

TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK  
INSTITUT FÜR SYSTEMARCHITEKTUR  
LEHRSTUHL FÜR SYSTEMS ENGINEERING

## Diplomarbeit

zur Erlangung des akademischen Grades  
Diplom-Informatiker

# Hardwarebeschleunigung für Software Transactional Memory

Stephan Diestelhorst  
(Geboren am 12. September 1981 in Halle/Saale)

Betreuer: Dr. Michael Hohmuth  
Hochschullehrer: Prof. Dr. Christof Fetzer

Dresden, 30. Januar 2008



---

# Aufgabenstellung

Aufgabenstellung Diplomarbeit Stephan Diestelhorst:

*Hardwarebeschleunigung für Software Transactional Memory*

Zukünftige Mikroprozessoren werden nicht mehr eine exponentiell wachsende Single-Thread-Performance aufweisen, sondern stattdessen mehr und mehr Rechenkerne (Cores) besitzen. Anders als bisher kommt bestehende Software durch ein Hardware-Upgrade nicht mehr automatisch in den Genuss höherer Ausführungsgeschwindigkeit, sondern muss auf die höhere Anzahl verfügbarer Cores angepasst, also parallelisiert werden. Herkömmliche Parallelisierungstechniken werden mit steigender Anzahl Threads sehr komplex, weswegen neuartige Programmiermodelle gefragt sind.

Ein vielversprechendes solches Modell ist transaktionaler Speicher (Transactional Memory), das den Programmierer durch Bereitstellung von Transaktionen von aufwändiger Synchronisationsarbeit entlastet. Bisherige Software-Implementationen von transaktionalem Speicher (Software Transactional Memory, STM) verursachen allerdings noch zu große Verwaltungs- und Synchronisationskosten, sodass deren Einsatz beim heute und in naher Zukunft anzutreffenden Parallelitätsgrad noch nicht gerechtfertigt erscheint. Eine Möglichkeit, diese Kosten zu senken, ist eine Hardwarebeschleunigung des STM-Systems.

Ziel dieser Diplomarbeit ist es, bestehende Vorschläge zur STM-Hardwarebeschleunigung zu untersuchen und gegebenenfalls neue zu erarbeiten. Dazu soll ein Hypervisor so erweitert werden, dass er Hardwaremechanismen zur STM-Beschleunigung emuliert. Außerdem soll ein bestehendes STM-System auf diese Hardwareschnittstellen portiert und anhand von typischen Benchmarks evaluiert werden.



---

# Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Informatik eingereichte Diplomarbeit zum Thema:

*Hardwarebeschleunigung für Software Transactional Memory*

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den 30. Januar 2008

Stephan Diestelhorst



---

## **Kurzfassung**

Diese Diplomarbeit stellt eine neue Art von Hardwareprimitiven vor, die es Anwendungen erlaubt, atomare Programmblöcke zu verwenden, welche sich ohne Sperrvariablen (Locks) synchronisieren. Diese Hardwareprimitive werden unter dem Titel “Advanced Synchronisation Facility” (Fortschrittliche Synchronisationseinrichtung, kurz ASF) zusammengefasst und basieren auf einem Vorschlag von Advanced Micro Devices (AMD). ASF wurde ursprünglich entwickelt, um das Erstellen von gemeinsam benutzbaren Datenstrukturen ohne Locks zu vereinfachen. Neben der Betrachtung solcher Datenstrukturen wird in dieser Arbeit hauptsächlich untersucht, ob sich die ASF-Primitive auch zur Beschleunigung von Software Transactional Memory (transaktionaler Speicher realisiert in Software, kurz STM) nutzen lassen. Dazu werden mehrere Möglichkeiten entwickelt, wie sich STM unter Zuhilfenahme von ASF beschleunigen lässt und Implementierungen von einfach und doppelt verketteten Listen vorgestellt. Für die quantitative Bewertung wird eine Simulatorinfrastruktur erstellt, die eine potenzielle Hardwareimplementierung von ASF realistisch nachbildet. Die experimentelle Auswertung zeigt, dass lockfreie Datenstrukturen mit ASF nicht nur wesentlich einfacher entwickelt werden können, sondern dass sie auch einen Leistungsvorsprung von 66 % gegenüber den konventionellen lockfreien Lösungen haben. Die Beschleunigung von STM ist ebenfalls erfolgreich und steigert die Leistung der verbesserten Implementierung um bis zu 17 % in den durchgeführten Experimenten.

## **Abstract**

This dissertation introduces a new kind of hardware primitives which provide applications with atomic blocks not needing locks for synchronisation. These primitives are jointly called “Advanced Synchronization Facility” (ASF) and are based on an in-house proposal by Advanced Micro Devices (AMD). ASF was originally designed to simplify the creation of lock-free data structures. In addition to the analysis of such data structures, this paper will also examine if ASF can be used to speed up software transactional memory (STM). For that purpose various approaches for acceleration of STM and implementations of singly- and doubly-linked lists are presented. A simulator which models a realistic hardware implementation closely is used for the quantitative evaluation of these proposals. Results indi-

---

cate that ASF does not only simplify development of lock-free data structures, but these implementations also perform better by 66 % compared to traditioal lock-free ones. The acceleration of STM is successful as well and boosted performance of the accelerated implementation by up to 17 % relative to the original version.

---

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Motivation . . . . .	5
1.2	Goals . . . . .	6
1.3	Contributions . . . . .	6
1.4	Outline . . . . .	7
<b>2</b>	<b>Background and Related Work</b>	<b>9</b>
2.1	Simulation . . . . .	9
2.2	Transactional Memory . . . . .	10
2.3	Modern Microprocessor Architecture . . . . .	11
2.3.1	Structure . . . . .	11
2.3.2	Branch Prediction . . . . .	13
2.3.3	Multi-Core . . . . .	13
2.3.4	Caches . . . . .	13
2.3.5	Memory Consistency and Cache Coherency . . . . .	14
<b>3</b>	<b>Design</b>	<b>17</b>
3.1	Enhancing the Simulator . . . . .	17
3.1.1	Infrastructural Extensions . . . . .	17
3.1.2	Simplified Interconnect . . . . .	18
3.1.3	Comparison of Core Models and Native Hardware . . . . .	20
3.2	Advanced Synchronization Facility . . . . .	21
3.2.1	Overview . . . . .	21
3.2.2	Low-Level Interface . . . . .	22
3.2.3	Hardware Implementation . . . . .	23
3.2.4	High-Level Interface . . . . .	24
3.3	Application Design with ASF . . . . .	26
3.3.1	Lock-free Data structures . . . . .	26

---

3.3.2	Accelerating Software Transactional Memory . . . . .	28
3.3.2.1	Potential . . . . .	28
3.3.2.2	Approaches . . . . .	28
<b>4</b>	<b>Implementation</b>	<b>31</b>
4.1	Microarchitectural Fixes in PTLsim . . . . .	31
4.2	ASF in PTLsim . . . . .	33
4.2.1	ASF in Out-of-order Cores . . . . .	35
4.2.2	ASF with Multiple Cores . . . . .	37
4.3	Accelerating TinySTM . . . . .	40
<b>5</b>	<b>Experimental Evaluation</b>	<b>45</b>
5.1	Evaluation Setup . . . . .	45
5.2	Simulator Precision . . . . .	47
5.2.1	Motivation . . . . .	47
5.2.2	Single-Core Performance . . . . .	47
5.2.3	Multi-Core Performance . . . . .	48
5.2.4	Memory Latency . . . . .	52
5.2.5	Discussion . . . . .	55
5.3	Lock-free Data Structures . . . . .	56
5.4	Acceleration of STM . . . . .	57
<b>6</b>	<b>Conclusion</b>	<b>63</b>
6.1	Summary . . . . .	63
6.2	Experience and Review . . . . .	63
6.3	Outlook . . . . .	64
6.3.1	PTLsim . . . . .	64
6.3.2	ASF . . . . .	64
6.3.3	STM . . . . .	65
	<b>Bibliography</b>	<b>67</b>

## Acknowledgments

I would like to thank Advanced Micro Devices for supporting this thesis financially and for allowing me to work in their Operating Systems Research Center, join their research activities and get an inside view into some aspects of the black magic that can be found inside modern microprocessors.

