlock safety property is not guaranteed. In such a case, we revert to type-1 implementation by draining the write-buffer before performing the RMW – thereby avoiding a deadlock.

There are two challenges to efficiently implementing this mechanism in hardware: (a) keeping track of the RMW addresses in the addr-list efficiently; (b) keeping the addr-list coherent across all processors. We implement the addr-list using a bloom

filter [Bloom, 1970], which is a well understood mechanism for maintaining sets and supporting membership queries. In order to keep the addr-list coherent we simply broadcast the address whenever a new RMW address is encountered by a processor. Our design exploits the fact that the number of unique RMW addresses is relatively small – our experiments show that typically around 1% of the number of dynamic RMWs are to unique addresses. This in turn means that the addresses of the RMWs can be stored efficiently in a relatively small-sized bloom filter, with a low probability of false positives. More importantly, the number of broadcasts required to keep the addr-list coherent is minimal.

We now explain the working of our mechanism in more detail. When an RMW is ready to perform, we first query the bloom filter for the RMW address. If the address is not found in the filter, we insert the RMW address into the local bloom filter. In addition to this, since the addr-list has changed, we broadcast the new address to all processors. Each of the other processors, upon receiving the address, inserts the address into its respective bloom filter and sends back an acknowledgement. Once all acknowledgements have been received (or if the RMW's address is found in the addr-list in the first place), we query the bloom filter with the pending writes' addresses. If any of these write addresses are found in the addr-list, this flags a potential deadlock. Consequently, the write-buffer is drained before performing the RMW like a type-1 RMW. On the other hand, if none of the pending writes' addresses are found, the RMW does not wait for the write-buffer to drain. It locks the cache-line and simply retires, while the write of the RMW is retired into the write-buffer.

To see why our scheme is correct note that an RMW can lock the cache-line and retire (with pending writes in the write-buffer) only when:

- c1: the RMW's address is made visible to all processors
- c2: none of the pending writes conflict with the addr-list.

Now, c1 implies that any write (W') that could be potentially involved in a deadlock with the original RMW will conflict with the local addr-list. c2 implies that an RMW with W' in its write-buffer will revert to type-1, thereby avoiding a deadlock. Consider the deadlock scenario shown in Figure 3.10(c). Recall that in the deadlock scenario, $R_a(y)$ and $R'_a(x)$ have retired, but their respective pending writes $W(x)$ and

$W'(y)$ are unable to complete (inducing the two fr orderings: $R'_a(x) \xrightarrow{fr} W(x)$ and $R_a(y) \xrightarrow{fr} W'(y)$). The fact that $R_a(y)$ and $R'_a(x)$ have retired implies that both x and y must be present in the bloom filter (from c1). In addition to this, $W(x)$ and $W'(y)$ should have checked the filter for conflicts (from c2). The assumed fr orderings imply that neither of the writes conflicted with the bloom filter. This in turn implies that neither x nor y are in the bloom filter leading to a contradiction.

False Positives.

Bloom filters suffer from false positives. The correctness of our scheme, however, is not compromised due to false positives. A false positive may result from either an RMW or a pending write checking the bloom filter. When an RMW, whose address has not been encountered before, queries the bloom filter and the bloom filter returns a false positive, the RMW address ends up not being broadcast. Similarly, when a pending write queries the bloom filter and the filter returns a false positive, the write-buffer is unnecessarily drained. The former case is safe, as any write which conflicts with an RMW address (which causes a false positive) will also similarly return a false positive. It is worth noting that false positives in this case may reduce the number of RMW broadcasts. In the latter case, the unnecessary write-buffer drains will affect performance. The correctness of the mechanism, however, is not affected.

Finally, in our design, the bloom filter keeps track of RMW addresses of all contexts. In other words, each bloom filter is independent of the thread context. While this may increase the probability of false positives, it again does not present any correctness issues.

It is worth noting that, the probability of false positives in the filter increases with the number of elements inserted into it, leading to a performance degradation over time. To handle this, we reset the bloom filters of all processors when the number of RMW addresses inserted into the filter exceeds a certain threshold, which is a function of the bloom filter configuration. To ensure correctness, when a processor receives a reset request, it waits until all in-flight RMWs have completed, and responds subsequently.

3.4.3 Type-3 RMW

Recall that a type-3 RMW, like a type-2 RMW, is not explicitly ordered with respect to memory operations before and after it. R_a need not wait for the write-buffer to be drained – it can retire even if there are pending entries in the write-buffer. However, type-3 atomicity still disallows conflicting writes and other RMWs from appearing between R_a and W_a in the global memory order. Since reads to the same memory address can appear between R_a and W_a, it is sufficient for R_a to get read permissions for the cache-line, unlike type-1/type-2 RMW which require read/write permission.

If the RMW is to a cache-line owned by the local cache, then it is locked in the cache itself before retiring R_a, similar to type-1/type-2 RMWs. If the RMW is to cache-line in *shared* state, however, locking the cache-line locally cannot prevent an RMW from another processor, which also has the cache-line in its local cache, from performing. To resolve this, we propose a *directory locking* protocol, wherein R_a to a cache-line in shared state is locked in the directory by transitioning the cache-line to a *locked* state. When W_a is issued from the write-buffer, the cache-line is transitioned out of the locked state allowing subsequent coherence requests to the cache-line to be serviced. This optimization removes any invalidation delay, incurred by the RMW, from the critical execution path.

Once R_a obtains a lock and retires, W_a simply retires into the tail of the write-buffer. At this point the RMW effectively retires, and allows memory operations following it to retire. Thus reads that follow a type-3 RMW will only have to wait until R_a obtains read permission for the cache-line and locks it. Finally, when W_a reaches the head of the write-buffer and completes, the lock on the cache-line is released. Similarly to type-2 RMWs, the implementation of type-3 RMWs also makes use of the bloom filter mechanism to avoid deadlocks.

3.5 Experimental Evaluation

The primary goal of our experiments was to compare the cost of type-1, type-2, and type-3 RMWs. Furthermore, we evaluated the impact of the different types of RMWs on the overall execution time of the benchmark programs. Since RMWs are also

used to implement C/C++11 SC-atomic-reads and/or SC-atomic-writes, we also investigated the performance of supporting C/C++11 concurrency model with type-1, type-2 and type-3 RMWs. We briefly describe our implementation before discussing the results.

3.5.1 Implementation

Table 3.3: Architectural Parameters

Processor	32 core CMP, inorder
Write-buffer	32-entry deep
L1 Cache	private, 32 KB 4-way 2-cycle latency
L2 Cache	shared, 1 MB per-core, 16-way 6-cycle latency
Memory	300 cycle latency
Coherence	MOESI distributed directory
Interconnect	2D Mesh, 1-cycle link, 4-cycle router latency

Simulator. We use the GEM5 simulator to implement our baseline system, which is an x86-based CMP composed of inorder processors, with local L1 caches and a shared-distributed L2 cache. Cache latencies were obtained from CACTI [Muralimanohar and Balasubramonian, 2009]. The baseline uses type-1 RMWs. The local caches are kept coherent using a distributed directory based on the MOESI coherence protocol. We chose inorder cores for our simulation as the GEM5's out-of-order processor model is unstable for full system simulation of the x86 processor architecture. The choice of inorder cores, however, is a valid design point owing to the fact that several present and future many-core processors, like the Intel Xeon Phi, Sun Niagara T2, and NVIDIA GPUs, make use of inorder cores as opposed to out-of-order cores to achieve better performance to power ratios. As mentioned in the previous section, we implemented a parallel write-buffer drain mechanism. This improves the baseline significantly over the serial write-buffer drain. We did not implement in-window speculation as it is not applicable to inorder processors. The architectural parameters for

Table 3.4: Benchmark Characteristics

Code	Suite	Problem Size	RMWs per 1000 mem-ops	% of RMWs ops with unique ad-dresses	% of RMWs ops which force write-buffer drains	RMW broad-casts per 100 RMW ops
radiosity	SPLASH-2	room	15.56	0.28	0.06	0.26
raytrace	SPLASH-2	car	13.83	0.02	0.12	0.02
fluidanimate	PARSEC	medium	17.43	0.46	0.09	0.46
dedup	PARSEC	medium	8.10	3.31	0.20	3.12
bayes	STAMP	bayes+	34.15	0.91	0.01	0.80
genome	STAMP	genome+	6.19	0.64	0.10	0.52
wsq-mst	Lock-free	10000 nodes	23.41	3.80	0.07	3.71

our implementation are presented in Table 3.3.

We modified the simulator to implement type-2 and type-3 RMWs with deadlock avoidance. In our implementation, we used a 128B bloom filter with 3 hash functions. It is worth noting that the only hardware overhead for type-2/type-3 RMWs is the 128B bloom filter and a RMW threshold counter per processor. Also, we did not make use of the threshold counter in our simulations as we ran only a single context which did not require a bloom filter reset for good performance.

Benchmarks. We evaluate our technique using benchmarks in Table 3.4, which includes both lock-based and a lock-free program. *radiosity* and *raytrace* are benchmarks from the Splash-2 suite which primarily use RMWs in lock/unlock primitives. Similarly, *fluidanimate* and *dedup* (from PARSEC) are also lock-based benchmarks. It is worth noting here that we chose only the top two benchmarks from each suite, in terms of the ratio of RMW instructions to other memory operations. We do this as the Splash-2 and PARSEC benchmarks are optimized to reduce synchronization and the resulting synchronization cost. Thus, even if we reduce the cost of RMWs, it may not have an impact on the overall execution time of the benchmark. On the other hand, lock-free programs use more RMWs taking advantage of low-latency communication on multicores. *wsq-mst* is a lock-free parallel spanning tree algorithm [Bader and Cong, 2005] using Chase-Lev work stealing queue. *bayes* and *genome*, from the STAMP (using TL2 [Dice et al., 2006]), use RMWs for locking writes in transactions and to commit transactions. We ran the benchmarks in their regions of interest, with the input sizes mentioned in Table 3.4.

C/C++11 concurrency. Because of the recency of the C/C++11 concurrency model, there is no corpus of C/C++11 code to test our ideas on. We therefore modified the *wsq-mst* program to make use of atomic reads/writes as prescribed by the C/C++11 model. *wsq-mst* uses Dekker-like synchronization to update the task queue pointers while removing tasks from the queue; thus the read and write of this synchronization primitive corresponds to an SC-atomic-read and SC-atomic-write respectively. As mentioned earlier, the C/C++11 concurrency model can be realized by replacing SC-atomic-writes and/or SC-atomic-reads with RMWs. We compare the performance of the different types of RMWs by replacing either the SC-atomic-reads (*wsq-mst_rr*) or SC-atomic-writes (wsq-mst_wr) with RMWs. We do not consider type-3 RMWs for

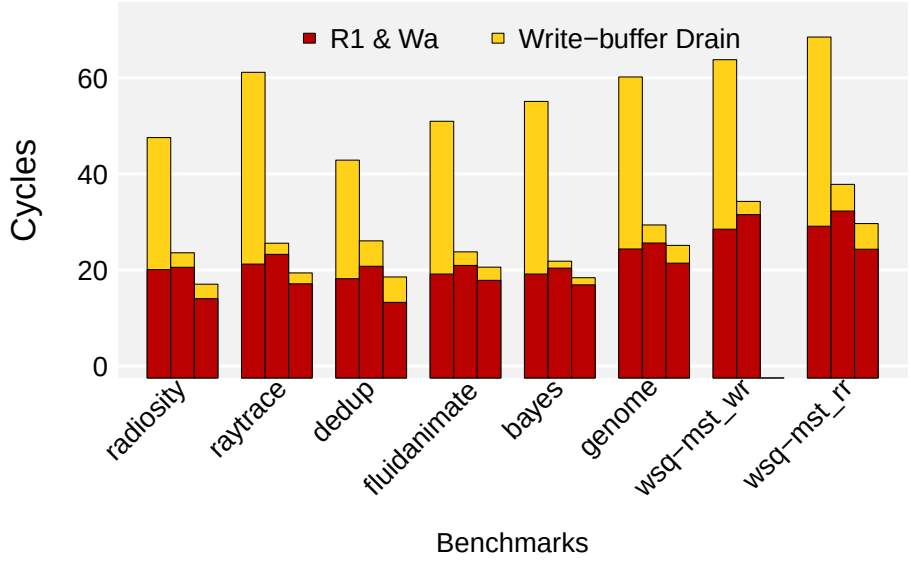


Figure 3.11: Cost of type-1, type-2, and type-3 RMWs

write replacement here as that cannot guarantee correctness (as described in §2.5).

3.5.2 Cost of RMWs

We split the cost of an RMW in two parts: the cost of performing the read and write (R_a/W_a); and the cost of handling the writes in the write-buffer. The average cost of an RMW across the chosen benchmarks for type-1, type-2, and type-3 RMWs is presented in Figure 3.11. As we can see, RMWs are expensive – the average cost of type-1 RMWs is as high as 69 cycles. We also observe that the write-buffer drain significantly contributes to the overall cost of an RMW (58.0% on average). We can infer from this that a significant number of RMWs have at least one write in the write-buffer which needs to send out invalidation requests. Also, a significant number of RMWs are to shared cache-lines which explains the cost contributed by R_a/W_a.

Using type-2 RMWs, the cost of an RMW reduces by 38.6%-58.9% when compared to type-1 RMWs across the benchmarks. As seen from Figure 3.11, a significant portion of the performance improvement is by avoiding the write-buffer drain in the general case. Recall that we revert to a write-buffer drain, when a write hits in the bloom filter. As seen from Table 3.4, the average number of hits of pending writes in

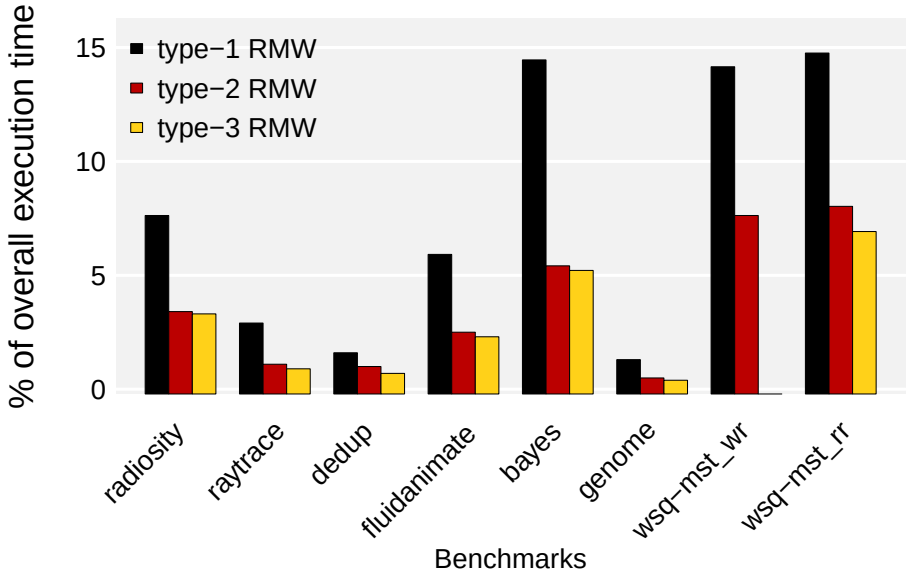


Figure 3.12: Impact of the performance overhead of RMWs on overall performance. Each bar represents the total execution time of RMWs as compared to the overall execution time of the benchmark when using type-1 RMWs.

the bloom filter is negligible for each benchmark, and is sometimes zero. This explains the low write-buffer drain cost for type-2 and type-3 RMWs. It is worth noting that the cost of R_a/W_a itself slightly increases when compared with type-1 RMWs as a portion of the RMWs require broadcasts in addition to the invalidation request. The number of such RMW broadcasts depends on the accuracy of the bloom filter. As shown in the table, the percentage of RMWs that require a broadcast is less than 1.0% for most lock-based benchmarks except for *dedup* (3.1%), which has a higher ratio of unique RMWs to begin with. We have not presented the increase in network traffic due to RMW broadcasts, as this number is negligible across all chosen benchmarks (<0.5%).

Type-3 RMWs reduce the cost of the RMW even further. The average cost of a type-3 RMWs is lower than type-1 RMWs by up to 64.3%. Type-3 RMWs reduce the cost of R_a/W_a but incur a similar write-buffer drain delay as type-2 RMWs.

C/C++11 concurrency. Similarly to lock-based benchmarks, we observe that using type-2 RMWs reduces the average cost of RMWs by 44.6% (write-replacement), and 43.2% (write-replacement) respectively, over type-1 RMWs. As mentioned earlier, type-3 RMWs cannot be used for write-replacement. For read-replacement, type-3

RMWs provide an additional 11.6% improvement over type-1 RMWs.