Many thanks to all the cubicle dwellers at the OSRC in Dresden, especially to Sebastian Biemüller, Thomas Friebel and Uwe Dannowski for insights, social cubicle life and discussions. I am most grateful to my tutor Michael Hohmuth who not only was a very sociable cubicle neighbour but has had a great deal of patience with me and persuaded me more than once to get my hands dirty and try out new ideas. Dave Christie, a true veteran at AMD, has been very kind and helpful in explaining architectural details. His reassuring comments have given me confidence in my implementation of ASF.

Torvald Riegel from TU Dresden has actually sparked my interest in STM and has since been full of ideas and helpful comments.

I want to thank Matt Yourst for developing PTLsim and for providing support on the corresponding mailing list.

Marten Gajda and Stefan Vogelsang, in addition to the reviewers already named above, made many helpful comments on drafts of this document and kindly took over my dish washing duties for the last days.

Hearty thanks go to Antje, who has not only tolerated my night shifts working on this document, but has also reassured and supported me in every way possible.

My parents and family are the ones that own the largest part of my success. For their constant support I can not be sufficiently grateful.

Finally, I have to apologise for some omissions in this thesis. Despite several night shifts and extended support by friends, some of the planned things have not made it in time for the strict deadline. Most notable are several missing illustrations, which I dropped in order to get more time to improve and fix the written content. Despite this I fear that some content is not as polished as it could have been.

I hope that you will still find the thesis interesting to read and I would be happy if it can convey some of the enthusiasm I felt for the project in the last months.



# 1 Introduction

## 1.1 Motivation

Microprocessors have recently hit a barrier, often referred to as the “power wall”: Power and heat constraints limit the feasible clock-speed to today’s numbers (about 2.5 - 3.5 GHz). Together with limitations in exploitable instruction level parallelism, it is unlikely that single processor cores can maintain their exponential performance increase in the near future.

Transistor counts, on the other hand, continue to grow, thanks to progressing reduction in structural size (with processors in 45 nm technology available off the shelf as of 2008). Recent processors (AMD’s K8 & Barcelona, Intel’s Core 2) have introduced additional cores on a single die that make efficient use of the high number of available transistors.

For the reasons outlined, the number of available processor cores in commodity systems will rise in the short- and mid-term future. Applications that wish to benefit from improved processor technology cannot rely on increased single-thread performance anymore, but have to make use of increasing core-counts. Parallelising applications is thus becoming increasingly important.

Architectural support for the main problem of parallelisation—synchronisation—is only minimal on general purpose microprocessors. Although it has been proven that the provided primitives (atomic compare-and-swap) are universal, using them for complex synchronisation problems is cumbersome. Software transactional memory (STM) provides a simple primitive with good scalability, but suffers from overheads due to additional book-keeping and complexity.

Chip vendors are aware of the problem and evaluate different proposals for hardware extensions which tackle the problem of complicated and slow synchronisation, but to date none of these proposals have been integrated into available hardware. This lack of solutions in silicon is partly caused by uncertainty regarding the exact requirements for such primitives, because commodity software has relied on instruction-level parallelism (ILP) and clock-speed increase to deliver necessary performance improvements in the past. Another instance of the *chicken-and-egg problem*.

One idea looked at by Advanced Micro Devices (AMD) is the “Advanced Synchronisation Facility”

(ASF) which primarily eases lock-free synchronisation between multiple threads by providing atomic blocks directly in hardware.

In this thesis I want to analyse the utility of ASF for concurrent data structures and furthermore whether it can help reducing the overheads induced by STM.

## 1.2 Goals

The main questions of this thesis are: Can ASF reduce the overhead of software transactional memory? Can it simplify lock-free programming?

To answer these questions, a few prerequisites are necessary:

**Simulator:** ASF has not yet been implemented in silicon, doing so in this thesis would be impractical.

Therefore, to actually measure effects of ASF, another “vehicle” is needed: a simulator. My first goal is to find (and extend, if necessary) a simulator that supports the AMD64 architecture and instruction set. Additionally, support for multi-core execution is required.

**ASF:** Although ASF has been studied inside AMD for a while, it has not been discussed in public or implemented yet. During the implementation of ASF inside the simulator, various details of such an implementation have to be specified.

**STM:** Once ASF can be used and evaluated, a way to accelerate an STM with ASF has to be found. This means that an idea, implementation and evaluation are needed.

**Lock-free:** ASF should simplify the design and implementation of lock-free algorithms, in particular for lock-free concurrent data structures. To evaluate that claim, an implementation of a concurrent data structure with ASF is needed. The data structure should be simple, but still highlight ASF’s feature of making lock-free programming easier.

If these four goals have been achieved, an evaluation can then assess the utility of ASF. Furthermore, observations regarding ASF’s implementation, applicability and limitations provide valuable insight for a possible future integration into native hardware. Together with results for the acceleration of STM this will provide hints as to what hardware can do to accelerate software transactional memory and synchronisation in general.

## 1.3 Contributions

In this thesis I will make the following contributions:

- Outline, analysis and refinement of the “Advanced Synchronization Facility” (ASF), a proposal for hardware extension by AMD.
- Enhancement of an existing full-system simulator to support multiprocessor systems with cache coherency.
- Implementation of ASF in the enhanced simulator, yielding valuable insight into details of the interplay between ASF primitives and out-of-order execution in current microprocessors.
- Partial correctness proofs for aspects of ASF’s implementation.
- Implementation and evaluation of lock-free singly- and doubly-linked lists using ASF.
- Design, implementation and evaluation of an acceleration for an available high-performance STM.

## 1.4 Outline

The rest of this thesis is organised as follows: Chapter 2 will introduce related work and give a background on simulation and modern microprocessor cores. Chapter 3 will present the high-level design of new primitives for lock-free atomic blocks and my ideas regarding their simulation. It will also show how these primitives can be used to simplify and accelerate software transactional memory and lock-free data structures. The next chapter, Chapter 4 will delve into details of implementation aspects, look at subtleties inside the simulator and present details of accelerating transactional memory functions. The large amount of modifications and enhancements is then evaluated step by step in Chapter 5, with the final goal of predicting performance of the proposed enhancements on future hardware. Finally, I will conclude in Chapter 6.

## Remarks

Throughout this thesis I will use the first person singular pronoun “I” to indicate opinions, contributions and general remarks made by me, the author. The plural form “we” is used to include you, the reader, in statements that I make. For example, when I say that “we will see shortly that...” I hope that my following illustration will later allow you to actually “see that...”. I hope that this handling of the pronouns avoids ugly and repetitive passive constructions and includes you more into my chain of arguments.