It is worth noting that the cost of RMWs in read-replacement (*wsq-mst_rr*) is higher than in write-replacement (*wsq-mst_wr*) for all types of RMWs; with read replacement, there are more entries in the write-buffer per-RMW, which increases draining cost. The cost of R_a/W_a, however, is oblivious to whether SC-atomic-read or SC-atomic-write were replaced. In case of type-2 RMWs, we observe that the number of writes conflicting with the bloom filter increases, thereby increasing the cost of an RMW. Also note that this lock-free program, unlike traditional benchmarks, has more RMW broadcasts (3.7%) owing to a relatively larger number of unique RMWs. This affects the performance of type-2 and type-3 RMWs. However, the write-buffer drain cost eclipses the broadcast overhead.

3.5.3 Execution time overhead

Although we achieve a significant reduction in the cost of an RMW in all chosen benchmarks, its impact on the overall execution time depends on the ratio of RMW operations to other memory operations. We call this the *density* of RMWs. Thus, benchmarks with a larger RMW density benefit more from cheaper RMWs. Table 3.4 shows the ratio of the number of RMWs to the number of other memory operations in each of the benchmarks. Figure 3.12 shows the impact of RMWs on the overall execution time for all the chosen benchmarks. As expected, lock-free algorithms suffer more from expensive RMWs than lock-based algorithms. Similarly, *bayes* and *wsq-mst* also spend a lot of time performing RMWs. Although *genome* is a lock-free benchmark, the impact of RMWs on the overall execution time is less owing to a lower RMW density. This is because genome performs a lot more operations per transaction. As for lock-based benchmarks, *radiosity* and *fluidanimate* spend more than 5.0% of their execution time on RMWs. This, however, is not the case with *raytrace* and *dedup*. This is a result of the effort put into optimizing traditional lock-based benchmarks. We can extrapolate that other benchmarks from Splash-2 and Parsec will show an even lesser impact of RMWs.

With type-2 RMWs, we get up to 9.0% reduction for *bayes*, where the write-buffer drain almost but eliminated, as seen from Table 3.4. We also observe a significant

reduction in the contribution of RMWs to the overall execution time in all other lock-free benchmarks as well. Even *radiosity* and *fluidanimate* show a reduction in overall execution time albeit lesser than 4%. Type-3 RMWs further improve the overall performance over type-2 RMWs, but only by a minimal amount (<0.5%).

C/C++11 concurrency. As for the C/C++11 concurrency model, replacing read atomics with RMWs results in a slightly higher overhead of RMWs as can be seen from the figure. The best performance can be obtained by replacing read atomics with type-3 RMWs (7.7% improvement over type-1 RMWs).

In summary, type-2 and type-3 RMWs are significantly cheaper than type-1 RMWs across all chosen benchmarks. This translates to a significant reduction in the overall execution time for the lock-free work stealing queue program which exhibits a higher RMW density. Traditional lock-based programs also show an improvement in performance. This improvement, however, is only visible in programs with a high RMW density. Other benchmarks show a negligible improvement in performance.

3.6 Related Work

Memory ordering. Over the years, researchers have proposed a number of techniques for achieving memory ordering efficiently [Blundell et al., 2009; Gniady et al., 1999; Ladan-Mozes et al., 2011; Lin et al., 2012; Singh et al., 2012]. While any of the above techniques can be used to efficiently implement the barrier-like ordering of a type-1 RMW, the goal of our work, however, is orthogonal. Instead of striving to implement the barrier-like ordering, we ask the question as to why a TSO RMW should be ordered like a memory barrier in the first place. Indeed, as we have shown through our weaker type-2/type-3 RMWs, implementing a barrier-like ordering is not necessary.

Weaker atomicity RMWs. Gharachorloo et al. [Gharachorloo et al., 1993] were the first to observe that it is sufficient for RMWs to use type-3 atomicity in the context of various memory consistency models. However, in order for their TSO specification to be compliant with the original TSO specification, they then added additional program order edges to RMWs, making the RMWs strongly ordered – hence equivalent to type-1 RMWs.

The load-reserve/store-conditional or load-linked/store conditional (LL/SC) instruc-

tion is a classic example of an RMW in weaker models such as Power [Machine and Staff, 1995] which uses type-3 atomicity semantics. None of the mainstream TSO architectures, however, provide such an RMW. However, even if a TSO architecture were to support such an RMW, it would be ordered like a type-1 RMW. This is because instructions following the LL/SC can only be retired after the SC completes, even if the processing (M in RMW) can be performed immediately following the LL instruction. Thus memory operations following such an LL/SC RMW will have to wait for pending writes in the write-buffer, making the store-conditional act as a full barrier. In the case of a type-2 or type-3 RMW, even if the read (R_a) and write (W_a) of the RMW are performed separately, the retirement of R_a implies that instructions following it can retire even before W_a completes in the write-buffer. This is true even in the case where the RMW is followed by a branch (as part of a loop). Once R_a has performed, the branch instruction following it can also be performed and retired. Even if the RMW fails in the context of the program, the next *RMW loop* can begin performing. However, in the case of an LL/SC, the success of LL alone cannot guarantee the successful execution of the RMW. Thus, even if an LL/SC (eventually) succeeds, the memory instructions following it have to be stalled until the LL/SCs success. This explains the barrier-like behaviour of an LL/SC instruction in TSO. **Hardware locking mechanisms.** There have been several proposals (e.g. [Vallejo et al., 2010]) which address issues related to hardware based locking mechanisms. It is worth noting that these locks refer to the synchronization primitive as a whole and not the RMW instructions used in these primitives. These proposals primarily deal with lock contention and fairness. Our proposal is orthogonal to such work as we deal with the overhead added by the RMW to the local thread.

3.7 Summary

We observed that the atomicity semantics of an RMW is the key factor which affects the RMW's ordering semantics, its programmability, and its implementation cost. Existing TSO RMWs use a strict definition of atomicity (type-1) which results in the RMW being strongly ordered like a memory barrier. Whereas type-1 RMWs are costly to implement, they can be easily used in synchronization idioms on TSO without re-

quiring additional memory barriers. In this work, we proposed two weaker atomicity definitions: type-2 and type-3 atomicity; we formally derived how type-2 and type-3 RMWs would be ordered, and demonstrated that the resultant ordering is strong enough to implement various synchronization idioms using the weaker RMWs. We then proposed efficient architectural implementations of the weaker RMWs – experimental results show that our proposed type-2 RMW (type-3 RMW) is 58.9% (64.3%) cheaper than an existing type-1 RMW on average.

Based on our analysis and experimental evidence, type-2 RMWs, while performing almost as well as type-3 RMWs, are also able to seamlessly replace existing type-1 RMWs in common synchronization idioms – except in situations where an RMW is used as a memory barrier. Thus, they appear to be a promising alternative to existing type-1 RMWs. We also show how the proposed type-2 and type-3 RMWs can be used to implement C/C++11 atomics – thus making it possible for the compiler to transparently utilize the proposed RMWs to realize C/C++11 more efficiently.

Chapter 4

SuperCoP: Memory supervision with correctness and performance

This chapter deals with the issue of memory supervision, where we tackle the problem of correctness of supervision in weaker consistency models. As described in the first two chapters, supervised memory systems make use of auxiliary data associated with each memory address (or blocks of memory addresses) in order to store monitoring information regarding that particular address. This monitoring information is used by supervised memory systems in order to perform important tasks such as synchronization [Bobba et al., 2008], race detection [Savage et al., 1997] error detection [Venkataramani et al., 2009, 2008] etc.

Supervised memory systems make use of RMW operations to operate atomically on both data and metadata. We revisit the formulation of Supervised Memory Instructions (SMIs) presented in chapter 1. A generic supervised memory instruction (SMI) is shown in Figure 4.1. As we can see, each memory read (memory write) is accompanied by additional memory operations which read metadata, process metadata (which can potentially trigger an exception), and (optionally) update metadata; furthermore, the entire sequence of data and metadata operations has to be 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*. As shown in Figure 4.1, each SMI is an RMW.

Supervised Memory Read (SMR)	Supervised Memory Write (SMW)

Atomic
{
Read Data (R_d)
/* Read & process Metadata */
Read Metadata (R_m)
if (Metadata == ...)
exception()
Metadata = f(Metadata)
Write Metadata (W_m)
}
	Atomic
	{
	/* Read & process Metadata */
	Read Metadata (R_m)
	if (Metadata == ...)
	exception()
	Metadata = f(Metadata)
	Write Metadata (W_m)
	Write Data (W_d)
	}

Figure 4.1: Supervised Memory Instructions

The supervision model for the state-of-the-art hardware-assisted supervised memory system (Memtracker [Venkataramani et al., 2009]) was also presented in the background chapter. Such a supervision model assumes sequential consistency. Adding features such as write-buffering which enables a weaker consistency model like total-store-order (TSO), introduces certain correctness issues in the supervised memory system like *imprecise exceptions* and *metadata read reordering*. The root cause for these correctness issues lie in the supervision model and how SMWs and SMRs are *ordered with respect to each other and with other non-supervised instructions*. In this way SMIs behave like the type-1 RMWs described in chapter 3. Here, SMIs are ordered like memory fences. While there are existing proposals which address these correctness issues [Bobba et al., 2011], they either exhibit poor performance or cannot be applied universally to all supervision schemes. Even in such cases, the SMIs are still ordered like memory fences. For increased performance, the ordering is restricted to SMIs and not regular read/write operations. Such a modification cannot be applied to all supervision schemes as shown by the authors themselves. Thus, there is a need to design a supervised memory system which provides correctness without compromising on performance. In this regard, we present SuperCoP - a supervision model which overcomes the above correctness issues for all supervision schemes without trading-off

on performance.

The rest of the chapter is organized as follows. We first describe the correctness issues that manifest in supervised systems for TSO consistency. We then provide our solution - SuperCoP - for these correctness issues followed by how SuperCoP provides correctness without compromising on performance. Now, SuperCoP changes the existing metadata-data atomicity present in Memtracker. Consequently, we present our solution for the metadata-data atomicity issue in SuperCoP. Finally, we present the simulation results comparing the performance of SuperCoP, TSOall, and TSOdata.

4.1 Introducing the Problem

When we extend existing supervised memory systems, like Memtracker, to weaker consistency models, it is prudent to discuss the supervision model of the weaker consistency model. In our case, we consider the total-store-order (TSO) consistency model which is prevalent in the popular x86 processor architectures used in Intel, AMD and Sun SPARC processors. We start with discussing the base TSO supervision model and then follow it up with the correctness issues that arise in memory supervision due to the consistency model.

4.1.1 Base Model for a TSO Supervised Memory System

The TSO consistency model, as discussed in chapter 2, is widely used in present day systems, including Sun's SPARC, Intel's/AMD's x86 and its variants. The base system model we consider is similar to that in *Safe Supervised Memory* [Bobba et al., 2011], which has a supervision mechanism similar to Memtracker [Venkataramani et al., 2009]. The Memtracker proposal itself does not make use of a write-buffer as shown in chapter 2. 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

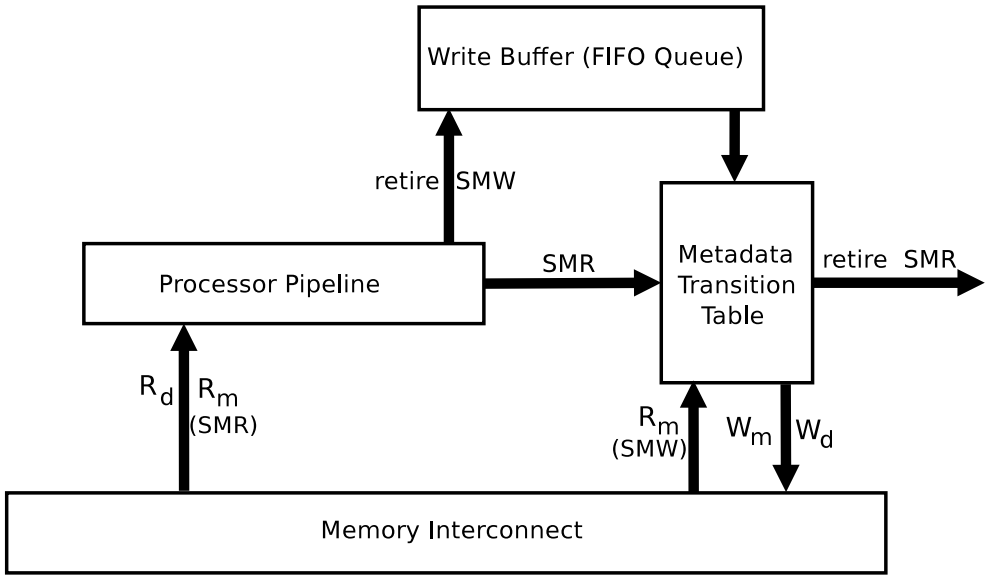


Figure 4.2: Base model for a TSO based supervised memory system. An SMR is retired after it has completed its data and metadata operations. An SMW is retired into the write-buffer, and performs its data and metadata operations on reaching the head of the write-buffer.

system model is described in Figure 4.2. Metadata-data atomicity is preserved in the same way as in Memtracker i.e. metadata/data and metadata/data writes are issued together. Atomicity between the read and write operations are ensured with a locking mechanism.

4.1.2 Correctness issues in a TSO supervised memory system

Bobba et al. [Bobba et al., 2011] 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.

Please recall the full-empty bits supervision scheme that was discussed in chapter 2 to explain the Memtracker supervision system and metadata-data atomicity. We use the full-empty bits supervision scheme to illustrate both the correctness issues discussed above. Consider the sequence of instructions shown in Figure 4.3(a). Here, SMR_3(B) (from T_0) synchronizes with SMW_4(B) (from T_1), to ensure that SMR_5(A) (from T_1) reads the value written by SMW_2(A) (from T_0). Initially, both A and B are in the *full* state. In the expected execution sequence, T_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*. T_0 then performs SMR_3(B) which *empties* address B. This is followed by T_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 T_1 reads the value written by SMW_2(A) i.e. r_3=2.*

Now, it is possible that T_0 is stalled or blocked, resulting in T_1 executing its instructions first. In such a case, T_1 first tries to perform SMW_4(B). In Memtracker

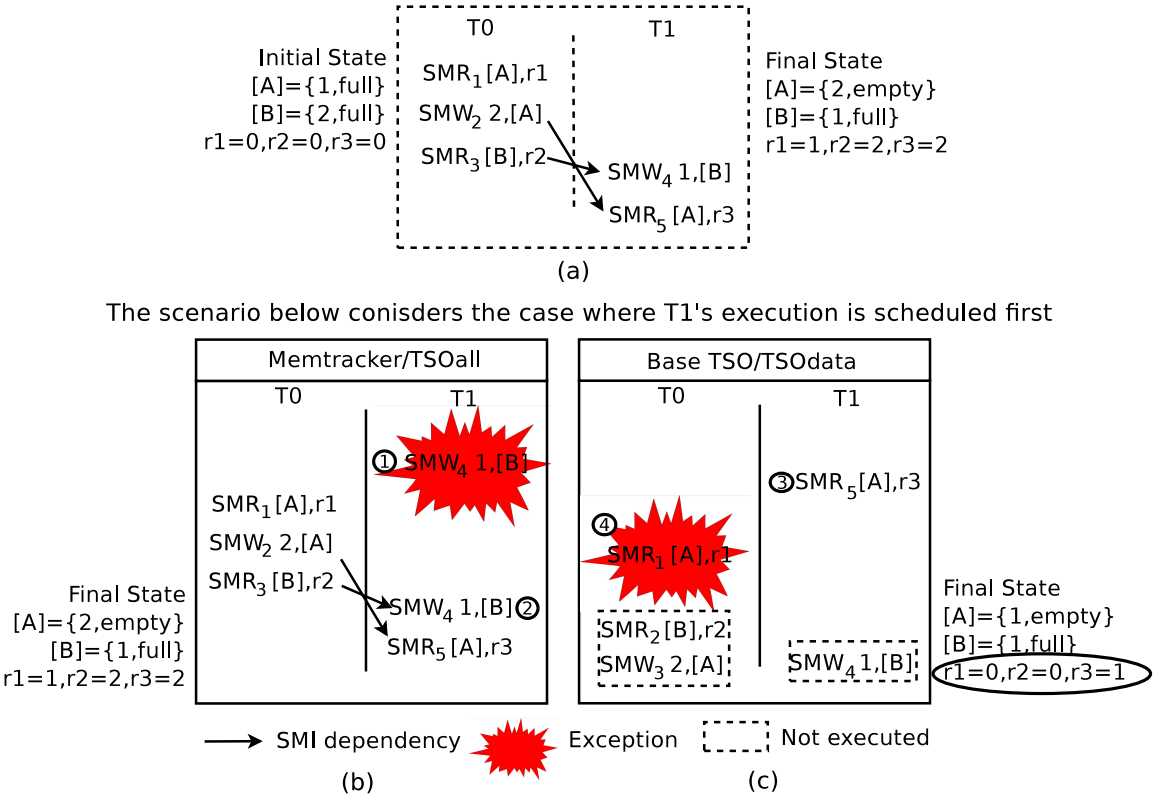


Figure 4.3: (a) shows the intended sequence of execution. (b) and (c) show the execution sequence in Memtracker and the base TSO model, when T_1's execution is scheduled before T_0. In Memtracker/TSOall, SMW$_4$(B) results in an exception ① blocking the thread until SMW$_3$(B) is performed. This results 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 4.3(b)), this immediately raises an exception ① and T_1 is blocked until this instruction can successfully execute. T_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 T_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 4.3(c)), however, T_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 T_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.