---

## 2 Background and Related Work

This chapter will put the thesis into context. In Section 2.1, we will have a look at *simulation* and various simulators, tools that allow us to evaluate changes to a microprocessor without actually building it. *Transactional memory* has had a lot of attention from researchers, Section 2.2 presents a selection of those publications related to this thesis. Section 2.3 will have a closer look at *microprocessor cores* and quickly roundup their key ingredients. Finally, we will inspect *memory consistency* in Section 2.3.5, shedding light on global ordering of memory operations in multiprocessor environments.

### 2.1 Simulation

Simulation of microprocessors can happen at various levels. To evaluate applicability and performance of an instruction set extension, we can safely ignore simulators that operate below the gate level (at least for now). Simulators that simulate at register-transfer level are slow and need a lot of work to extend the model and are therefore impractical too. On the other hand, purely functional simulation, as done for example in QEMU [Bel05], does not allow comprehensive performance predictions.

Simulators operating at the pipeline level of a microprocessor core provide these details. These simulators can be divided into two groups: Those, in which the pipeline is modelled after a trace of execution has been recorded are called *trace-driven*. The ones that simulate timing and execution at the same time are called *execution driven*. The latter are useful, because the pipeline's timing model can influence the execution of instructions. Such an influence is important when execution relies on timing behaviour, for example in multiprocessor systems, where the outcome of races between two cores (for example trying to grab a global lock variable) can radically change taken paths of execution. This section will therefore concentrate on such execution-driven timing simulators.

Besides the internal tools employed by CPU vendors, various tools, such as M5 [BDH<sup>+</sup>06], GEMS [MSB<sup>+</sup>05] or TFSim [MHW02], model the microarchitecture of a modern out-of-order processor. Unfortunately, they are not directly applicable in this thesis' context, because the hardware extension, called "Advanced Synchronization Foundation" (ASF, Section 3.2) I want to examine enhances an AMD64 [AMD07c] based architecture and instruction set, but the simulators just mentioned simulate only the

SPARC architecture [SPA92] (M5 also features Alpha architecture). Additionally, GEMS and TFsim use Simics [MCE<sup>+</sup>02] as their functional reference core, which is not available under an open-source license.

This thesis uses PTLsim [You07] because, opposite to the other solutions, it is freely available and supports the AMD64 instruction set to a large extent. In addition, it supports *co-simulation*, transparent switching between simulation and native hardware to quickly execute uninteresting parts of the application under test. PTLsim also supports full-system simulation and features a rich CPU model, which provides detailed architectural statistics and can be tuned to match an AMD K8 core closely.

PTLsim consists of two major parts: one part is the actual model of the processor core which executes a given sequence of code, updates the model's architectural and internal state and keeps statistics for several key components (such as cache hits and misses). PTLsim contains two models, a simple in-order-model, useful for first steps and debugging, and a full out-of-order core simulator, which contains a detailed model of a modern multithreaded, out-of-order processor (Section 2.3) with caches and pipelines.

The other part of PTLsim provides the infrastructure, in particular the functionality of switching seamlessly between native execution and simulation. For full-system simulation, PTLsim uses Xen [BDF<sup>+</sup>03] to add a layer between the operating system and the hardware, which makes the transfer between simulation and native execution easier. PTLsim also relies on Xen to provide virtualised peripheral devices, this feature is not explored in this thesis, as all code that is simulated hardly does any I/O.

## 2.2 Transactional Memory

Herlihy proposed transactional memory (TM) in [HM93], implemented in hardware. The first *dynamic software* implementation of TM was also later proposed by Herlihy *et al.* [HLMS03].

Recently, a large number of software implementations (STM) have been developed. Several contributions to key aspects can be classified: Harris and Fraser made STM available for unmanaged / word-based programming languages with their *word-based* STM [HF03]. Both, Herlihy's and Harris & Fraser's, implementations operate non-blocking, *i.e.*, without taking any locks, thus avoiding starvation and priority-inversion.

As additional indirections, decreased locality and book-keeping required for non-blocking STMs cause a large overhead, Ennals has suggested to drop the requirement of non-blocking operation [Enn06] and introduced a *locking* STM with better performance. Since then, most high-performance STM's employ locks to protect the data, eliminating the need for added indirections. Dice *et al.* for example use locking in their STM [DSS06] and report about twice the throughput of Harris/Fraser's implementation.

Recently, Riegel *et al.* have combined their idea of tracking validity of object versions by using ranges of time [RFF06, RFF07] with a lock-based infrastructure. They have created TinySTM, a *time-based* locking STM which is among the fastest STMs, according to Felber *et al.* [FFR08].

But despite these steady improvements, STMs are still about an order of magnitude slower than native hardware in single-thread performance. Even though there is a significant amount of revised proposals for hardware transactional memory (HTM) [AAK<sup>+</sup>05, MBM<sup>+</sup>06, RHL05], none of this functionality has been introduced in commercially available microprocessors to date, supposedly because of the profound architectural changes caused by these extensions.

Less elaborate hardware support to just support STMs has been suggested earlier, to keep architectural extensions modest proposals primarily restrain (1) either the size (HyTM [DFL<sup>+</sup>06, KCJ<sup>+</sup>06], PhTM [LMN07]) of supported hardware transactions, or (2) limit the offered expressiveness (LogTM-SE [YBM<sup>+</sup>07], SigTM [MTC<sup>+</sup>07]) or both (HASTM [SATJ06]).

Each of these hardware approaches is accompanied by software that works around the limitations and provides the interface and features of STM: flexibility, expressiveness and large transaction sizes at reduced overhead.

ASF, in contrast, has a broader scope than only the acceleration of TM and can be implemented with very moderate hardware extensions. The result is a mechanism that has comparatively (to those listed in (1)) small capacity and richer expressiveness (than those in (2)) but requires a more static setup than both, hardware proposals in (1) and (2), and STMs.

## 2.3 Modern Microprocessor Architecture

Modern microprocessors are complex machinery. Hennesy and Patterson [HP02] give a good overview about the matter and provide many details. In this section, I want to roughly sketch modern processor cores, to provide a basic understanding for the microarchitectural details this thesis touches.

### 2.3.1 Structure

The general structure of a modern microprocessor core is that of a pipeline: Instructions are processed at various stages and eventually leave the core. At the heart of a modern *out-of-order* core operates the *Reorder Buffer* (ROB), which (as the name suggests) reorders the microoperations ( $\mu$ ops) delivered from the fetch and decode units. The ROB schedules them on the functional units (FUs) and keeps track of all  $\mu$ ops in flight in the core in general.

Usually some form of Tomasulo's algorithm [Tom67] is used to track the dependencies between dependent  $\mu$ ops. Reordering helps when some instructions have a large execution time, such as floating point arithmetic and loads from main memory. The ROB then can still execute other independent instructions on available FUs, regardless of programme order which improves utilisation of the core's resources. Higher utilisation increases instruction throughput, measured in instructions-per-cycle (IPC), by exploiting available instruction level parallelism (ILP). This reordering differentiates out-of-order cores from the classical in-order cores, in which  $\mu$ ops progress through the pipeline stages in strict order.

Other stages in the processor core's pipeline are:

**Fetch & Decode:** Fetches the instruction stream from an instruction cache and decodes the instructions from the external representation (for example x86-opcodes) into an internal representation, which is usually simpler, so called microoperations ( $\mu$ ops). Complex x86-opcodes are usually split into multiple simple  $\mu$ ops: `addq %rax, (%rdx)` might be translated into: `ld %tmp, (%rdx); add %rax, %tmp; st %tmp, (%rdx)`, where `ld` and `st` are the load and store primitives. The main goal of this stage is to separate complex instructions into memory accesses, arithmetic instructions and branches. Operands are evaluated and checked for dependency.

**Issue:** This stage is controlled by the  $\mu$ op scheduler and comprises the distribution of the ready  $\mu$ ops (those where all operands are available) to available FUs of the requested type (arithmetic, load-store, branch, floating point).

**Execute:** The actual processing of the  $\mu$ op takes place in this stage. For example, loading the value from memory, computing the arithmetic operation.

**Bypass, Forward & Write-back:** In these stages the results of the execute stage are distributed to waiting dependent  $\mu$ ops and the internal register file is updated. The term "bypassing" is used to stress the fact that results do not have to be written back into the register-file first to make them available to dependent  $\mu$ ops. The result can rather be used as soon as it is computed by the FU.

**Retire / Commit:** This stage puts all out-of-order results back in order, removes the  $\mu$ ops from the core, frees their entry in the ROB and resolves any speculation. Results of multiple operations are merged to create a consistent architectural (externally visible) state.

The input and output stages (Fetch, Decode and Commit) process instructions in programme-order (*in-order*), all other stages process  $\mu$ ops as soon as the necessary resources (FUs and operands) are available (*out-of-order execution*).

### 2.3.2 Branch Prediction

A large performance boost and additional complexity is introduced by branch prediction: Conditional branches can alter the instruction flow, but usually results affecting the outcome of the conditional branch (taken / not taken) are available late. Stalling the input stages until the branch is resolved lets many of the core's resources lie idle. *Branch prediction* predicts the outcome of conditional branches early, so that the fetch unit keeps the core filled by fetching from the predicted location. These predictions cannot be perfect and hence the processor sometimes executes the wrong stream of instructions. When a conditional branch is finally resolved (it is known for sure, whether it is taken or not) and detected as mispredicted, the core must discard all wrongly executed instructions by flushing the pipeline. This is possible as all instructions and their results remain speculative (that is, subject to false prediction) until they reach the commit stage. The commit stage operates in-order and hence any instruction that is committed has to be on the right path of execution, as all preceding branches have been committed as well and their ambiguity has been resolved.

### 2.3.3 Multi-Core

Many recent processors consist of multiple independent cores; consistently increasing transistor counts (Moore's Law [Moo98]) do not provide much speed-up to the single-core anymore, because ILP is exhausted and power and heat constraints limit the switching speed of the single transistor. The increased number of transistors is then used by "just" multiplying entire cores on a single die. Such processors with multiple cores are often referred to as multi-core chips or *chip multiprocessors* (CMPs).

### 2.3.4 Caches

Main memory is very slow in comparison to the core's speed, latency ranges usually in the magnitude of 100-200 processor cycles, meaning that a load which requests data from main memory has to wait that many cycles, before the data is usable by dependent instructions. To reduce this delay, processors feature several levels of content-addressable memory, so called *caches*, which store a frequently used subset of main memory for faster access. Caches "closer" to the core get lower numerical levels, are faster and smaller than those further away from the core. For example, a first-level cache (L1) of size 64 kB with latency of three cycles could be combined with an L2 cache with 512 kB capacity and 15 cycles latency.

Accessing main memory is further complicated by the translation between virtual addresses used by applications, and physical addresses actually specifying the cell in the memory module. This translation is called *paging* and implemented by page-tables. As the process of translation is slow, the most often

used mappings between virtual and physical addresses are stored in a fast buffer, called the translation lookaside buffer (TLB).

Multi-core chips often connect several caches and cores in a sophisticated fashion, cores usually have some dedicated lower-level caches, the large outer-level caches are often shared among cores.

Caches have many more properties besides size and latency, a thorough treatment is beyond the scope of this section, [HP02] provides a comprehensive treatment of all things discussed so far, including caches.

### 2.3.5 Memory Consistency and Cache Coherency

Current x86 microprocessors use a variety of buffers, caches and out-of-order execution units. For systems with a single processor it suffices to maintain a locally consistent view of the execution order, which is actually an illusion, given the previously introduced principle of out-of-order execution. External visibility can then be perturbed freely<sup>1</sup> without compromising correctness.

In environments with multiple CPUs, cores or hardware-threads, the external ordering of events is important to ensure correctness of multithreaded applications. Memory *consistency models* are guarantees about the externally visible order of memory accesses relative to a given programme order of memory load and store instructions. They constrain the external visibility of the reordering that occurs inside the cores.

Various such models exist, but in general, it is safe to assume that a less restrictive memory consistency model allows more aggressive optimisations inside the core and translate to higher performance [AG96].

The most intuitive consistency model is *sequential consistency* (SC) [Lam79], which enforces ordering among memory instructions for each processor to be identical to programme order, but allows memory accesses of different processors to interleave freely.

Despite being rather intuitive, SC is rarely found in current high performance microprocessors because restrictive consistency models limit various performance optimisations of out-of-order microprocessors. For example, SC enforces all loads to wait after a store which causes a miss in the L1 cache, eventually stalling the entire core.

For that reason, recent x86 / AMD64-compatible processors do not support SC, but a different class of memory consistency, sometimes called *processor consistency*. This term, however, is not used consistently [GAG<sup>+</sup>92, Goo89]. The most important relaxed requirement compared to SC is that loads can be scheduled ahead preceding<sup>2</sup> writes.

---

<sup>1</sup>with the exception of special regions of memory that access I/O devices

<sup>2</sup>in programme order

In older documentation by Intel and AMD, loads from different locations were not ordered externally, but that behaviour has been dropped in recent publications [AMD07a, Int07], making the resulting consistency model equivalent to *total store order* (TSO) as defined by SPARC in [SPA92].

Both, Intel and AMD, have implemented ways to relax or strengthen the consistency model for certain ranges of memory, but that is not considered here. Additionally, several primitives exist which enforce ordering among a certain class of memory accesses on a single processor. These primitives are called *fences* and are also known as barriers. Store fences ensure that stores preceding the fence instruction have taken effect before the ones following the fence. Load fences work like-wise for loads, full (or m-)fences act as both, store and load fence.

Additionally, atomic read-modify-write (RMW) instructions are defined which ensure that no concurrent access to the memory location in question can be made between the read and the write part of the composed instruction. One frequently used example for these instructions is compare-and-swap, an instruction that swaps a memory location with a register, given that the memory location contained some expected value. Atomicity ensures that no other core can modify the memory location between the successful compare and the swap.

Besides memory consistency that constrains external visibility of out-of-order execution of memory accesses on a single block of memory, multiprocessor systems face an additional problem: Copies of a single memory location may be located in different memory buffers across the entire system simultaneously. If the system guarantees *cache coherence*, it guarantees that a single logically valid value can be determined consistently in the system, despite the existence of multiple copies of the data with potentially different values. For that different approaches exist, however, in current x86 hardware some form of the well known MESI protocol [PP84] is usually employed.