4.1.3 Existing proposals which address correctness Issues in a TSO supervised memory system

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 4.2. 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 [Zhou et al., 2007]. 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. In this regard, we discuss the ordering requirement of

SMIs in the supervision model and show how it can be weakened in order to provide increased performance.

4.2 SuperCoP - Correctness of memory supervision for TSO consistency

4.2.1 Revised Ordering Requirement of SMIs

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. However, such an ordering requirement results in poor performance similar to sequential consistency. Since each SMI is both a read and a write, the existing ordering under the TSO consistency model orders all normal reads, writes and SMIs with other SMIs. Each SMI acts like a memory fence like in type-1 RMWs shown in chapter 3.

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 alone are to 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. SuperCoP does not enforce any additional ordering as compared to the underlying TSO consistency model. The only ordering which is relaxed is the write to read ordering. This extends to both data and metadata. Thus, metadata or data reads can be reordered with respect to metadata or

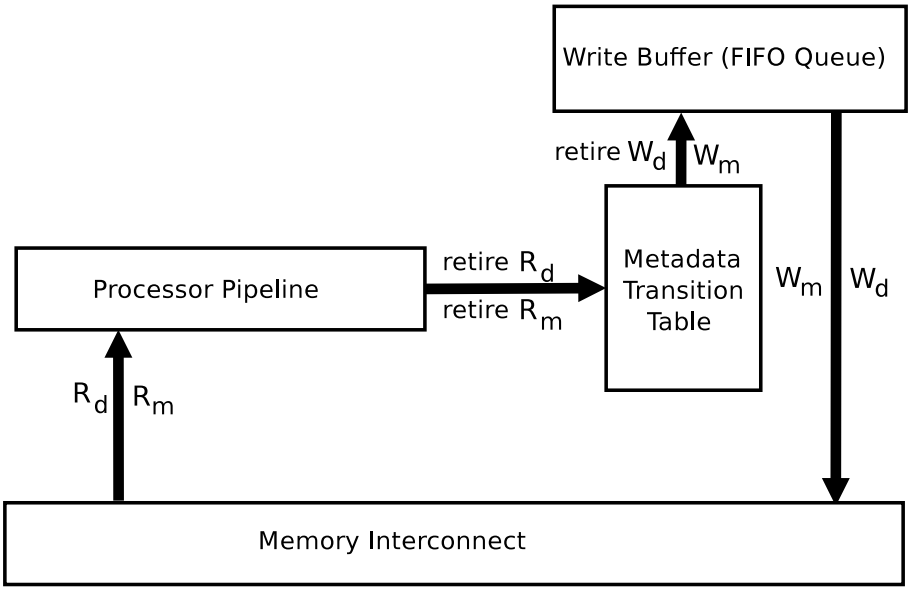


Figure 4.4: Implementation of SuperCoP. All SMIs are separated into their constituent operations and are retired separately. For an SMR, R_m/R_d are retired once they are completed, and W_m is retired into the write-buffer. For an SMW, R_m is retired once the read is complete. W_d/W_m are retired into the write-buffer.

data writes. SuperCoP only modifies the execution of SMIs to reflect this ordering. This correctness requirement is applicable to weaker consistency models as well. As long as metadata read ordering is maintained and metadata-data atomicity is guaranteed, data reads and/or data writes can be reordered with respect to other data reads and/or reads.

It is worth noting that, SuperCoP orders SMIs similar to the type-2 RMWs described in chapter 3. The performance improvement of SuperCoP when compared with TSOall is also similar to that achieved by type-2 RMWs over type-1 RMWs.

4.2.2 Implementation of SuperCoP

The implementation of SuperCoP is illustrated in Figure 4.4. 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

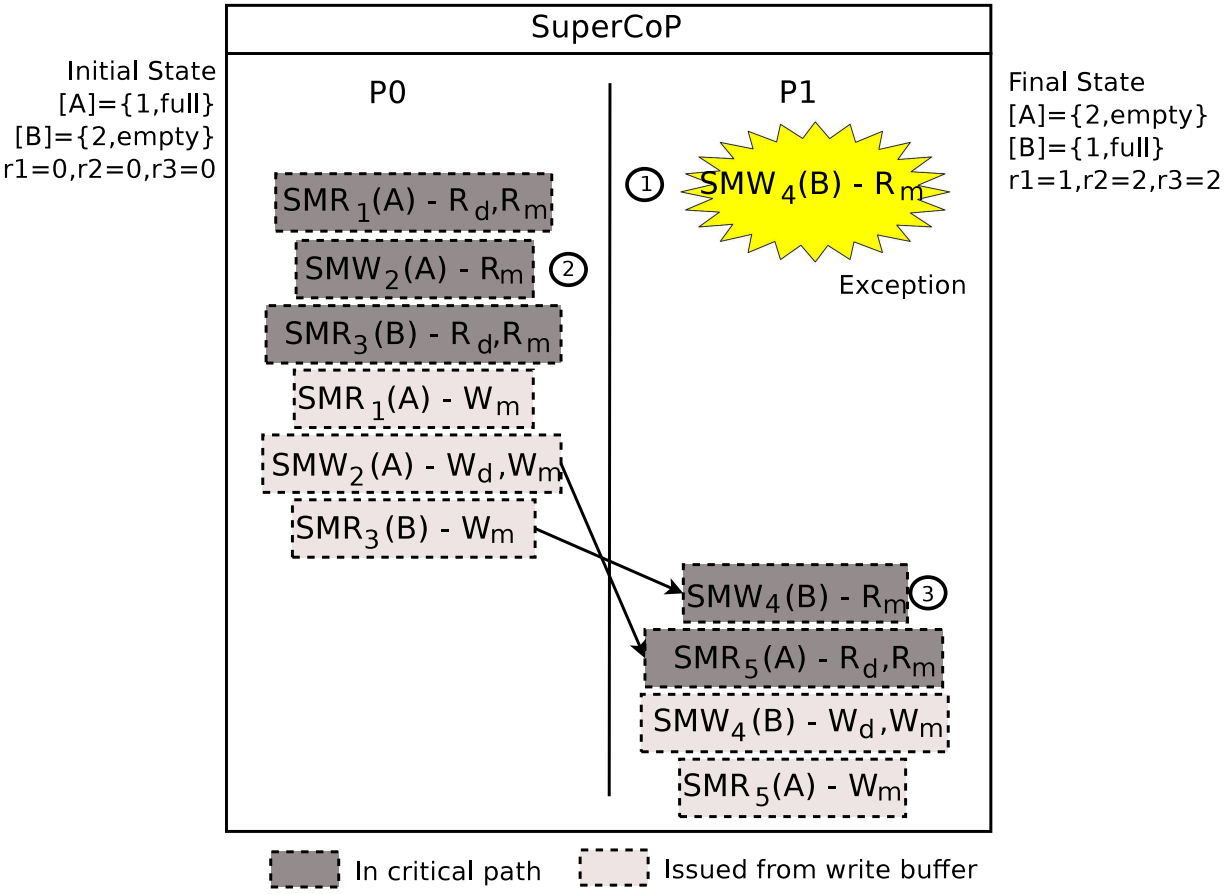


Figure 4.5: If T_0 is scheduled before T_1, SMW$_4$(B) performs its R$_m$ first, which raises an exception ①. The thread is then blocked until this R$_m$ can succeed. In T_0, SMR$_1$(A) retires its metadata write to the write-buffer, from which SMW$_2$(A) reads its metadata ②. SMW$_4$(B) can finally 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.

the processor continues with its execution. This removes W_m from the critical path. An SMW also performs its R_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.

4.2.3 How correctness is achieved in SuperCoP

Consider the same sequence of instructions as in Figure 4.3. The execution order as per SuperCoP is shown in Figure 4.5. Assuming the same scenario (where T_0 has been stalled or blocked), T_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 T_1 until the metadata of B becomes *empty*. Now when T_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*, T_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 T_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.

4.3 Ensuring Atomicity in SuperCoP

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 4.6. For ease

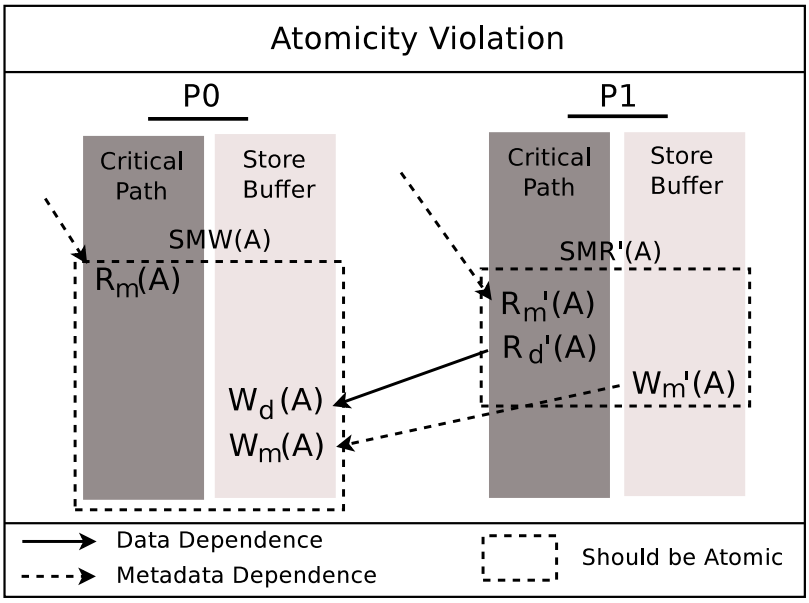


Figure 4.6: R_m (from T_0) retires and inserts W_m and W_d into the write-buffer. Meanwhile, R'_m (from T_1) is performed and reads the same metadata value as R_m, instead of reading the metadata value written by W_m, as W_d is performed after R'_d.

of explanation, we use a generic supervision scheme instead of the full-empty bits scenario.

Here, T_0 performs SMW(A), and T_1 performs SMR′(A). First, T_0 performs and retires R_m(A) in the critical path. Once the metadata processing is done, T_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 T_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.

4.3.1 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

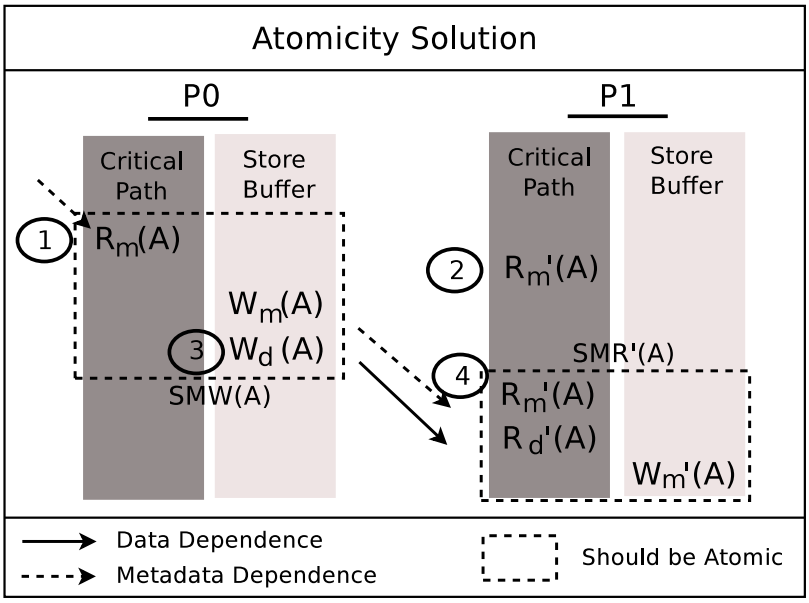
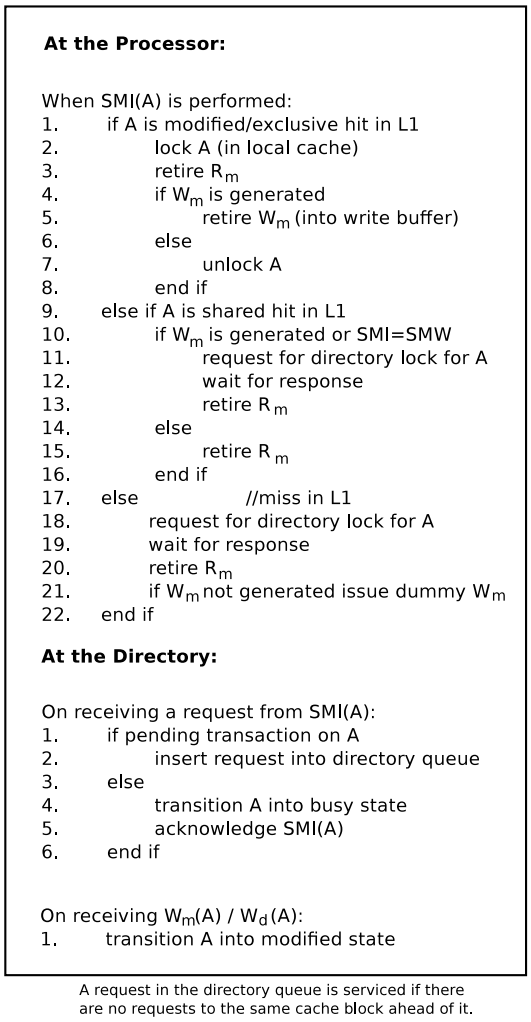


Figure 4.7: R_m locks address A ① prohibiting R'_m from being performed ②. W_m unlocks A ③ following which R'_m is performed successfully ④. Thus, R'_m correctly reads the metadata written by W_m.

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 an SMI until it completes. An execution pattern for a generic fine grain locking based atomicity scheme is illustrated in Figure 4.7. 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.



At the Processor:

When SMI(A) is performed:
1. if A is modified/exclusive hit in L1
2. lock A (in local cache)
3. retire R_m
4. if W_m is generated
5. retire W_m (into write buffer)
6. else
7. unlock A
8. end if
9. else if A is shared hit in L1
10. if W_m is generated or SMI=SMW
11. request for directory lock for A
12. wait for response
13. retire R_m
14. else
15. retire R_m
16. end if
17. else //miss in L1
18. request for directory lock for A
19. wait for response
20. retire R_m
21. if W_m not generated issue dummy W_m
22. end if

At the Directory:

On receiving a request from SMI(A):
1. if pending transaction on A
2. insert request into directory queue
3. else
4. transition A into busy state
5. acknowledge SMI(A)
6. end if

On receiving $W_m(A)$ / $W_d(A)$:
1. transition A into modified state

A request in the directory queue is serviced if there are no requests to the same cache block ahead of it.

Figure 4.8: Protocol for the proposed Directory locking mechanism to preserve metadata-data atomicity.

4.3.1.1 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 4.7, 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 T_0. An SMI performs invalidations only when a metadata write is generated. 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.

4.3.1.2 Atomicity using directory locking

To address the drawbacks in cache locking, 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 Fig-

ure 4.8. 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 scenario) The working of the directory locking based atomicity scheme is illustrated in Figure 4.9, where T_0 performs SMW(A) and T_1 performs SMR$'$(A) (as shown in Figure 4.7). Assume that A is initially shared between T_0 and T_1. First, SMW(A) (from T_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 T_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 T_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 T_0's write-buffer, ⑧ T_1's copy of A is invalidated, and ⑨ the lock on A

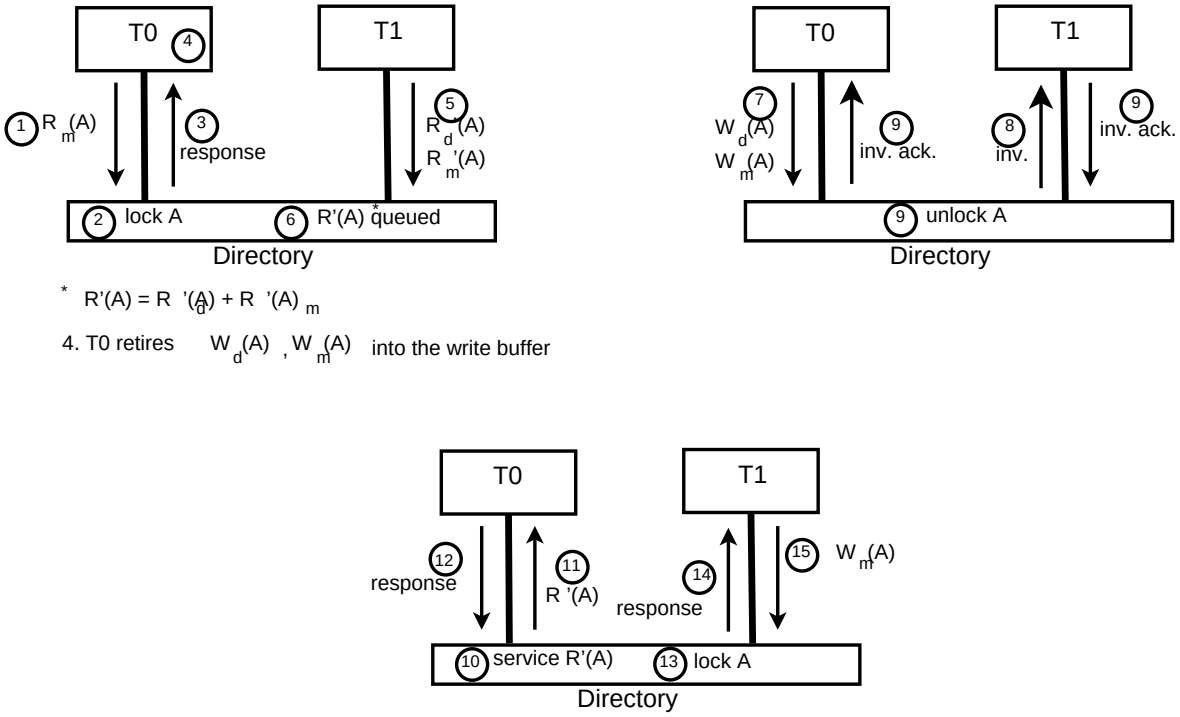


Figure 4.9: Here, T_0 performs SMW(A) and T_1 performs SMR′(A). The sequence of events are numbered in the ascending order. The figure on the left shows how SMW(A) locks address A, and SMR'(A) is queued in the directory. The figure in the center shows steps involved in unlocking A. In the figure on the left, SMR′(A) locks A, when it is serviced by the directory. Eventually, SMR′(A) issues its metadata write to unlock A.

is relinquished by T_0. The directory transitions the cache block A to modified state owned by T_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 T_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 T_0 and T_1 having copies of it. ⑮ Eventually, SMR′(A) issues its metadata write to unlock A in the directory.

4.4 Experimental Results

4.4.1 System Specification

We built a hardware simulation infrastructure using the PIN tool [Luk et al., 2005], 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 [Woo et al., 1995] benchmark suite. The benchmarks and their respective input sizes are listed in Table 4.1. We evaluate SuperCoP with both cache based locking and directory based locking.

4.4.1.1 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. [Zhou et al., 2007]. HARD is used 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

Table 4.1: 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

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. [Bobba et al., 2011] uses HARD to compare TSOdata and TSOall.

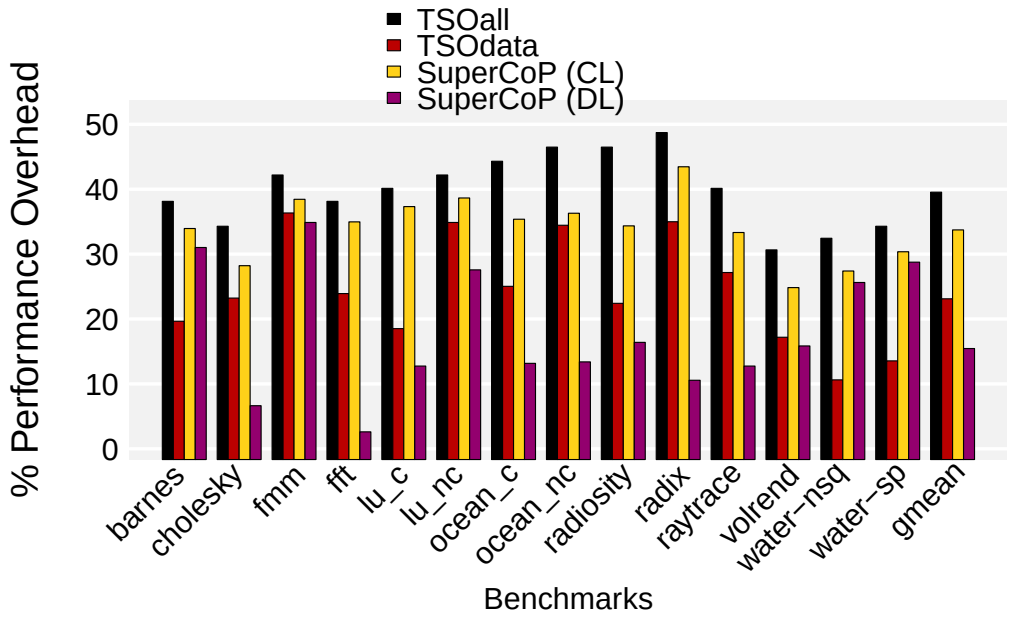


Figure 4.10: Performance overhead comparison for the HARD supervision scheme. The performance numbers here represent the overhead suffered by the supervised system when compared to an unsupervised system. For example, TSOall has a 38.89% overhead compared to an unsupervised execution of the *barnes* benchmark. CL and DL represent the cache locking scheme and directory locking schemes, respectively.

4.4.2 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 4.10. The percentage of SMIs which update metadata varies from 0.2% (*fmm*) to 57.8% (*lu-contiguous*) as shown in Table 4.2, with an average (geometric mean) of 4.5%. We observe from Figure 4.10 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%, *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

Table 4.2: 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

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.

4.4.3 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 4.11 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 SuperCoP with the cache locking scheme as it is evident from the results of the HARD supervision

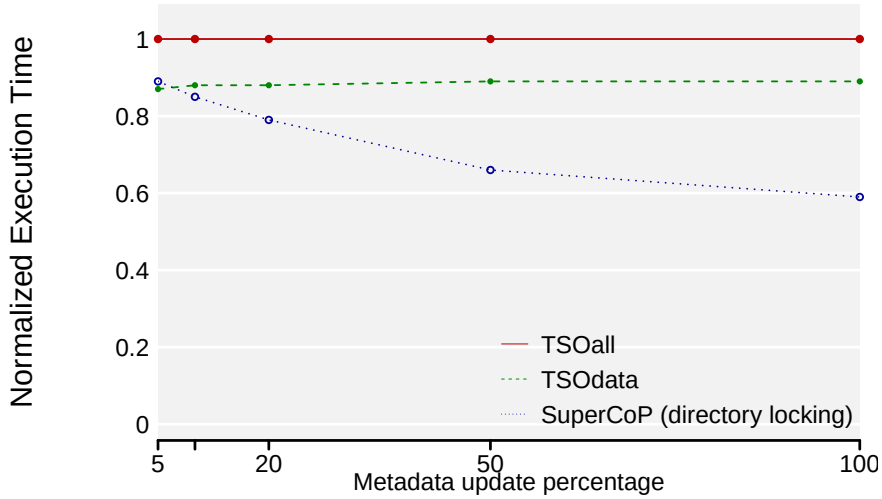


Figure 4.11: Scalability of TSOall, TSOdata, and SuperCoP with respect to the number of SMIs which update metadata. The execution time is normalized to that of TSOall to better represent the scalability. We only present SuperCoP with directory locking as we have shown that it consistently performs better than the cache locking scheme.

scheme that the cache locking scheme performs as bad as TSOall.

Figure 4.11 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. 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.

4.5 Summary

Existing supervised memory systems, implicitly or explicitly, assume SC [Bobba et al., 2011]. 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.

Chapter 5

DyFCoM: Performance and Fairness for Lock-free Programs

This chapter addresses the issue of contention in the case of RMWs used in synchronization and profiling applications. Please recall from the first chapter that, apart from the orderings enforced by an RMW, contention also poses a bottleneck for RMW performance. While we observe that contention of RMWs themselves is not a big problem, the way in which RMWs are used in synchronization (and profiling) increases the performance overhead. The programs we examine use conditional RMWs (like CAS, TAS, etc.) to enable synchronization (or counting). These programs are structured such that a successful conditional RMW from a thread will trigger the failure of RMWs from concurrent threads, resulting in the re-execution of certain pieces of the code in the concurrent threads, leading to a performance overhead. In addition, when contention is involved, the fairness amongst concurrent threads in accessing shared objects is also an issue to contend with. We use the term *contention-based failures* to describe this issue.

In this work, we present **D**yFCoM: a **D**ynamic **F**airness and **C**ontention **M**anagement scheme, to monitor the success and failure of RMWs and manage contention in order to provide increased performance and fairness. We target finer-grained synchronization mechanisms like lock-free programs, as contention is of greater concern in such cases. We also evaluate our scheme with profiling applications using RMWs to update shared counters.

In this chapter, we first present and motivate the problem of contention. This is followed by the dynamic contention management scheme we propose to alleviate the performance overhead owing to contention. Next, we present our dynamic fairness management scheme. The contention management and fairness management schemes are combined to form DyFCoM. Finally, we present our experimental results for DyFCoM when compared with similar contention management schemes, with regards to both performance and fairness.

5.1 Introducing the problem

The poor scalability of traditional blocking algorithms led to the design of non-blocking algorithms [Cederman et al., 2013a,b; Michael, 2002; Michael and Scott, 1996]. Such algorithms, unlike blocking algorithms, ensure that threads competing for an update to a concurrent data structure do not have their execution indefinitely postponed. Most existing non-blocking algorithms provide a *lock-free* progress guarantee [Herlihy and Shavit, 2008], ensuring that among all threads that compete to update the data structure, at least one will succeed – thereby ensuring system-wide progress. For other progress guarantees (wait-freedom and obstruction-freedom), please refer to the background chapter. We focus on lock-free algorithms due to the large existing corpus of programs, and its widespread usage.

Unfortunately, lock-free programs suffer from scalability issues with regard to both throughput and fairness. Indeed, several works in literature [Cederman et al., 2013a; Dice et al., 2013a; LaMarca, 1994; Morrison and Afek, 2013] show that on increasing contention, throughput and fairness fall significantly. We observe a similar pattern for performance and fairness in our experiments conducted on a 32-core Intel Xeon processing platform as evidenced from Figures 5.1 and 5.2. We present the throughput and fairness for micro-benchmarks operating on a concurrent queue, stack and a shared counter. The fairness values fall to as low as 0.53 in the case of a lock-free stack when running 64 threads. In case of performance, the throughput of the stack and queue micro-benchmarks fall by a factor of as much as 70 going from 2 threads to 64 threads. The shared counter micro-benchmark falls by around a factor of 36 from 2 to 64 threads. We explain these benchmarks in detail in the section on evaluation

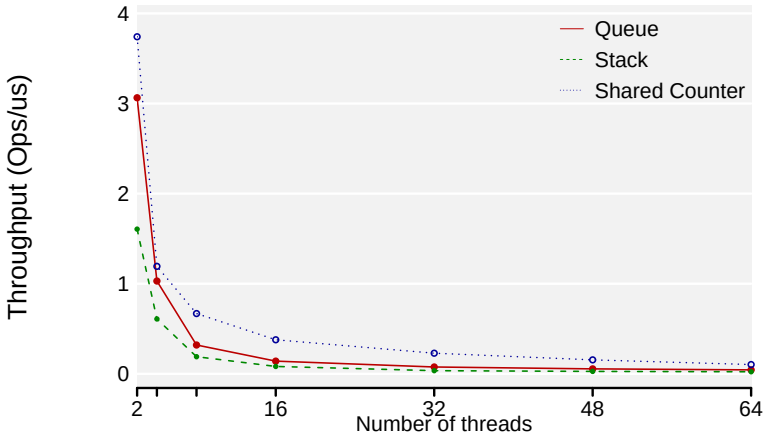


Figure 5.1: This graph shows the variation of throughput of a lock-free queue,stack and shared counter. Throughput is measured as the time taken for each lock-free operation averaged across all threads. Please note that we present only enqueue operations for queues and push operations for stacks. The results are similar for dequeue and pop operations. Each thread performs either 1 million enqueue or dequeue operations for queues. Similarly for stacks, each thread performs either 1 million pushes or 1 million pops.

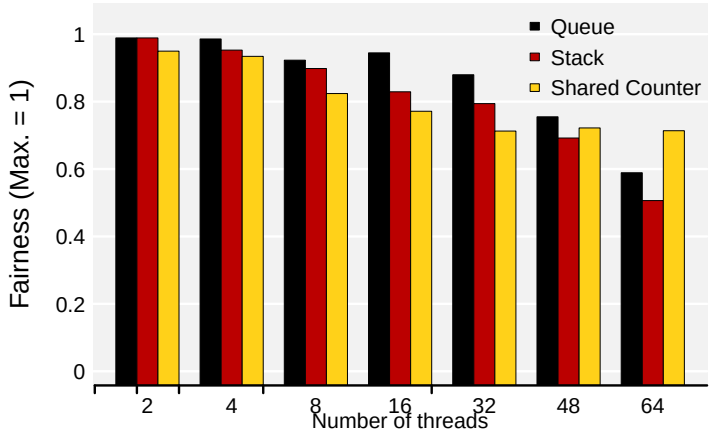


Figure 5.2: This graph shows the variation of fairness of a lock-free queue,stack and shared counter. Each thread performs 1 million lock-free operations.

methodology.

We attribute the degradation in performance and fairness to the contention of RMWs from concurrent threads. Most existing lock-free algorithms [Cederman et al., 2013b; Gidenstam et al., 2010; Michael and Scott, 1996] synchronize using RMW loops (especially compare-and-swap (CAS) loops): a thread observes the shared state, performs a computation, and uses a CAS to update the shared state (called the *linearizing CAS*). If the linearizing CAS succeeds, this read-compute-update sequence appears to be atomic; otherwise the thread must retry the entire loop. This is shown in Algorithm 1. A CAS loop represents a lock-free operation. Here x is the *shared* variable, and y is a *thread-local* variable. x is first read into y at the beginning of the CAS loop. For the loop to be successful, $CAS(x,y)$ should be successful. If x has been written by another thread before the CAS is performed, the CAS loop (or operation) fails. Please note that threads only use RMWs to access x. The requirement of *atomicity* of the CAS loop implies that only one thread at a time can successfully complete a CAS loop. The remaining CAS loops will result in failures of the CAS, and thus the respective threads must re-perform their CAS loops. This is the main cause for the performance overhead. With increased contention, the number of CAS failures will increase which results in poor throughput. This increased contention can be in the form of increased number of concurrent threads, or reduced amount of work done between successive

synchronization operations. Indeed, existing literature also attributes the degradation in performance in lock-free programs to CAS failures [Dice et al., 2013a; Morrison and Afek, 2013].

Algorithm 1 General structure of a Lock-free operation

```
loop
  Set y ← x
  Perform Computation
  CAS(x, y) //Linearizing CAS
  if CAS successful then
    Exit Loop //Thread progress
  end if
end loop
```

The issue with fairness is due to the nature of the lock-free progress guarantee. Since lock-free programs only guarantee system-wide progress, individual threads can *potentially starve*, or *run ahead* of other threads. Increase in contention increases the possibility of a thread to starve or run ahead. In either case, the access of the shared data structure is unevenly distributed across the concurrent threads. Although higher throughput is the generally preferred, maintaining fairness is equally important, especially in NUMA architectures where the latency of memory access amongst threads is different especially between different NUMA nodes. Fairness, here, indicates the availability of shared resources (in this case - atomically accessed cache-lines) to different threads. High fairness means that all threads have an *equal* opportunity to progress by accessing the shared resource. This is preferable in order to reduce starvation and unequal distribution of work.

Typically, fairness and performance are considered orthogonal wherein optimizing one parameter compromises the other. In such a case, a dynamic scheme which constantly or periodically monitors the execution of the program can possibly address both issues in a holistic manner. In this work, we study the impact of contention on throughput and fairness and propose a universal mechanism which provides both high throughput and fairness under high contention, irrespective of the program, or the synchronization construct used. We start by showing how the existing state-of-the-art con-

tention management scheme [Dice et al., 2013a] is highly sensitive to the contention and the program structure, We follow this with our dynamic contention management scheme and show how it overcomes the drawbacks of existing schemes. We also show how contention resolution affects the fairness measure. We then present our fairness management scheme. Please note that the fairness measure we address is not a strong guarantee, but a weak one where we try and ensure that all threads have equal opportunity in updating the concurrent data structure. Both the dynamic management schemes involve monitoring successful linearizing RMW/CAS operations from each thread. We use this information to implement a dynamic backoff-based contention management to provide increased throughput. For fairness, we force-fail CAS loops deliberately so that *potentially starving* threads get an opportunity to progress, over faster threads. A potentially starving thread is one which performs a lesser number of CAS loops than other threads in a given period of time. We finally present the combination of both fairness and contention management (DyFCoM) and evaluate its performance and fairness when compared with existing schemes.

In summary, our contributions are as follows:

- We show that the throughput of existing CAS contention management schemes is dependent on the level of contention, concurrency (and the architecture). We also show that the CAS contention management scheme impacts the fairness values adversely in many cases.