## 3 Design

This chapter presents the key ideas behind the work in this thesis. Section 3.1 will have a look at enhancements of PTLsim, the full-system simulator that forms basis of the entire project. Section 3.2 will outline the new instruction set extension called ASF, which will help with building simple, fast lock-free synchronisation primitives. Its use for lock-free data structures and STMs is explored in Section 3.3.

### 3.1 Enhancing the Simulator

#### 3.1.1 Infrastructural Extensions

As shown in Section 2.1, Yourst demonstrates in [You07] that PTLsim is capable of simulating a single K8 core with about 5% of error for several key measures (number of cycles, total instructions, cache misses and mispredicted branches). This precision was measured in a complex full-system benchmark containing multiple threads (rsync, ssh). However, Yourst makes no predictions for simulating multi-core machines and admits that precise multi-core simulation would require more work. In addition, closer examination of the simulators' core reveals also missing components essential for single-core performance.

PTLsim does support simulation of multiple CPUs, however, this is achieved by only multiplexing the front- and back-end structures (instruction fetch and update of architectural state) of the simulated core, just as real hardware does this with the *simultaneous multithreading* (SMT) approach, implemented for example in Intel's Pentium 4 CPUs[KM03]. The major drawback of this technique is that many of the internal functional units and the caches are not split between the concurrently executing threads. Threads that compete for these resources execute slower than they would if they could run on the entire core each by themselves[BP04].

In order to get comparable results to a native SMP system, PTLsim needs to be modified to support multiple independent cores, with a more sophisticated cache and memory topology. However, modelling a full interconnect network is a complex task, for example the larger part of GEMS (Section 2.1) consists

of such a network model. In the scope of this thesis I have decided to simplify the interconnect and memory hierarchy.

With these modifications, PTLsim now has multiple independent cores, each with a private cache hierarchy. Each of the cores is attached to the same physical memory, with uniform access times from all cores and immediate visibility of memory updates. As with the original model, all data is held in physical memory, caches merely decide about the latency of when the data arrives.

### 3.1.2 Simplified Interconnect

As outlined above, the simulator uses caches only to store tags of the data and keeps the actual values in main memory. Hence, a coherency protocol is not necessary inside the simulator to ensure correctness. However, the cache coherency model, together with the general memory consistency model, can have a huge impact on overall system's performance [AG96]. There is a large variety of different designs and implementations for cache coherency. In my simplified design I have concentrated on the most immediate effects of cache coherence.

Under the following assumptions:

- Unlimited interconnect bandwidth
- Negligible latency of coherency messages (probes) on the interconnect
- Infinite store-buffer size in each core

most of the single-core structure can remain unaltered and no explicit modelling of the interconnect is necessary. Despite these restrictive simplifications, the model can still capture two essential effects of cache coherency protocols:

**Invalidation:** Updates of a given cache line cause invalidation of this line in all other cores' caches, resulting in increased delays if those cores try to access the given line again later. This is important because it models the fact that upon modification only one core in the system can exclusively own the modified cache line in its cache.

**Reduced cache-to-cache latency:** If core B has modified a cache line recently (and thus has the only valid copy of), implementations may allow cache-to-cache forwarding of lines when core A accesses this line. Because there is no need to write data to memory first, the costs of cache line write-back are avoided, resulting in a different cache miss penalty for cache misses that hit in other caches.

These effects can be incorporated into PTLsim's existing multithreaded and the proposed multi-core simulator model as follows:

**Loads:** Loads perform a cache-lookup in the local cache hierarchy, on a hit they proceed as before, on a local miss, the cache hierarchy of the other simulated cores is probed (emulating a coherency probe). If that remote probe hits the latency of the load is set to the cache-to-cache transfer latency, otherwise to the latency of main-memory, as in the original model. Checking the remote presence is subject to the assumptions above, mainly the unlimited bandwidth and negligible latency assumptions, hence probing the other caches is done immediately when processing the load. Enqueueing of probing bus-messages into a modelled network is not necessary. Averaged delays introduced by the network can however just be made part of the global cache-to-cache latency.

**Writes:** In the original core-model, writes update the data just as the corresponding  $\mu\text{op}$  reaches the (in-order) retire / commit stage 2.3. In case of a cache miss, retrieval of the missing cache line happens after retirement. In native hardware, this is often loaded off to a subsequent *store-buffer* [BJ00]. With multiple cores present, the writing core also has to acquire *ownership* of the line to be modified first, by sending invalidating probes and waiting for acknowledgements (ACKs) from the other cores in the system. This invalidation protocol also happens *after* the store has been retired from the core.

Although several authors (for example [WAFM07]) analyse the performance impact of the limited size of the store buffer and propose various techniques to reduce store-misses, I have assumed a store buffer of unlimited size for simplicity. Together with the small probe latency (see above), this allows writes to invalidate the modified cache line immediately in all other cache hierarchies present in the system when it reaches the commit stage.

With this approximation, this simplified MOESI protocol obviously has limitations, in particular, when the assumptions made above are violated. If for example the interconnect bandwidth is the bottleneck in the system, the model will not capture the reduced throughput. Such a situation may occur in workloads with many cache line transfers, for example due to high contention on a frequently updated variable. Frequent stores may lead to a full store-buffer, because the stores have to wait until the line is present in the local core in exclusive state, which may take longer and occur more often in a multiprocessor system than with single-core store-misses. Once the store-buffer is filled, no stores can retire from the out-of-order core. Retirement proceeds in-order, so eventually the entire core has to wait for the store-buffer.

The assumption of zero latency for coherency probes does not only decrease the waiting time of retiring stores, it also changes timing between two cores, as cache line invalidation happens immediately. This can cause skew among several threads, which influences results more prominently if threads synchronise often, especially with RMW instructions.

Feature	Original Model	Improved Model	AMD Barcelona
Core Model	Single-core, SMT	SMP	Multi-core, ccNUMA
Interconnect	none	simplified	Hyper-Transport
Latency		zero	topology dependant
Bandwidth		infinite	limited
L1 D-Cache	physically indexed	virtually indexed	virtually indexed
L3 Cache	single	private for each core	shared in same package
Hierarchy	inclusive	inclusive	exclusive, adaptive L3
TLB	L1	L1 & L2	L1 & L2
HW Prefetcher	none	preliminary support	adaptive prefetchers
Atomic RMW	internal locks	internal locks	two-phase, optimistic
Memory Consistency	Immediate visibility	Immediate with cache coherency	processor consistency with MOESI cache coherency

Table 3.1: Differences between PTLsim’s original core, the improved model proposed in this thesis and an AMD Barcelona based system.

### 3.1.3 Comparison of Core Models and Native Hardware

Despite the improvements described above, the new simulator model is still different from real hardware. One issue has been addressed already: the simplified interconnect model (see previous section). Table 3.1 summarises the differences.

One prominent disparity is the different treatment of atomic read-modify-write (RMW) instructions: On modern AMD64 hardware, an optimistic attempt is made by monitoring the modified data for incoming probes, retrying if these probes are detected. In order to guarantee success, eventually some form of locking protocol is employed, at least there are speculative and non-speculative phases [AMD07b]. The treatment of atomic RMW instructions in the new model has been carried over from the original “smtcore”<sup>1</sup>. It uses locks inside the simulator that shield the RMW instruction from any read / write instruction to the same word from another core. These locks are an internal resource of the simulator and are not modelled temporally, specifically, they can be acquired, released and handed from one core to another *instantly*. In Section 5.2.4 we shall see how this different treatment affects performance.