- We propose DyFCoM, a dynamic fairness and contention management scheme, which monitors successful RMWs, periodically determines the throughput of individual threads, and dynamically alters backoff values for contention management. DyFCoM addresses fairness by holding back threads which have a higher throughput and allowing potentially starving threads to complete their updates to the shared data structure (in the presence of lesser contention).

- We also present a practical implementation of DyFCoM in the form of an RMW library. In order to achieve increased throughput and fairness, programmers have to simply replace the linearizing CAS operation in existing lock-free programs with the CAS function provided by us. We augment the CAS function with the required monitoring, backoff and force-fail methods in order to implement DyFCoM.

5.2 Dynamic Contention Management for Performance

This section first describes the existing state-of-the-art contention management scheme proposed by Dice et al. [Dice et al., 2013a] and its drawbacks. Then we present our dynamic contention management scheme which addresses these drawbacks.

5.2.1 RMW/CAS contention management

Dice et al. [Dice et al., 2013a] proposed a lightweight software-based contention management scheme which specifically addresses CAS operations. They study the impact of typical contention management schemes like *constant backoff*, *exponential backoff*, *time-slicing* and *array-based signaling* when they are applied to the CAS operations used in lock-free algorithms. Out of the contention management approaches present in this paper, the three simplest algorithms - constant backoff, exponential backoff and time-slice - yielded the best results, primarily because they had very small overheads. In this thesis, we present only the constant backoff scheme as it is the simplest to implement and has the least parameters to deal with. Also, it provides the best performance of all contention management schemes proposed in the paper. The more complicated approaches like array-based signaling were consistently outperformed by the simpler approaches.

Listing 5.1: CAS library function

```
1  CAS(long *mem, unsigned long old, unsigned long new) {
2          unsigned long r;
3          asm volatile ("lock cmpxchgl %k2,%1"
4                        : "=a" (r), "+m" (*mem)
5                        : "r" (new), "0" (old)
6                        : "memory");
7    if (r == old)
8      return 1;
9    else
10     return 0;
11 }
```

Listing 5.1 shows the code for a CAS operation as used in programs. The function *CAS(mem,old,new)* takes in the reference to the memory address (*mem*) whose contents are to be compared with the value *old*. If the data stored in *mem* is equal to *old*, then the *cmpxchgl* instruction (x86 assembly language instruction for a CAS) is successful and writes *mem* with the value store in *new*. The *cmpxchgl* sets or resets the zero flag depending on its success, which is typically then checked to see if the CAS is successful or not. As part of the code, however, we store the value in *mem* in a local variable *r*, which is then compared with *old* to indicate success (or failure) of the CAS.

Listing 5.2: CAS library function

```
1  CAS(long *mem, unsigned long old, unsigned long new) {
2          unsigned long r;
3          asm volatile ("lock cmpxchgl %k2,%1"
4                        : "=a" (r), "+m" (*mem)
5                        : "r" (new), "0" (old)
6                        : "memory");
7    if (r == old)
8      return 1;
9    else {
10     usleep(WAITING_TIME);
11     return 0;
12   }
13 }
```

Listing 5.2 shows how the constant backoff contention management scheme is applied in the case of CAS operations. No per-thread state is required for this algorithm. The only difference of a constant backoff CAS as compared with a normal CAS is that if the native CAS fails, then the thread busy-waits for a platform-dependent period of time, after which the CAS operation returns.

A drawback of this technique, which the authors also state, is that the backoff value is platform dependent. Additionally, as we observe, the backoff value is also dependent on the concurrency level (number of threads), and the level of contention, or the lock-free program in question. Figure 5.3 shows the improvement in performance of a lock-free queue when using the constant backoff contention management. We choose

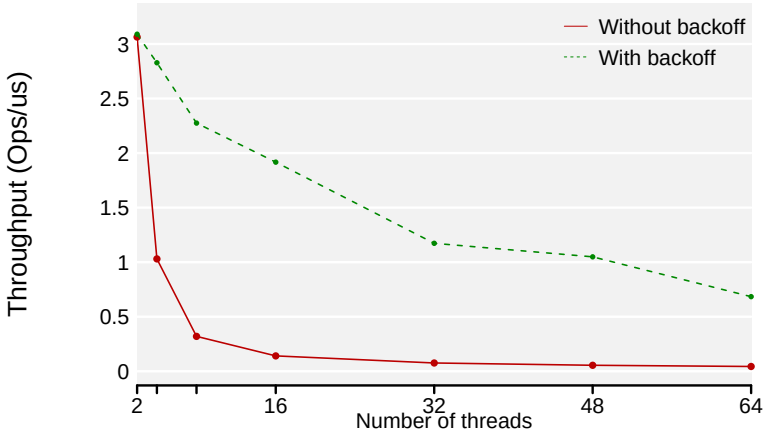


Figure 5.3: This graph shows the variation of throughput of a lock-free queue with and without the constant backoff contention management scheme. Each thread performs 1 million lock-free operations.

a backoff value of 250ns after iterating the experiments through several backoff values. These results show that improving the contention performance of RMWs improves the overall throughput of the program.

To study the sensitivity of the backoff value with concurrent level, we manually varied the backoff value for each of the chosen thread counts. For each concurrency level, we chose the highest throughput achieved with contention management. The values presented in Figure 5.4 are the backoff values that are used to achieve this optimal throughput. As it can be seen the optimal backoff value varies from 20ns to 2000ns. Thus, fixing the backoff value based on the architecture alone is not effective. The backoff values also vary with the lock-free algorithm in use.

Thus, tuning these parameters per data-structure and concurrency level may yield better results. Moreover, a dynamic tuning may provide a general, cross data-structure, cross CPU, solution.

5.2.2 Monitoring CAS success/fails for tuning Backoffs

During our experiments, we observed an interesting pattern in the variation of throughput with the backoff value, given the number of threads. Figure 5.5 shows the sensitivity of throughput with backoff value for a lock-free queue running 32 threads. We vary

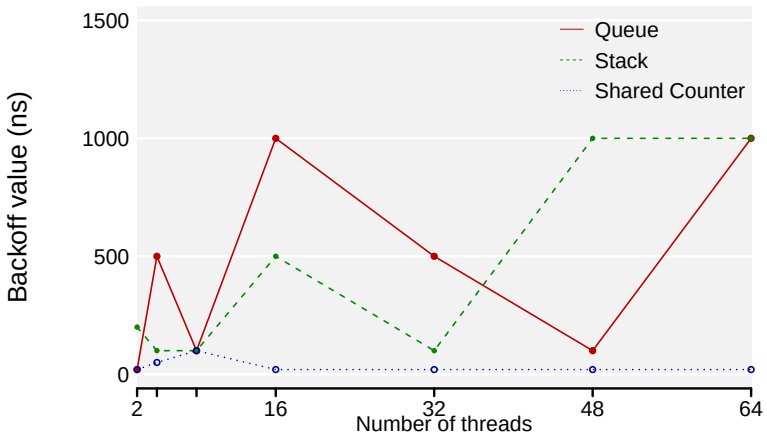


Figure 5.4: This graph shows the variation of optimal backoff for a lock-free queue. Each thread performs 1 million lock-free operations.

the backoff values from 10ns to 5000ns. From the graph, it is clear that the throughput increases on initially increasing the backoff value. Throughput reaches a maximal point (backoff of 500 ns) and then decreases on further increasing the backoff value. We attribute this to the nature of how constant backoff contention management works. In constant backoff, the thread which performs a failed CAS waits for a predetermined time before continuing. During this waiting period, another thread which has already started its CAS loop iteration has more chance to succeed. However, if the waiting period is higher than the CAS loop latency, throughput gets degraded. Such a pattern can be leveraged to design a dynamic contention management mechanism without using any complex learning algorithms. Therefore, in our dynamic contention management scheme, we need to gradually increase the backoff value until it reaches peak performance, and reduce backoff if the performance dips again on increasing backoff.

In order to achieve dynamic tuning, we first need a framework to provide some runtime information about the program execution. Since the CAS instructions are the core of the issue, we add thread-local counters to track the number of successes and failures of the CAS instructions. The success of a CAS indicates the progress of a thread, and failure indicates contention wherein the thread backs off before continuing. Please note that only the *linearizing CAS* has to be monitored. As specified in the background chapter, the linearizing CAS is the point at which the CAS loop (or lock-

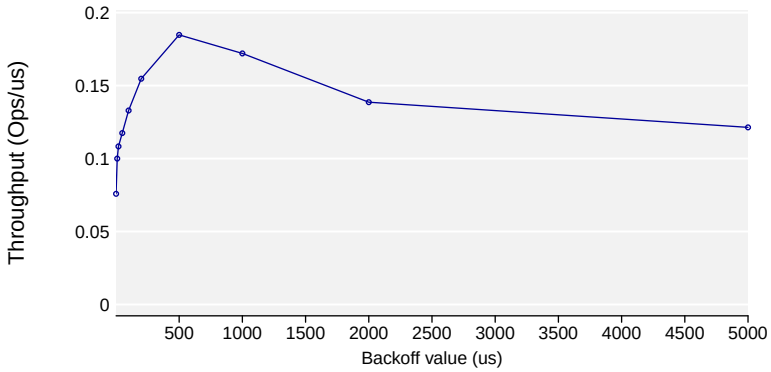


Figure 5.5: This graph shows the variation of throughput of a lock-free queue on varying the constant backoff contention management scheme. The throughput increases initially on increasing backoff, reaches a maximum value and falls again. Our scheme proposes to achieve this maximal throughput.

free operation) is deemed to have completed atomically. This is shown in Listing 5.3.

Listing 5.3: CAS library function

```
1  #define CAS_EPOCH 1000
2
3  unsigned long num_fail, num_success;  //thread−local variables
4  unsigned long long start_time, cur_time, epoch_time; //thread−local variables
5
6  static __inline__ unsigned long long rdtsc(void) { //function to read the timestamp counter
7    unsigned hi, lo;
8    __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
9    return ( (unsigned long long)lo)|( ((unsigned long long)hi)<<32 );
10 }
11
12 //start_time is initialized to the value of the timestamp counter at the start of execution.
13 CAS(long *mem, unsigned long old, unsigned long new) { //modified CAS function.
14           unsigned long r;
15           asm volatile ("lock cmpxchgl %k2,%1"
16                         : "=a" (r), "+m" (*mem)
17                         : "r" (new), "0" (old)
18                         : "memory");
19   if (r == old) {
```

```
20
21      num_success++;
22      cur_time=rdtsc();
23      if(num_success >= CAS_EPOCH) {
24        epoch_time = cur_time−start_time;
25        start_time=rdtsc();
26      }
27      return 1;
28    }
29    else {
30      num_fail++;
31      return 0;
32    }
33  }
```

We also use the hardware timestamp counters available in x86 architectures [Intel Corporation, 2009] to obtain the time taken to complete a certain number of CAS operations. We define a *CAS_EPOCH* variable to hold the number of successful CAS loops (1000 in Listing 5.3) whose latency is required. If the number of successful CAS instructions is greater than this value, the time difference from *start_time* and *cur_time* is stored in *epoch_time*, and *start_time* is reset to 0. *start_time* should initially be set before the CAS loop. With this setup we have the number of successful/failed CAS instructions, and the amount of time it takes to complete *CAS_EPOCH* number of CAS loops. With this, we can obtain the throughput for every *CAS_EPOCH* number of CAS instructions.

5.2.3 Proposed Approach to Dynamic Contention Management

With the monitored information that we obtain from the framework described above, we formulate our approach to tuning the backoff value dynamically as shown in Figure 5.6. We use the variable *direction* to keep track of whether we have to increase or decrease the *backoff* value. Initially, *direction* is set to 1, *epoch_time_old* is set to 0, and *backoff* is set to 1. Assume that we sample every 1000 CAS successes. Once 1000 CAS successes is reached, we first check *direction* variable to check whether we

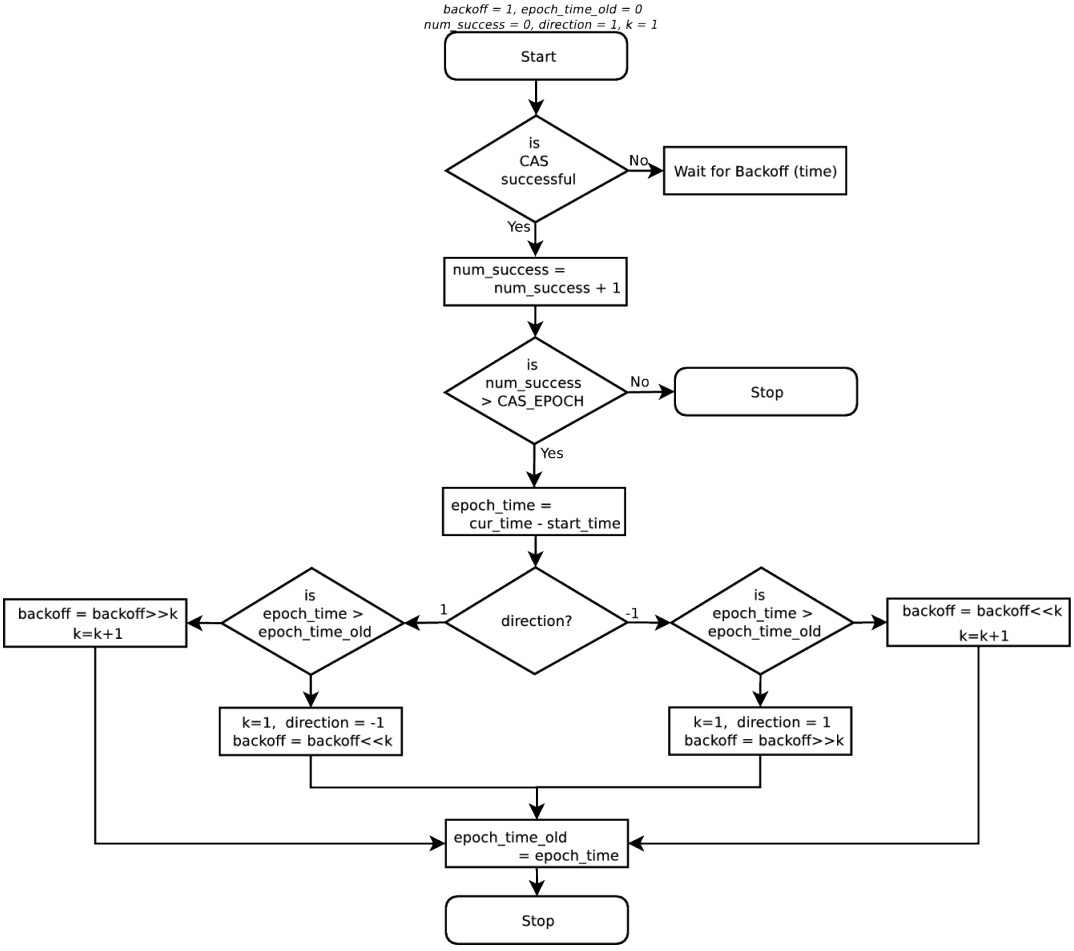


Figure 5.6: Dynamic Contention Management for Performance. The process detailed in this flow chart is performed for every CAS instruction. Initial values for the variables are as follows: direction = 1, epoch_time_old = 0. backoff = 1, backoff scaling factor k = 1, num_success = 0.

have to increment or decrement the backoff. Now if the new recorded *epoch_time* is lesser or equal to old epoch time i.e. the time taken to perform 1000 CASs is lesser in this set of 1000 than the previous set, then we increase the backoff value by a factor in multiples of 2 (starting with 2).

This process will continue until the peak throughput is achieved, after which the throughput tapers off as seen from Figure 5.5. In such a case, the new *epoch_time* will be greater than the old epoch time. At this point, the *direction* is changed to -1, and we start to decrease the backoff value starting from a factor of 2 again. The mechanism operates similarly when the *direction* is -1. However, the criterion for when to increase or decrease backoff and the next change in direction differs.

For example, we start with the backoff value, direction, and *epoch_time_old* set to 1, 1, and 0ns respectively. Considering a pattern of change similar to Figure 5.5, where the optimal backoff value is around 500ns, the throughput increases until the backoff value reaches 1024ns. On the next set of 1000 successful CAS operations, the backoff value will increase to 32768ns (1024ns x 64). With this backoff value, the throughput is lesser than with a backoff of 1024ns. In this scenario, the direction is 1, and the new epoch time is greater than the old epoch time. Thus, we start reducing the backoff value and change the direction to -1. The new backoff value will be 16384ns (32768ns / 2). On the next set of 1000 successful CAS operations, the backoff value will further fall to 4096ns, reduced by a factor of 4. The next backoff value in progression is 256ns after which the direction will change again and reach 512ns. The proposed mechanism will settle around this value.

We change the scale with which the backoff value is varied so that we prevent shooting over the optimal point of operation.

We evaluate this dynamic contention management scheme with the constant backoff contention management scheme. We study the sensitivity of the performance to the sampling frequency. The other parameters are dynamically tuned and need not be manually varied. The experimental results are provided at the end of this chapter. In summary, we present a dynamic contention management mechanism wherein we continuously monitor number of successful CAS instructions and the time taken to perform a certain number of CASs, at a thread-local level. We use this information to gradually tune the backoff value to achieve optimal throughput across different

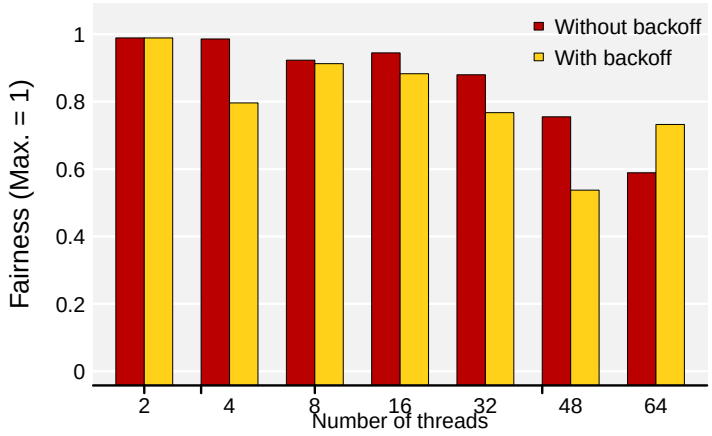


Figure 5.7: Fairness with constant backoff

processor architectures, concurrency levels, and lock-free algorithms.

5.3 Dynamic Fairness Management

In addition to the need for tuning, contention management also skews the fairness values and not necessarily in a positive manner. Figure 5.7 shows the change in fairness values for a lock-free queue when constant backoff is applied. The fairness value [1] drops to as low as 0.562. Thus, if fairness is required along with performance, a contention management scheme must work in tandem with a fairness management scheme. Indeed, the mechanism we propose does exactly this, as described in the following section.

Fairness, unlike throughput, is a trickier problem to tackle. For increased performance, we required only thread-local counters and support variables. Whereas for fairness, it is necessary to know the state of progress across all concurrent threads. The problem of fairness is also different from performance in that the strength of fairness required can be vastly different. In this work, we focus on a weaker fairness measure, which is calculated on the basis of how far ahead or how far behind are certain threads with respect to other threads.

We follow an approach similar to our dynamic contention management scheme in

[1] Fairness is measured as explained in chapter 2. A fairness value of 1 indicates perfect fairness, and a value of 0 indicates no fairness.

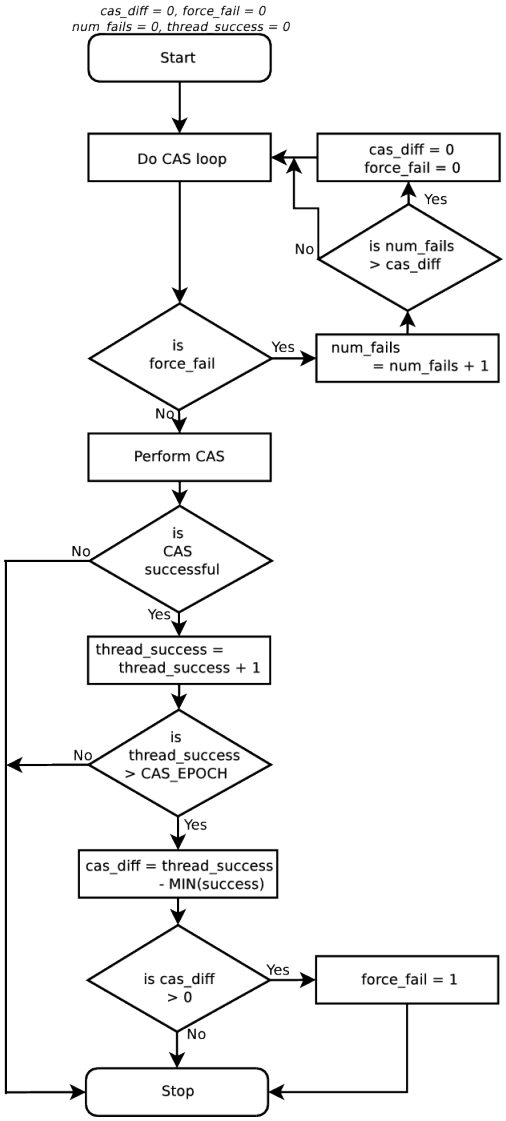


Figure 5.8: In this flow chart variable force_fail, cas_diff and num_fails are initially set to 0. MIN(success) indicates the minimum number of successful CASs across all threads. thread_success indicates the number of successful CASs from that particular thread.

order to solve the issue of fairness in lock-free programs. Unlike contention management, however, we make use of the monitored performance metric across all processors in order to maintain a level of fairness. The broad outline of our dynamic fairness management mechanism is as follows. Similarly to contention management, we track the number of successful CAS instructions. Once every *CAS_EPOCH* number of CAS instructions, the fairness algorithm checks the time taken to perform so many CAS loops, across all threads. With this information, we use a concept we call *force-failing*, where if we identify a thread to be ahead of the slowest thread, then we deliberately fail the CAS loops of the faster threads by not performing the linearizing CAS. The number of times we force-fail a CAS loop depends on how far ahead the thread is when compared to the slowest thread.

This fairness management mechanism is illustrated in Figure 5.8. In this figure, we use a variable *force_fail* to indicate whether a thread is to deliberately fail its CAS loop or not. *force_fail* is set if the current thread has more successful CAS instructions than the slowest thread, represented by *cas_diff*. The number of successes of the slowest thread is represented by *MIN(success)*. In the force-failing path of execution, we increment the number of forced failures stored in *num_fails*, and check if this value is greater than *cas_diff*. If so, then the force-fail variables are reset and we continue with normal execution.

This leads to a very valid question of why we do not introduce a waiting time for the faster threads instead of force-failing. The force-failing approach is more flexible than the waiting approach, as stricter fairness guarantees can be ensured with this mechanism. For example, in Taubenfeld's work, fairness requires that a thread that performs a successful CAS loop waits for all concurrent CAS loops (that fail) to complete before performing its next CAS loop. Introducing a waiting period once a thread completes its CAS loop will not solve this issue. In such a scenario, we just check if the previous CAS loop of a thread was successful and if there are failed CAS loops in other threads. If this criterion is satisfied, then the thread which performed the successful CAS loop force-fails until all other threads register a successful CAS loop. The force-failing acts like a polling mechanism to continuously check if the thread can continue with its regular execution. It is worth noting that such an execution will guarantee a fair lock-free progress guarantee only with a benevolent scheduler as with Taubenfeld's approach.

We evaluate our dynamic fairness management scheme with the fairness levels of stock lock-free programs, with and without contention management. We study the sensitivity of the fairness to the sampling frequency. The other parameters used in the protocol are dynamically tuned and need not be manually varied. The experimental results are provided at the end of this chapter. In summary, we present a dynamic fairness management mechanism wherein we continuously monitor number of successful CAS instructions and the time taken to perform a certain number of CASs, at a thread-local level. We then use this information to compare the execution times of CAS loops across threads, and proceed to throttle faster threads in order to provide increased access of shared variables to slower threads. Also, due to its dynamic nature, this approach works across architectures, concurrency levels, and lock-free algorithms.

5.4 Evaluation Methodology

In this section, we discuss the platform used for evaluating both our dynamic fairness and contention management schemes. We also discuss the benchmarks used in detail.

5.4.1 Execution Platform

We performed our experiments on an Intel based workstation with 8 sockets of the quadcore Intel Xeon L7555 (Nehalem) [Thomadakis, 2011] running at 1.87GHz. Each processing core can support 2 hardware threads, taking the total count to 32 processing cores, and 64 logical cores. The processing system is supported by 64 GB of DDR3 main memory running at 1333MHz. The Intel machine is provided with Quick-Path Interconnect for connectivity between chips and I/O subsystem. In simultaneous multithreading (SMT) mode, two threads can share the resources on each physical core.

5.4.2 Benchmarks

In our experiments, we used the following programs to evaluate DyFCoM:

- **L**ock-free concurrent queue: We use the version of a lock-free queue presented by Michael Scott and Maged Michael [Michael and Scott, 1996]. A brief listing of this

algorithm was shown in the introduction chapter. For the queue data structure, CAS loops are used to insert elements (enqueue) and remove elements (dequeue) from the queue. Like traditional queues, two pointers, front and rear (or head and tail), are used to track the status of the queue. An enqueue operation inserts elements at the front pointer, and a dequeue operation removes elements at the rear pointer.

- **L**ock-free concurrent stack: We adopt the lock-free stack algorithm presented by Treiber [Treiber, 1986] for use in our experiments. Unlike the queue algorithm, both of the two basic operations involving a stack (push and pop) work on a single pointer (top). Since the purpose of this work is to study performance and fairness under contention, we do not implement more advanced stack algorithms, elimination-backoff stack (EB stack) [Shavit and Touitou, 1995]. EB stacks reduce contention by pairing push and pop operations and eliminating them without even updating the stack.

- **S**hared concurrent counter: Each thread in the shared counter algorithm we implement, simply updates a shared variable using a CAS instruction. We first read the shared counter value and update it with a CAS, making the update a CAS loop. We also add an extra *tag* to the counter to avoid the ABA problem. The ABA problem occurs when multiple threads (or processes) accessing shared memory interleave with each other. A thread, on reading from a memory address twice, sees the same value stored in the address indicating that the value has not changed. However, between the two reads, the data could have changed twice returning it to its previous value. For example, consider that thread t_0 reads value A from address x. Thread t_1 modifies the value in x from A to B and back to A. Now, thread t_0 reads from x again and sees the value A. Thread t_0 does not see the update of x by thread t_1. The shared counter algorithm represents all profiling applications using RMWs.

For our experiments, we used the lock-free versions of a concurrent queue, concurrent stack, and a shared counter as mentioned above. Each of the programs is a micro-benchmark where all threads performs lock-free operations on a shared concurrent data structure. In each case we structured our experiments in such a way that each thread performs 1 million successful CAS loops. In order to calculate the throughput we consider the cumulative time taken by all threads in completing their lock-free operations, and average it over the sum of the number of CAS loops performed suc-

cessfully. In case of the queue and stack benchmarks, we assign each thread as either a producer or a consumer. In case of a queue, the producer enqueues 1 million values to the shared queue, and the consumer performs 1 million dequeues. For a stack, the producer performs only push operations, and the consumer performs pop operations. For the shared counter, all threads increment a shared counter 1 million times in a CAS loop. The level of contention is represented by the concurrency (number of threads), which we vary from 2 to 64 in each of the experiments. It is worth noting that the threads do not perform any work in between consecutive CAS loops other than regular thread maintenance.

The throughput is measured as the number of successful lock-free operations i.e. CAS loops per unit time period. The fairness is measured using the metric prescribed in [Ha et al., 2007], as discussed in chapter 2. Please recall that the maximum value of 1 for fairness indicates a perfectly fair execution where each thread has an equal opportunity to perform their RMWs, and do perform their RMWs in the fairest manner. A low fairness value indicates either a thread that runs ahead of other threads or a thread that starves.

5.5 Experimental Results

In this section we present the results of the experiments we conducted based on the evaluation methodology described above. We first present the results for our dynamic contention management scheme in comparison with Dice et al.'s constant backoff scheme. We follow this with an evaluation of our fairness management scheme. Finally we combine both the fairness and contention management schemes (DyFCoM), and evaluate the combined scheme in terms of both fairness and throughput.

5.5.1 Dynamic Contention Management

5.5.1.1 Lock-free Queue

Figure. 5.9, shows the throughput of lock-free queue algorithm in terms of number of operations per microsecond for the proposed dynamic contention management scheme (MSQ-DCM) and the constant backoff scheme (MSQ-CB). For MSQ-CB, we manu-

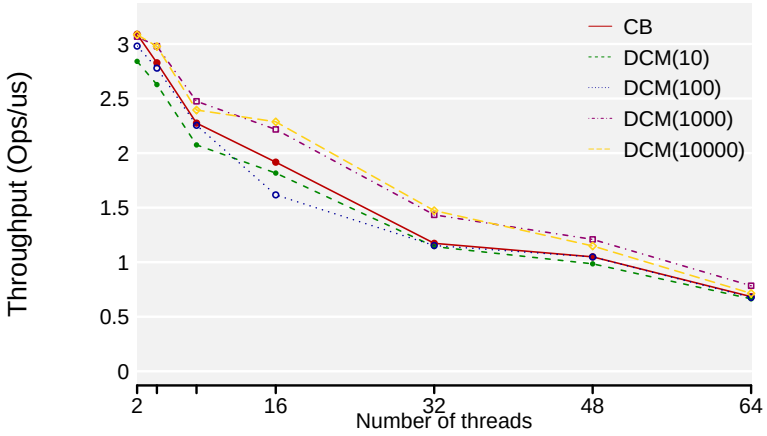


Figure 5.9: This graph shows the variation of throughput of a stock lock-free queue (MSQ), with constant backoff (MSQ-CB), and with dynamic contention management scheme (MSQ-DCM). The parameter indicates the sampling frequency.

ally tuned the backoff parameter for each of the execution points to provide optimal performance. For the DCM scheme, we varied the sampling frequency from 10 to 10000. The sampling frequency is the number of successful RMW operations after which the algorithm decides whether to change the backoff delay or not. We can see from the figure that for a sampling frequency of 10, the overhead of MSQ-DCM outweighs any performance improvement over MSQ-CB. MSQ-DCM(10) consistently performs worse than CB by upto 8.05%. Decreasing the sampling frequency (increasing the number of RMWs sampled) to 100 shows a slight performance improvement over MSQ-DCM(10). However, in some cases MSQ-DCM(100) performs worse than MSQ-DCM(10) despite the reduced overhead of the DCM algorithm. This can be attributed to the dynamic execution pattern, and also some interference from the underlying operating system.

MSQ-DCM(1000) clearly outperforms MSQ-CB in most cases barring for the scenario with 2 threads where it performs worse than CB by 0.71%. However, we also see performance gains of upto 22.20%. MSQ-DCM(1000) provides an average performance improvement of 11.60% over MSQ-CB. Although we chose the optimal backoff delays for CB in our experiments, the increase in performance of DCM can be attributed to the fact that DCM manipulates the backoff delay on a per-thread basis

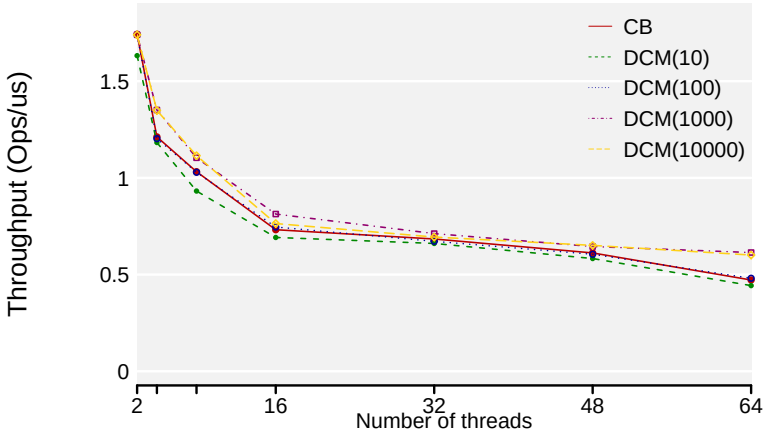


Figure 5.10: This graph shows the variation of throughput of a stock lock-free stack with constant backoff (TS-CB), and with dynamic contention management scheme (TS-DCM). The parameter indicates the sampling frequency.

and thus can tune each thread individually as compared to a constant backoff for all threads. While increasing the sampled RMWs further to 10000 shows performance gains for some concurrency levels (up to 25.57% over CB), it averages less than MSQ-DCM(1000) (9.89%). Higher sampling numbers does not necessarily mean an increased performance (due to reduced algorithm overhead), as the backoff delays are changed with a lesser frequency. The final performance gains depend on the execution pattern of the program, and cannot be theoretically ascertained.