In addition to the changes in the interconnect network discussed above, the simulator and native hardware

<sup>1</sup>Needed fixing of an intricate bug that made the instruction non-atomic during rare interplay of out-of-order pipeline mechanisms in the simulator.

also differ in their design of the cache hierarchy. AMD's Barcelona core contains three levels of cache, with levels L1 and L2 being exclusive, meaning that a line is never present in both caches[AMD07c]. PTLsim models an inclusive cache hierarchy, where lines in a cache level have to be present in all higher levels (*e.g.* a line in L1 has to be present in L2 and L3). The L3 cache in Barcelona cores is shared between the four cores on a single chip and has some sharing-aware policy, meaning that it is not exclusive for all cache lines. PTLsim does not share the L3 cache between cores, but rather has a fully separated complete cache hierarchy for each core. Due to time constraints, I have decided to leave the inclusion properties for my improved model as they are in PTLsim's original core, but have adapted the sizes and associativity to match those of real Barcelona cores. Another minor modification added to the new model is the indexing of the L1 data cache, with the L1D cache now being virtually indexed like in AMD's core, whereas PTLsim used physical indexing originally.

Barcelona cores use a second level of larger buffers to reduce the amount of misses in the translation lookaside buffer (TLB). This feature was not present in PTLsim's original core, hence I have added it to the improved core. Memory performance and the effect of the improvements are evaluated later in Section 5.2.4.

Finally, I have experimented with prefetching to improve the performance of the simulated core and implemented a predicting hardware prefetcher similar to a combination of the ones described in [BC95] and [Jou90], inspired by what is known about Barcelona's prefetchers [AMD07c]. But as the inner details of Barcelona's hardware prefetcher are unknown and the used benchmarks did not profit from prefetching due to highly randomised access patterns and small working-set sizes, they are not evaluated in this thesis.

## 3.2 Advanced Synchronization Facility

### 3.2.1 Overview

AMD's Advanced Synchronisation Facility (ASF) has been proposed as a lightweight solution to make flexible atomic read-modify-write (RMW) constructs available to user- and kernel-space programmes. These constructs are realised as speculative critical sections (ASF-CS) without locks. Such a section starts with the *specification phase*, which specifies the memory locations the sequence should operate on—the *speculative set*. In the following *atomic phase*, these locations then can be modified with plain AMD64 instructions<sup>2</sup>. ASF ensures that the modifications made during an ongoing atomic phase are

---

<sup>2</sup>Modifications to memory locations not in the speculative set are legal, but are not subject to ASF's consistency checks and undo-operation.

not observed by other cores before this atomic phase commits the results, ensuring that the entire critical section actually appears to have executed atomically. More precisely, ASF checks no element of the speculative set is updated by another core and that elements updated speculatively in the atomic phase (or specified with `LOCK PREFETCHW`) are not observed (read) during the critical section. Once ASF detects an inconsistent observation / update (inconsistent probe) by another core, it aborts the running atomic phase and undoes all modifications to the locations from the speculative set (other memory locations can be modified as well in the atomic phase, but these modifications are not undone). The application is then free to retry immediately or employ contention control, such as random exponential back-off or more elaborate contention control mechanisms (for example from [IS05]).

ASF makes lock-free programming easier, because more convenient primitives, such as atomic double compare-and-swap (DCAS) or load-linked (LL) / store-conditional (SC) [DMMJ01], can be constructed easily. The next section will show such an implementation.

### 3.2.2 Low-Level Interface

The AMD64 instruction set is extended by four new instructions that control ASF's operation:

**LOCK MOV / LOCK PREFETCHW:** The lock-prefix extends the original version of these load<sup>3</sup> instructions to add the supplied address to the speculative set. The first such instruction begins the specification phase.

**ACQUIRE:** This instruction ends the specification phase (no new locations can be added to the speculative set) and attempts to begin the atomic phase. `ACQUIRE` has an input operand that contains the number of locations added with `LOCK MOV` and `LOCK PREFETCHW`, which will be discussed shortly. Before commencing the atomic phase, `ACQUIRE` checks the location count and ensures that the values read in the specification phase are still valid. If these conditions are satisfied, `ACQUIRE` starts the atomic phase and returns with an output operand of zero (and set zero-flag), otherwise `ACQUIRE` returns a non-zero exit code (and clears the zero-flag). The application should then check for this exit code and handle it, usually by simply jumping back to the first `LOCK MOV` instruction, restarting the specification phase.

**COMMIT:** Once this instruction has completed, speculative modifications of the specified locations are made permanent, the speculative set is cleared and the atomic phase is ended.

**VALIDATE:** This instruction can be used during the specification phase to determine validity of the previously specified locations before the `ACQUIRE`. Like the `ACQUIRE` instruction, `VALIDATE`

---

<sup>3</sup>`LOCK MOV` between registers or writing to memory is an undefined instruction.

expects the number of speculative locations in an input operand and returns zero when validation is successful and non-zero otherwise. In contrast to the ACQUIRE instruction, VALIDATE does not end the specification phase and also allows later LOCK MOVes to extend the speculative set. This feature of possible later extension makes VALIDATE crucial for the acceleration of STM.

In order to make ASF available to kernel- and multiple user-space applications, the entire ASF state is discarded on far control-transfers (such as interrupts, exceptions and system-calls), in particular on page-faults and context switches. To detect these events during the specification phase, ACQUIRE and VALIDATE expect a location count, which specifies the number of locations (*not* cache lines) the application expects in the speculative set. Because the set is cleared on far-control transfers, discrepancies between the size of the set and the specified location count hint that there have been far control transfers and the integrity of the critical section cannot be established.

During the atomic phase ASF continuously monitors the speculative set and aborts the atomic phase if necessary. It discards modifications made to entries in the speculative set and restores control at the ACQUIRE and executes it again, making it fail. User code will then restart the critical section or take other action as seen fit. To allow this roll-back, ACQUIRE snapshots the instruction and stack pointer when it successfully enters the atomic phase and restores them on abort. Besides these two registers and the values of the specified locations, no other state is saved and restored by ASF. Far-control transfer in the atomic sections also causes an abort before the far control-transfer request is actually acted upon.

Figure 3.1 shows an example of a simple double compare-and-swap function implemented using ASF.

### 3.2.3 Hardware Implementation

The given interface of ASF can be implemented in various ways; this thesis bases on the idea that speculative modifications are written through and original content is backed up before entering the atomic phase in a locked-line buffer (LLB). Holding the original content in an extra-buffer makes it possible to evict entries from the caches and hardly changes the operation of stores. Additionally, COMMIT is fast, it just clears the LLB, the more seldom aborts have to pay the penalty of the LLB writing back the original contents.

The LLB not only provides backup storage, but it also checks incoming coherence probes whether they access data that is used speculatively. For that to work, the LLB has to receive all incoming coherency probes. If these are usually filtered by outer caches, forwarding to the LLB has to be ensured. Because the LLB receives all coherency probes, no other core can see speculative updates that have been written through. In Section 4.2 I will examine the mechanism more closely and prove that it works correctly.

```

# double compare-and-swap
# checks RAX = (RSI) and RBX = (RDI)
# true: swap (RSI) <-> RDX, (RDI) <-> RCX
# false: do nothing
# clobber: R8, R9, RBP
1:
    lock mov $(%rsi), %r8
    lock mov $(%rdi), %r9
    acquire $2, %rbp
    jnz 1b                #retry
    # start atomic phase
    cmp %rax, %r8
    jne 2f
    cmp %rbx, %r9
    jne 2f
    # swap both values
    mov %rdx, (%rsi)
    mov %r8, %rdx
    mov %rcx, (%rdi)
    mov %r9, %rcx
2: # end atomic phase
    commit

```

Figure 3.1: Double compare-and-swap (DCAS) built with ASF (in AT&T syntax).