5.5.1.2 Lock-free Stack

Lock-free stacks (Figure. 5.10) show a similar trend like with lock-free queues. TS-DCM(1000) shows a performance gain of up to 30.06% compared to TS-CB (average 9.84%). Stacks, in general have a lesser throughput as compared to queues due to increased contention for the single status pointer (top). Both the push and pop operations contend for this pointer. In queues, enqueue operations contend with other enqueue operations and dequeue operations contend with other dequeues. TS-DCM(10000) provides a slightly lesser performance advantage as compared with TS-DCM(1000) ($<$ 1%) It is clear from these results that for both stacks and queues, a sampling number of 1000 consistently provides a better performance than CB. We choose this sampling

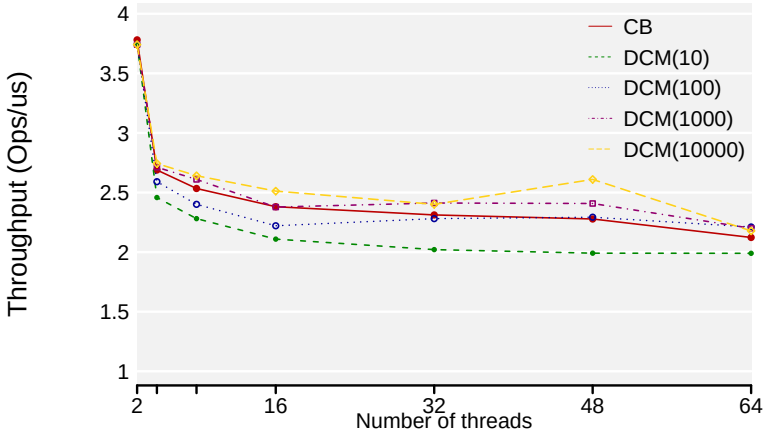


Figure 5.11: This graph shows the comparison of throughput of a stock lock-free shared counter (LFC) with constant backoff (LFC-CB), and with dynamic contention management scheme (LFC-DCM). The parameter indicates the sampling frequency.

point in our experiments for fairness management along with contention management.

5.5.1.3 Shared Counter

The shared counter microbenchmark, while showing similar trends (Figure. 5.11) to queues and stacks, differs from them in that a sampling number of 10000 (LFC-DCM(10000) provides almost twice the performance improvement (4.53%) provided by LFC-DCM(1000) (2.33%). Also LFC-DCM(1000) performs worse than CB for more concurrency levels than for queues and stacks.

5.5.2 Dynamic Fairness Management with contention management

We combine our dynamic fairness management (DFM) scheme with the dynamic contention management (DCM) scheme in order to ascertain the trade-offs between fairness and performance in DyFCoM. We only use the sampling point of 1000 RMWs in the following experiments as it shows a consistently good performance improvement for DCM over CB.

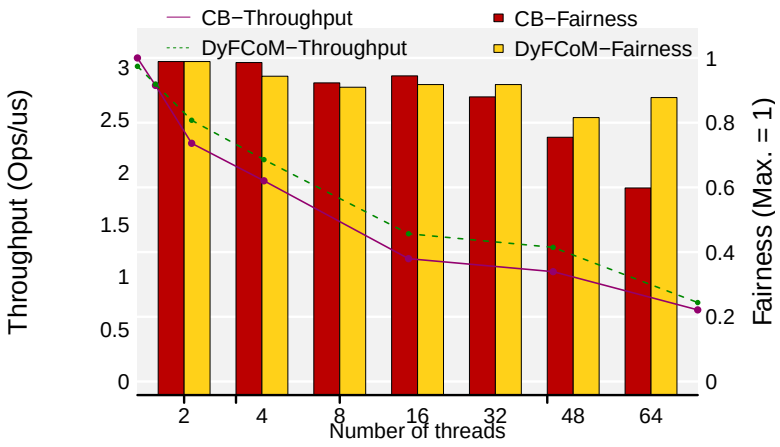


Figure 5.12: This graph shows the comparison of fairness and throughput of a queue with constant backoff (CB), and with dynamic fairness and contention management scheme (DyFCoM).

5.5.2.1 Lock-free Queue

Figure 5.12 shows the comparison of DyFCoM(1000) with CB for a lock-free queue with regards to both throughput and fairness. The throughput shows a similar trend to DCM, although the increase in throughput differs from that of DCM. We only get an average of 6.87% performance improvement for DyFCoM compared to CB. This is because of the additional monitoring and control operations to be done for fairness management on top of contention management. For fairness, we get an improvement of up to 43.64% for DyFCoM. There are cases where DyFCoM achieves slightly less fairness than CB. These are cases where CB already has a high level of fairness ($>$ 0.93).

5.5.2.2 Lock-free Stack

Unlike with the queue, for a stack we improve fairness for all concurrency levels, achieving an average improvement of 10.00% (Figure 5.13. However, similarly to a queue, the performance improvement drops by around 3.71% when compared to DCM(1000). However, we do achieve an average performance improvement of 6.14%.

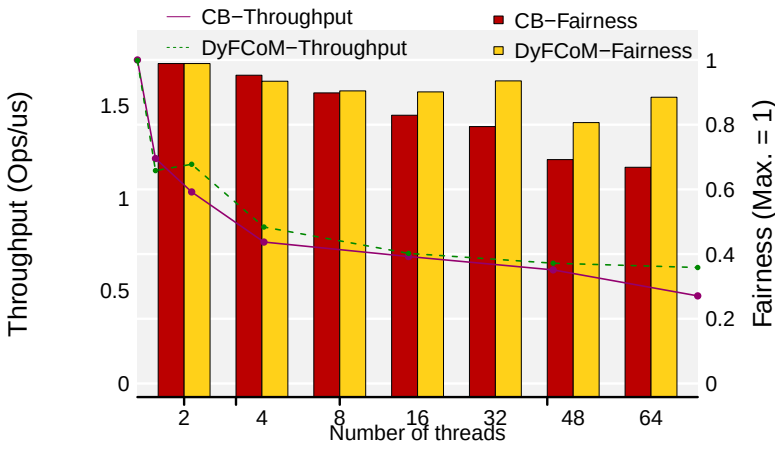


Figure 5.13: This graph shows the comparison of fairness and throughput of a stack with constant backoff (CB), and with dynamic fairness and contention management scheme (DyFCoM).

5.5.2.3 Shared Counter

For a shared counter, the performance drop is much larger compared to DCM than for stacks or queues. The average performance improvement for DyCoM over CB is as low as 0.73% for the counter micro-benchmark. On the other hand, we observe that the fairness improvement is much larger (avg. 12.66%).

In summary, on average we get a performance improvement of 3.55% (up to 32.38%) for DyFCoM over CB across all the benchmarks considered and across all concurrency levels. With DCM alone we get an average performance improvement of 7.93%. With regards to fairness, we show fairness improvements of up to 43.64% for DyFCoM over CB. In the cases where the fairness level of DyFCoM is slightly less than CB, the fairness is already very high, and thus the slight reduction is negligible. This shows that DyFCoM provides better performance and fairness across different lock-free algorithms, and concurrency levels without manually tuning the contention management parameters.

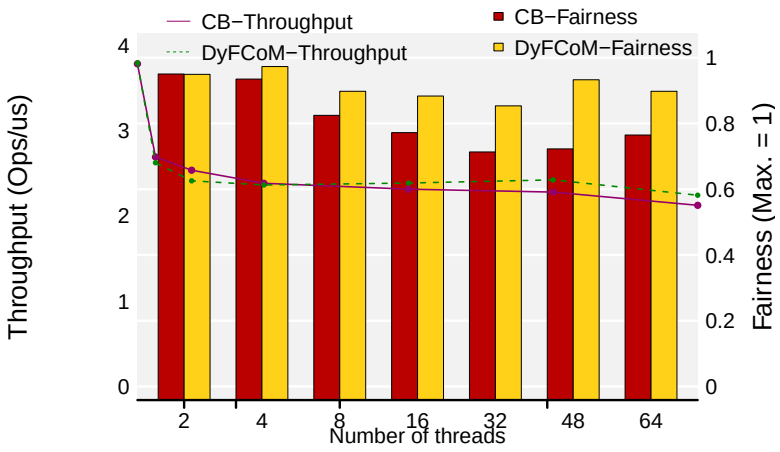


Figure 5.14: This graph shows the comparison of fairness and throughput of a shared counter with constant backoff (CB), and with dynamic fairness and contention management scheme (DyFCoM).

5.6 Related Work

There has been considerable work highlighting the issue of RMW contention in lock-free data structures [Dice et al., 2013a; Morrison and Afek, 2013; Misra and Chaudhuri, 2012; Cederman et al., 2012], all of which address the degradation in performance with respect to contention. Dice et al. [Dice et al., 2013a] showed that simple software contention mechanisms applied over CAS operations will greatly improve performance under heavy/medium contention. However, they suffer a small overhead under light contention. Also, the backoff parameters used is heavily dependent on the processor architecture, contention, and concurrency. We overcome these drawbacks by varying the backoff parameters dynamically depending on the measured throughput of each thread. We also address the issue of fairness, which Dice et al. neglect.

Morrison and Afek [Morrison and Afek, 2013] proposed concurrent FIFO queues using fetch-and-add (FAA) instead of CAS in order to reduce the performance overhead due to failed CAS operations (FAA operations always succeed). Such an approach provides very good performance improvement, sometimes better than contention management (up to 2.5X). However, the FAA-based scheme involves complete revamping of the lock-free algorithm, which implies that other lock-free algorithms have to be rebuilt from scratch. Also, fairness is not handled in the proposed scheme.

DyFCoM implements its contention management as part of the RMW library, leaving the source code of the lock-free algorithm untouched. This eases the effort of the programmer, as just the simple act of replacing existing linearizing RMWs with managed RMWs solves the issues of throughput, as well as fairness.

Hendler et al. [Hendler et al., 2010] proposed ”flat combining”, which serializes synchronization regions to save on synchronization cost. Again, this work involved rewriting existing lock-free algorithms in order to employ combining.

Using MCS queues also alleviates contention, as shown in [Dice et al., 2013a]. Here, RMWs are serialized and thus is similar to the case where all RMWs complete successfully, like in [Morrison and Afek, 2013] (here CAS is replaced by FAA and thus does not have the notion of success).

Taubenfeld [Taubenfeld, 2013] proposed a scheme to implement a strong fairness guarantee wherein a thread which performs a successful linearizing operation waits until all other threads which failed their operations complete their CAS loops successfully. Taubenfield does not employ a helping mechanism, but relies on a benevolent scheduler to maintain lock-freedom. DyFCoM can be modified easily to implement such a scheme. However, without a helping mechanism, we will still rely on the scheduler.

5.7 Summary

In this section, we proposed DyFCoM, a dynamic fairness and contention management scheme which provides both increased fairness and throughput. We implemented DyFCoM as part of the RMW library, by augmenting RMWs with backoff and force-failing features. We monitor the success and failure of linearizing CAS operations and use it to dynamically vary the backoff value for throughput and force-fail CAS loops for fairness. Our experimental results show that DyFCoM provides better throughput (avg. 7.93%) when compared with the state-of-the-art contention management scheme when implementing contention management alone. After combining fairness management, DyFCoM shows an average performance improvement of 3.35% compared to the constant backoff scheme. DyFCoM also provides up to 43.64% better fairness. The main advantage of DyFCoM is that it is independent of the contention and con-

currency conditions. For future work, we plan to extend DyFCoM to provide stronger fairness guarantees by tweaking the fairness condition and the force-failing feature.

Chapter 6

Conclusion and Future Work

This chapter presents the summary of contribution of this thesis and discusses possible future directions. The main contribution of this thesis is the study and optimization of read-modify-writes (RMWs)when used in various applications such as synchronization, memory supervision, and profiling.

6.0.1 Summary of Contributions

As discussed earlier in the first chapter,the main factors influencing the cost of RMWs are the ordering semantics and contention-based failures. RMWs used for synchronization suffers from both these problems. We address the ordering semantics of RMWs for both blocking and non-blocking synchronization. Contention, however, is more prevalent in the non-blocking lock-free algorithms. Profiling applications also fall under the classification of lock-free algorithms and thus also suffer from increased contention. We attack both these application classes together. Memory supervision also suffers from poor performance due to its ordering requirements for correctness. We discuss this problem separately. In this thesis, we proposed – weaker RMWs for synchronization purposes, an efficient supervision architecture which ensures correctness, and a dynamic contention and fairness manager which provides better throughput and fairness for lock-free programs, including profiling applications. We first summarize these contributions in detail and discuss the effectiveness of each proposal.

6.0.1.1 Fast RMWs for Total-Store-Order Architectures

In our first contribution, we studied the behavior of RMWs in a weak consistency model, specifically Total-Store-Order (TSO), in order to ascertain the causes for the increased cost of RMWs. We observed that the atomicity semantics of an RMW is the key factor which affects the RMW's ordering semantics, its programmability, and its implementation cost. Existing TSO RMWs use a strict definition of atomicity (we call this type-1) which results in the RMW being strongly ordered like a memory barrier. This makes type-1 RMWs costly to implement and perform. They can, however, be easily used in synchronization idioms on TSO without requiring additional memory barriers. In particular, the C/C++11 concurrency model can be implemented easily by replacing the SC-atomic-reads and/or SC-atomic-writes with type-1 RMWs.

The ordering semantics required for these synchronization idioms, however, does not necessarily need a barrier-like semantics for an RMW. In light of this, we proposed two weaker atomicity definitions: type-2 and type-3 atomicity. We processed to formally derive how type-2 and type-3 RMWs would be ordered in a TSO architecture. In order to assure programmers of the portability of these RMWs in implementing synchronization primitives, we demonstrated that the resultant ordering of the weaker RMWs is sufficient to provide the required ordering semantics, including for the C/C++11 model.

In the weaker RMWs, we drain the write-buffer between the read and write of the RMW in order to remove this cost from the observed cost of the RMW. In doing so, however, there are scenarios wherein the execution may run into a deadlock, specifically due to writes which are in the write-buffer when the RMW is performed. In order to avoid these *write-deadlocks*, we propose a mechanism to track unique addresses accessed by RMWs and check the writes in the write-buffer against these addresses before performing the RMW. We use a bloom filter for this purpose, in order to reduce the implementation cost and the latency of operation of the mechanism. With this implementation, our experimental results show that type-2 RMWs (type-3 RMWs) are 58.9% (64.3%) cheaper than an existing type-1 RMW on average. This results in up to 9.0% improvement in the overall execution time of the benchmarks we experimented with.

Based on our analysis and experimental evidence, type-2 RMWs, while performing almost as well as type-3 RMWs, are also able to seamlessly replace existing type-1 RMWs in common synchronization idioms – except in situations where an RMW is used as a memory barrier. Thus, they appear to be a promising alternative to existing type-1 RMWs. We also show how the proposed type-2 and type-3 RMWs can be used to implement C/C++11 atomics – thus making it possible for the compiler to transparently utilize the proposed RMWs to realize C/C++11 more efficiently.

In order to utilize these weaker RMWs with contemporary architectures effectively, they have to be made available in hardware as separate instructions make use of *unused* opcodes. On the programming side, the burden can either be left to the programmer, or to the compiler. In cases other than when the programmer needs to utilize an RMW as a fence, the weaker RMWs can be used to increase performance. For a compiler, we have to perform a static analysis of the memory orderings enforced by RMWs, and use type-1, type-2, or type-3 RMWs appropriately. We consider this to be one of the future directions of the research work in this thesis.

6.0.1.2 Correct and Efficient Memory Supervision

In our second contribution, we addressed the usage of RMWs in supervised memory systems. Supervised memory systems use RMWs to access and update auxiliary metadata associated with each memory address. Existing supervised memory systems, implicitly or explicitly, assume sequential consistency. While working with weaker consistency models, memory supervision is subject to correctness issues, including imprecise exceptions and metadata-read reordering. Bobba et al. [Bobba et al., 2011] proposed two systems; TSOall which has significant performance overhead, TSOdata which is not general and still suffers from correctness issues for certain supervision schemes, in order to overcome the correctness issues. Bobba et al.'s work on safe supervised systems implicitly assumes that inorder execution of supervised memory instructions to ensure correctness. Our study led us to the conclusion that this ordering requirement is too strong for correct supervision, and that *metadata read ordering* is sufficient. To this end, we proposed SuperCoP, a supervision system which separates metadata reads and writes, and ensures metadata read ordering by performing the metadata reads in the critical path, and retiring metadata writes to the write-buffer.

Doing so, however, compromises on the existing metadata-data atomicity. We tackle this issue by designing 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. As such, there is no additional implementation cost except a change in the control of execution in the processor pipeline. The directory locking scheme has a similar overhead to the existing cache coherence schemes. Thus, at a relatively low implementation cost, SuperCoP achieves correct supervision with increased performance.

In summary, memory supervision has to be both correct and efficient. For supervision systems which modify the processor pipeline to perform supervision can achieve both efficiency and correctness through our approach. Decoupled supervision systems already address weaker consistency models but are wasteful in resource utilization as mentioned in chapter 2. We propose that generic supervised instructions be made available as part of the instruction set architecture (ISA), with the implementation as proposed in our work. The only requirement for our approach is that the read and write instructions of a supervised instruction be performed separately. The atomicity is preserved by our directory locking scheme.

6.0.1.3 Fairness and Contention Management for Lock-free Programs

Finally we tackle the issue contention in both non-blocking synchronization and profiling applications. Due to the increased dynamic frequency of RMWs in these applications, contention and more specifically RMW failures due to contention degrades the performance. Additionally, these programs do not have any inherent fairness guarantees. Existing proposals to address contention neglect fairness altogether.