ASF operates on aligned blocks of memory, the current implementation uses cache lines with a size of 64 bytes.

Another implementation of ASF would be to simply use the speculative execution in the out-of-order core to keep instructions inside the atomic phase speculative, by not retiring them from the ROB until the COMMIT has been processed. Such an approach is limited by the size of the ROB (currently between 50-100  $\mu$ ops), which makes it unusable for longer atomic phases.

### 3.2.4 High-Level Interface

A higher level interface for application-specified atomic sections is desirable. Several authors analyse possible language extensions to support transactional memory [DMSS07, CLL<sup>+</sup>07, HF03], in general ASF might be interfaced in a similar way. Moreover, if the compiler can determine statically that a transaction fits within the size limitations and can also be decomposed into a specification phase and following atomic phase automatically, ASF can simply be used for the entire transaction. Tools that do such kind of analysis exist already, specifically Tanger [FFM<sup>+</sup>07], as implemented by Felber *et al.* uses LLVM [LA04] to do link-time optimisation, analysis and integration of calls to STM library functions.

But as this thesis concentrates on the mechanism, implementation and performance impact of ASF, language integration of ASF is well beyond the scope of this work. I hence propose a very simple

low-level, library-like interface to ASF developed for the C programming language, making the newly introduced instructions directly visible to the programmer. Such an approach is well in line with many other researches who separate means used for and the language integration of atomic sections.

The resulting interface then looks like this:

**lock\_loadT(T\*):** Loads a value of type T from the specified address and marks the value as speculative for the following ASF atomic phase.

**lock\_prefetchw(void\*):** Adds the given address to the speculative set, without loading the value into a variable or register. Addresses specified with `lock_prefetchw` are treated by ASF as if they have been speculatively modified, even if they have not, causing aborts of the critical section on read probes to these cache lines.

**acquire(fail, state, #loc):** Starts the atomic critical section, returns error-codes in “fail” and awaits the expected size count (number of addresses, not cache lines) of the speculative set as second argument. Acquire needs a word of backup-storage (to store the frame-pointer register), the local variable in the “state” parameter provides this space. This is an implementation artifact, necessary because the used C compiler (GCC) does not honour declarations of frame pointer clobbering.

**commit():** Ends the critical section, commits the speculative values.

**validate(fail, #loc):** Like the native primitive, `validate` checks whether the specified set is still consistent during the specification phase. The expected size of the speculative set is provided as parameter and `validate` returns zero in “fail” on success and non-zero otherwise.

**touchN(..), acquire-fail-clobber(state):** These two helper functions are tweaks to support the tentative integration of ASF. The former makes sure that data specified in `lock_loads` is actually used and the calls to the `lock_loads` are not optimised away by the compiler. Otherwise the count for the `acquire` might be incorrect. The latter ensures that during an abort in the critical section all used registers are not assumed to contain a certain value and restores internal state using the “state” word, which was filled by `acquire` before.

ASF introduces some fundamentally different concepts, such as memory modifications being undone and unexpected control-flow redirection without full register preservation, which the compiler does not know about. Therefore such a macro / library based interface has serious drawbacks, as it has to rely on certain undocumented or compiler specific behaviour and restricts effectiveness and applicability of compiler optimisations, such as inlining, common sub-expression elimination and loop-unrolling.

Proper compiler support for ASF primitives is therefore desirable, but until that is in place, ASF can still be used by (inline) assembly, the proposed library interface and perhaps even more helpful in several

```

void transfer(long *acc1, long *acc2, long transfer) {
    long tmp1, tmp2, asf_fail;
    volatile long    asf_state;
retry:
    tmp1 = lock_load64(acc1);
    tmp2 = lock_load64(acc2);

    acquire(asf_fail, asf_state, 2);
    if unlikely (asf_fail) {
        asf_fail_clobber(asf_state);
        goto retry;
    }

    /* We're in the atomic phase now! */
    *acc1 = tmp1 + transfer;
    *acc2 = tmp2 - transfer;
    release();
}

```

Figure 3.2: Simple atomic transfer with ASF using a simple macro/function call interface.

pre-built larger building blocks, such as libraries for lock-free data structures and common operations, for example double compare-and-swap, etc. The proposed library interface has a small performance overhead because the frame pointer is backed up to and restored from the stack.

Example code implementing atomic bank transfers without locks using the given interface just specified can be found in Figure 3.2.

## 3.3 Application Design with ASF

### 3.3.1 Lock-free Data structures

Synchronising accesses to shared data structures through one big lock for the entire instance of the data structure has obvious performance drawbacks, hence researches have been searching for alternatives for quite a while. One such alternative are lock-free data structures which avoid the use of locks altogether. Herlihy has shown in 1990 [Her90] that in general every concurrent data structure can be implemented without locks using only the atomic compare-and-swap instruction, which is available on today's hardware. Despite this general finding, only few lock-free implementations for data structures have been proposed so far, suggesting that Herlihy's construction is impractical and / or has large overheads.

Valois has proposed the first lock-free implementation of a linked list [Val95] in 1995, which according to Harris contained errors and was too complex [Har01]. Harris then proposed a simpler implementation for the linked list in 2001. This fairly long delay between the initial proposal and the correct and usable

```

int set_remove(set_t *set, int val) {
    ...
retry:
    /* Traverse the list to the element,
       without any locks / ASF protection. */
    find(set, val, &prev, &next);
    ...
    contained = 0;
    prev_next = lock_loadpt(&prev->next);
    next_next = lock_loadpt(&next->next);
    next_val = lock_load32(&next->val);

    touch3(prev_next, next_next, next_val);
    acquire(fail, asf_store, 3); /* atomic start */
    if unlikely (fail) {
        acquire_fail_clobber(asf_store);
        goto retry;
    }
    /* check for chaining errors */
    if unlikely (prev_next != next) {
        commit();
        goto retry;
    }
    if (next_val == val) {
        contained = 1;
        prev->next = next_next;
        next->next = (node_t*)NULL;
    }
    commit(); /* atomic end */
    ...
    return contained;
}

```

Figure 3.3: Lock-free linked list removal with ASF.

implementation suggests that finding correct and efficient lock-free data structures is a non-trivial task.

For the straight-forward linked-list, the problem is the delete operation which needs to swap a pointer (the “next”-field of the preceding element) and lock another (the “next”-pointer of the element to be deleted) simultaneously, which cannot be done with any single instruction present in current x86 or AMD64 instruction sets.

ASF can be used easily to simplify such a lock-free list implementation, swapping the pointer and maintaining an unchanged next-pointer at the same time, by specifying both locations in the specification phase, just as shown in Figure 3.3.

Note that the ASF implementation can also traverse the list without any protection, under the assumptions of type stable memory which is also made in the lock-free implementation. Type stable memory [GC96] ensures that objects of different type are allocated from different memory pools. For example if an element in the linked list is deleted and the memory for the element is freed, type stable memory

guarantees that this memory is only used for other elements of this type of linked lists. Usually, memory is not type stable because freed memory may be used for objects of different types, in the example perhaps for nodes of a red-black-tree.