In this work, we proposed DyFCoM, a dynamic fairness and contention management scheme which provides both increased fairness and throughput. We implemented DyFCoM as part of the RMW library, by augmenting RMWs with backoff and force-failing features. DyFCoM monitors the success and failure of linearizing CAS operations and uses this information to dynamically vary the backoff value for throughput and force-fail CAS loops for fairness. Our experimental results show that DyFCoM provides better throughput (avg. 7.9%) when compared with the state-of-the-art contention management scheme when implementing contention management alone. After combining fairness management, DyFCoM shows an average performance improvement of 3.4% compared to the constant backoff scheme. DyFCoM also provides up to 43.6% better fairness. The main advantage of DyFCoM is that it is independent of the contention and concurrency conditions. Another merit of DyFCoM is that it is implemented as an RMW library. Programmers only need to use these *managed* RMWs instead to regular RMWs in order to obtain increased performance and fairness.

6.0.2 Future Work

This thesis has addressed the applications of RMWs and the causes for the increased cost of RMWs individually. The scenarios we consider are, however, limited. We studied weaker RMWs and efficient memory supervision only with TSO architectures alone owing to their widespread usage. For lock-free programs, we only consider programs with a CAS loop. Although this covers a majority of the existing corpus of lock-free programs, it is not a general solution to contention and fairness management. Including these limitations, we describe possible future directions for research work with regard to RMWs, keeping in mind our observations and conclusions on the use of RMWs for various applications.

6.0.2.1 RMWs for Release Consistency

In this thesis we specifically tackled the ordering constraints of RMWs with respect to TSO architectures. While the definitions of atomicity that have been derived in this thesis is not specific to the consistency model, the orderings induced by atomicity varies with the underlying enforced memory orderings defined by the consistency

model. In case of consistency models weaker than TSO, like release consistency (RC), the induced atomicity orderings will drastically change. Also, generally in RC, the burden of the ordering of RMWs lies with the programmer, who has to insert the required fences before and/or after the RMW.

This burden can be lifted by providing generic RMWs which enforce all the required orderings for the sake of synchronization without sacrificing on the programmability or performance. This cannot be done in software alone, as issues such as write-deadlocks will still plague the implementation. Again, in order to implement generic weaker RMWs in RC will require a comprehensive study of the orderings enforced by RC and the atomicity orderings enforced due to this. With the increased usage of ARM-based processors which implement RC, it will be interesting to study the power budget of RMWs as well, along with the hardware required to implement weaker RMWs.

6.0.2.2 Contention Management of Weaker RMWs

Since the weaker RMWs proposed in this thesis require architectural modifications, we evaluated them with a full system simulator. We observed that such simulators perform badly under high contention and often underestimate the performance improvement that can be provided with software contention management. Thus we were not able to study the contention bottleneck of weaker RMWs. Theoretically, since weaker RMWs lock a cache-line for a longer duration (until the write-buffer is drained), it is possible that contention-based failures increase resulting in poor performance. It will be worth the effort to transfer the software contention management scheme into a hardware implementation and couple them with weaker RMWs.

In case of consistency models weaker than TSO, the memory ordering enforced by the RMW in the form of user-inserted memory fences can be added to the contention and fairness management code. With this, we obtain an RMW operation that is monitored for success or failure and also adds the necessary fences to achieve correct synchronization. Programmers can customize the options provided by a managed RMW. In such a case, however, issues such as write-deadlocks (if any) have to be addressed in hardware.

6.0.2.3 Stricter Fairness and Other Progress Guarantees

This thesis addresses only a weak definition of fairness which is represented by a measured parameter rather than a theoretical guarantee. However, a strict fairness guarantee, like the one enforced by Taubenfeld [Taubenfeld, 2013], requires a more complex helping mechanism even for fairness. With a benevolent scheduler and OS, such helping mechanisms can be done away with, but still mechanisms should be in place to detect violation of the existing non-blocking progress guarantee. Implementing various degrees of stricter fairness will be problem which finds use in real-time applications. Also, the implementation of fairness guarantees will change with the existing thread progress guarantee. Wait-free algorithms already make use of a helping mechanism in order to guarantee individual thread progress. It has to be studied whether the fairness guarantee supersedes the progress guarantee. Whichever be the case, it will be interesting to see if implementation of one makes the other redundant.

Bibliography

Advanced Micro Devices (2009). *AMD® Architecture Programmer's Manual*.

Adve, S. V. (1993). *Designing memory consistency models for shared-memory multiprocessors*. PhD thesis, Madison, WI, USA. UMI Order No. GAX94-07354.

Adve, S. V. and Gharachorloo, K. (1995). Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76.

Afek, Y., Kaplan, H., Korenfeld, B., Morrison, A., and Tarjan, R. E. (2012). Cb-tree: A practical concurrent self-adjusting search tree. In *Proceedings of the 26th International Conference on Distributed Computing*, DISC'12, pages 1–15, Berlin, Heidelberg. Springer-Verlag.

Agarwal, A., Bianchini, R., Chaiken, D., Johnson, K. L., Kranz, D., Kubiatowicz, J., Lim, B.-H., Mackenzie, K., and Yeung, D. (1995). 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. ACM.

Alglave, J. (2010). *A Shared Memory Poetics*. PhD thesis.

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

Anderson, T. E. (1990). The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16.

Attiya, H., Guerraoui, R., Hendler, D., and Kouznetsov, P. (2006). Synchronizing without locks is inherently expensive. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, PODC '06, pages 300–307, New York, NY, USA. ACM.

Attiya, H., Guerraoui, R., Hendler, D., Kuznetsov, P., Michael, M. M., and Vechev, M. (2011). Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 487–498, New York, NY, USA. ACM.

Bader, D. A. and Cong, G. (2005). A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps). *J. Parallel Distrib. Comput.*, 65(9):994–1006.

Batty, M., Memarian, K., Owens, S., Sarkar, S., and Sewell, P. (2012). Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *Proc. POPL*.

Batty, M., Owens, S., Sarkar, S., Sewell, P., and Weber, T. (2011). Mathematizing C++ concurrency. In *POPL*, pages 55–66.

Becker, P., editor (2011). *Programming Languages — C++*. ISO/IEC 14882:2011. A non-final recent version is available at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426.

Blundell, C., Martin, M. M. K., and Wenisch, T. F. (2009). Invisifence: performance-transparent memory ordering in conventional multiprocessors. In *ISCA*, pages 233–244.

Bobba, J., Goyal, N., Hill, M. D., Swift, M. M., and Wood, D. A. (2008). 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. IEEE Computer Society.

Bobba, J., Lupon, M., Hill, M. D., and Wood, D. A. (2011). Safe and efficient supervised memory systems. *High-Performance Computer Architecture, International Symposium on*, 0:369–380.

Boehm, H. (2011). The atomic_ops project. HP Labs http://www.hpl.hp.com/research/linux/atomic_ops/.

Boehm, H.-J. and Adve, S. V. (2008). Foundations of the c++ concurrency memory model. In *PLDI*, pages 68–78.

C (2011). *Programming Languages — C*. ISO/IEC 9899:2011. A non-final recent version is available at `http://www.open-std.org/jtc1/sc22/wg14/docs/n1539.pdf`.

Cederman, D., Chatterjee, B., Dang, N. N., Nikolakopoulos, Y., Papatriantafilou, M., and Tsigas, P. (2013a). A study of the behavior of synchronization methods in commonly used languages and systems. In *IPDPS*, pages 1309–1320.

Cederman, D., Chatterjee, B., and Tsigas, P. (2012). Understanding the performance of concurrent data structures on graphics processors. In *Euro-Par*, pages 883–894.

Cederman, D., Gidenstam, A., Ha, P. H., Sundell, H., Papatriantafilou, M., and Tsigas, P. (2013b). Lock-free concurrent data structures. *CoRR*, abs/1302.2757.

Chen, S., Falsafi, B., Gibbons, P. B., Kozuch, M., Mowry, T. C., Teodorescu, R., Ailamaki, A., Fix, L., Ganger, G. R., Lin, B., and Schlosser, S. W. (2006). 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. ACM.

Chung, J., Dalton, M., Kannan, H., and Kozyrakis, C. (2008). 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.

Culler, D. E., Gupta, A., and Singh, J. P. (1997). *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

David Levinthal (2009). *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors*.

Devietti, J., Lucia, B., Ceze, L., and Oskin, M. (2009). 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. ACM.

Dice, D., Hendler, D., and Mirsky, I. (2013a). Lightweight contention management for efficient compare-and-swap operations. In *Euro-Par*.

Dice, D., Lev, Y., and Moir, M. (2013b). Scalable statistics counters. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 43–52, New York, NY, USA. ACM.

Dice, D., Shalev, O., and Shavit, N. (2006). Transactional locking ii. In *In Proc. of the 20th Intl. Symp. on Distributed Computing*.

Gharachorloo, K. (1995). Memory consistency models for shared-memory multiprocessors. Technical report.

Gharachorloo, K., Adve, S., Gupta, A., Hennessy, J., and Hill, M. (1993). *Specifying system requirements for memory consistency models*. Computer Systems Laboratory, Stanford University.

Gharachorloo, K., Gupta, A., and Hennessy, J. L. (1991). Two techniques to enhance the performance of memory consistency models. In *ICPP (1)*, pages 355–364.

Gidenstam, A., Sundell, H., and Tsigas, P. (2010). Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. In *Proceedings of the 14th international conference on Principles of distributed systems*, OPODIS'10, pages 302–317, Berlin, Heidelberg. Springer-Verlag.

Gniady, C., Falsafi, B., and Vijaykumar, T. N. (1999). Is SC + ILP = RC? In *In Proceedings of the Twenty Sixth Annual International Symposium on Computer Architecture*, pages 162–171. IEEE Computer Society Press.

Graunke, G. and Thakkar, S. (1990). Synchronization algorithms for shared-memory multiprocessors. *Computer*, 23(6):60–69.

Ha, P. H., Papatriantafilou, M., and Tsigas, P. (2007). Efficient self-tuning spin-locks using competitive analysis. *J. Syst. Softw.*, 80(7):1077–1090.

Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C., and Olukotun, K. (2004). Transactional memory coherence and consistency. *SIGARCH Comput. Archit. News*, 32(2):102–.

Hendler, D., Incze, I., Shavit, N., and Tzafrir, M. (2010). Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, pages 355–364.

Hennessy, J. L. and Patterson, D. A. (2003). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition.

Herlihy, M. (1991). Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13:124–149.

Herlihy, M. and Shavit, N. (2008). *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Intel Corporation (2009). *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253669-033US.

Jain, R., Chiu, D.-M., and Hawe, W. (1998). A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *CoRR*, cs.NI/9809099.

Kannan, H. (2009). 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. ACM.

Ladan-Mozes, E., Lee, I.-T. A., and Vyukov, D. (2011). Location-based memory fences. In *SPAA*, pages 75–84.

LaMarca, A. (1994). A performance evaluation of lock-free synchronization protocols. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 130–140, New York, NY, USA. ACM.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput.*, 28(9):690–691.

Lev, Y. and Moir, M. (2011). Lightweight parallel accumulators using c++ templates. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, IWMSE '11, pages 33–40, New York, NY, USA. ACM.

Levon, J. and Elie, P. (2004). Oprofile: A system profiler for linux.

Lin, C., Nagarajan, V., and Gupta, R. (2010). Efficient sequential consistency using conditional fences. In *In Proceedings of the Nineteenth International Conference on Parallel Architectures and Compilation Techniques*.

Lin, C., Nagarajan, V., Gupta, R., and Rajaram, B. (2012). Efficient sequential consistency via conflict ordering. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 273–286, New York, NY, USA. ACM.

LLC, B. (2010). *Free Memory Management Software: Valgrind, Memcached, Mtrace, Leb128, Splint, Duma, Electric Fence, Memory Pool System, Mpatrol, Memwatch*. Books Nippan.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). 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. ACM.

Machine, I. B. and Staff, A. C. I. (1995). *PowerPC Microprocessor Common Hardware Reference Platform: A System Architecture*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Mellor-Crummey, J. M. and Scott, M. L. (1991). Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9:21–65.

Michael, M. and Scott, M. (1995a). Implementation of atomic primitives on distributed shared memory multiprocessors. In *High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on*, pages 222 –231.

Michael, M. and Scott, M. (1995b). Implementation of atomic primitives on distributed shared memory multiprocessors. In *High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on*, pages 222 –231.

Michael, M. M. (2002). High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 73–82, New York, NY, USA. ACM.

Michael, M. M. (2004). Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504.

Michael, M. M. and Scott, M. L. (1996). Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA. ACM.

Mirapuri, S., Woodacre, M., and Vasseghi, N. (1992). The mips r4000 processor. *IEEE Micro*, 12:10–22.

Misra, P. and Chaudhuri, M. (2012). Performance evaluation of concurrent lock-free data structures on gpus. In *ICPADS*, pages 53–60.

Moore, G. E. (1965). Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117.

Morrison, A. and Afek, Y. (2013). Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 103–112, New York, NY, USA. ACM.

Muralimanohar, N. and Balasubramonian, R. (2009). Cacti 6.0: A tool to understand large caches.

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

Narayanasamy, S., Pokam, G., and Calder, B. (2005). Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 284–295, Washington, DC, USA. IEEE Computer Society.

Natarajan, A. and Mittal, N. (2014). Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 317–328, New York, NY, USA. ACM.

Nethercote, N. and Seward, J. (2007a). 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. ACM.

Nethercote, N. and Seward, J. (2007b). 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. ACM.

Newsome, J. and Song, D. (2005). 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 2005.

Owens, S., Sarkar, S., and Sewell, P. (2009). A better x86 memory model: X86-tso. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 391–407, Berlin, Heidelberg. Springer-Verlag.

Qin, F., Wang, C., Li, Z., Kim, H.-s., Zhou, Y., and Wu, Y. (2006). 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. IEEE Computer Society.

Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. E. (1997). Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411.

Schaller, R. R. (1997). Moore's law: Past, present, and future. *IEEE Spectr.*, 34(6):52–59.

Seward, J. and Nethercote, N. (2005). Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 2–2, Berkeley, CA, USA. USENIX Association.

Shavit, N. and Touitou, D. (1995). Elimination trees and the construction of pools and stacks: Preliminary version. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 54–63, New York, NY, USA. ACM.

Singh, A., Narayanasamy, S., Marino, D., Millstein, T. D., and Musuvathi, M. (2012). End-to-end sequential consistency. In *ISCA*, pages 524–535.

Sorin, D. J., Hill, M. D., and Wood, D. A. (2011). *A Primer on Memory Consistency and Cache Coherence*. Morgan and ClayPool Publishers.

SPARC International, Inc., C. (1992). *The SPARC architecture manual (version 8)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

SPARC International, Inc., C. (1994). *The SPARC architecture manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Spear, M. F., Michael, M. M., and von Praun, C. (2008). Ringstm: scalable transactions with a single atomic instruction. In *Proceedings of the twentieth annual symposium*

on Parallelism in algorithms and architectures, SPAA '08, pages 275–284, New York, NY, USA. ACM.

Speziale, E., di Biagio, A., and Agosta, G. (2011). An optimized reduction design to minimize atomic operations in shared memory multiprocessors. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1300–1309.

Suh, G. E., Lee, J. W., Zhang, D., and Devadas, S. (2004). 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. ACM.

Taubenfeld, G. (2013). Fair synchronization. In Afek, Y., editor, *Distributed Computing*, volume 8205 of *Lecture Notes in Computer Science*, pages 179–193. Springer Berlin Heidelberg.

Terekhov, A. (2008). Brief tentative example x86 implementation for C/C++ memory model. `cpp-threads` mailing list, `http://www.decadent.org.uk/pipermail/cpp-threads/2008-December/001933.html`.

Thomadakis, M. E. (2011). The architecture of the nehalem processor and nehalem-ep smp platforms. *A Research Report of Texas A& M University, Tech. Rep.*

Treiber, R. K. (1986). Systems programming: Coping with parallelism. *Technical Report RJ 5118, IBM Almaden Research Center*.

Vallejo, E., Beivide, R., Cristal, A., Harris, T., Vallejo, F., Unsal, O., and Valero, M. (2010). Architectural support for fair reader-writer locking. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*.

Venkataramani, G., Doudalis, I., Solihin, Y., and Prvulovic, M. (2008). Flexitaint: A programmable accelerator for dynamic taint propagation. In *In 14th International Symposium on HighPerformance Computer Architecture (HPCA-14*.

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

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

Wall, D. W. (1991). Limits of instruction-level parallelism. *SIGOPS Oper. Syst. Rev.*, 25(Special Issue):176–188.

Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. (1995). 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. ACM.

Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24.

Zhou, P., Qin, F., Liu, W., Zhou, Y., and Torrellas, J. (2004). iwatcher: Simple, general architectural support for software debugging. *IEEE Micro*, 24:50–56.

Zhou, P., Teodorescu, R., and Zhou, Y. (2007). Hard: Hardware-assisted lockset-based race detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 121–132, Washington, DC, USA. IEEE Computer Society.

Zhu, W., Sreedhar, V. C., Hu, Z., and Gao, G. R. (2007). 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. ACM.