As an almost trivial extension, a doubly-linked list can be constructed in a similar fashion, as all necessary additional pointer modifications in the add and remove operations can be performed atomically, too.

In order to show the simplicity and evaluate the performance I have implemented both, a singly- and a doubly-linked list using ASF, performance results can be found in Section 5.3.

### 3.3.2 Accelerating Software Transactional Memory

#### 3.3.2.1 Potential

Software transactional memory has shown good scalability properties, but suffers from various overheads (see Section 2.2). In non-blocking STMs (such as [HLMS03, HF03]) substantial overhead (more than 100% [DSS06]) is incurred because of additional levels of indirection being introduced. The additional levels are a direct result of the lack of flexible atomic constructs on current CPUs, where atomic RMW instructions to a *single* memory cell, such as compare-and-swap, are the only available primitives. These indirections break down the atomic modifications of multiple words into pointer replacement. For these STMs, ASF could remove this level of indirection, by providing atomic updates of multiple memory location.

For example in DSTM [HLMS03], the additional indirection added by the “start” reference in the transactional object “TMOject” can be omitted. The “start” reference points to a “Locator” which can be obsoleted by embedding its fields (transaction status, pointer to old and new version) directly into the “TMOject”. Furthermore, if objects themselves are smaller than cache lines, they may be embedded directly inside the “TMOject”, thereby removing yet another level of indirection.

But many of today’s STMs are blocking, simply using locks and thus avoiding the need for atomic changes to multiple items in the STMs meta-data, which in turn makes additional indirections unnecessary. TinySTM [FFR08] aims to be a simple implementation of a lock-based STM and according to Felber *et al.* it is also among the fastest STMs, but it is still much (by a factor of 5-7) slower than the single-threaded code or implementations that serialise concurrent accesses with a single big lock (see Section 5.2), at least for the provided benchmarks of random concurrent operations on a linked-list and red-black-tree.

### 3.3.2.2 Approaches

But despite these overheads, there is not much “low-hanging fruit” to grab; there are, for example, no indirections to remove (see above). Due to ASF’s static specification phase, it cannot be easily used as a drop-in replacement for small transactions: ASF supports speculative modifications only after the specification phase is finished, making it impossible to add more locations to the speculative set. But there is still an overhead which can be alleviated: TinySTM reads the value of the lock associated to the requested datum twice / three times during the transactional load operation, making sure that no concurrent thread has acquired the lock in the meantime. ASF can remove the additional loads by converting the transactional load into a small ASF sequence and monitoring the lock variable by adding it to the read-set, a similar approach works with the transactional store primitive.

In addition to this optimisation of limited scope, the idea of using ASF directly for tracking application data is appealing, as it should remove (at least for some transactions) considerable amounts of overhead, especially due to reduced number of locations for the STM’s validate operation. An issue is, however, the limited capacity of ASF (currently 8 cache lines) and flexibility, as the order of specification phase and atomic phase is static.

ASF is still useful for tracking (parts of) the read-set of the transaction by using the specification phase of ASF. Each `stm_load` is then basically converted to a LOCK MOV (ASF). Two issues in design arise from such an algorithm. First, ASF did not provide an explicit VALIDATE instruction originally. Validation of data accessed in the specification phase happened only at the ACQUIRE. Till then, the application might have operated on stale state<sup>4</sup>. Second, the size-limitation of ASF has to be sidestepped. Various approaches for similar situations exist (see Section 2.2), but they either require very elaborate changes to the entire run-time system [LMN07] or do not face the same problems of expressiveness and flexibility [KCJ<sup>+</sup>06, DFL<sup>+</sup>06]. Those with limited expressiveness often have a much larger number of traceable locations, *i.e.*, less size restrictions and are still more flexible than ASF.

With the VALIDATE instruction being part ASF, the remaining issues are the size-limitation and the low flexibility. The following approaches would work with these prerequisites:

**Try-and-Fail:** Try all transactions in ASF first, if they exceed the size and / or expressiveness limitations, abort and retry in software.

This may cause many aborts, especially, when the expected size of transactions is larger than the maximum size of ASF. Additionally, meta-data is still be necessary to track whether written lines

---

<sup>4</sup>After a successful ACQUIRE, validation happens immediately, as incoming inconsistent probes will cause an instant abort and roll-back of the current atomic phase.

have been specified already and to buffer writes.

**Swap-on-Overflow:** Track the read-set with ASF, as soon as the maximum capacity is hit, transfer all locations from ASF to the STM.

This avoids additional aborts and can inline accesses to the STM's meta-data into a larger loop (at the transfer point), thereby giving more room for optimisations. Detection of multiple accesses to the same cache line is possible at little cost, because some meta-data has to be kept for the later transfer into the STM. This potentially increases utilisation of ASF's tracking facilities (8·64 bytes traceable vs. 8 memory reads).

**Switch-on-Overflow:** Similar to “Swap-on-Overflow”, but make all new loads that exceed ASF's capacity use the STM.

Avoiding the later transfer of lines from ASF into the STM saves time and also allows us to drop the meta-data kept for that purpose. A simple check for aliasing is still possible by comparing each new speculative load's address to the last one. If they are in the same cache line, the new load does not have to be added to ASF.

Both approaches “Swap-” and “Switch-on-Overflow” require proper interplay between ASF and TinySTM. With the write-through variant of TinySTM, modifications made by `stm_stores` to locations monitored by ASF will trigger an error at the next `VALIDATE`. As ASF state is at the latest validated during the transaction's commit, this validation would fail and trigger an abort of the transaction.

If the `stm_store` operation happens before the accelerated `stm_load`, the latter would detect that the lock for the modified location is already taken by the `stm_store` and abort the reading transaction. This works, because TinySTM locks variables as soon they are modified with `stm_stores` and the accelerated `stm_load` checks the lock variable for the requested location. Section 4.3 contains implementation details.

In Section 5.4, I will compare the very simple scheme and the “Switch-on-Overflow” schemes. I have omitted the “Try-and-Fail” approach, because I do not think that the amount of wasted work due to size-aborts is justified. In particular, if the available number of entries is small and flexibility is severely limited additionally.

An initial implementation of “Swap-on-Overflow” did not produce satisfactory results, likely because of the larger overhead for additional meta-data and the transfer from ASF to STM control. I have therefore postponed a detailed analysis.

## 4 Implementation

This chapter highlights some interesting problems that emerged during the implementation of the proposed design from the previous Chapter 3 in more detail. Section 4.1 will take a closer look at the microarchitectural model in PTLsim, show some problems in the model and discuss possible solutions. In the following section (Section 4.2) we examine how ASF can be added to PTLsim. Intricacies of out-of-order cores are highlighted and the interplay between multiple ASF critical sections running simultaneously is analysed. The final section (Section 4.3) discusses the already sketched acceleration of TinySTM in more detail.

### 4.1 Microarchitectural Fixes in PTLsim

A good match in simulated and real performance for benchmarks from the problem domain is essential for meaningful results derived from the simulator. One major element in this chain of similarities is the initial precision of the original core model present in PTLsim. Yourst claims that the original model can be tuned easily to match several key measures of an AMD K8 processor [You07], in fact, the modifications to PTLsim’s parameters is part of PTLsim’s distribution.

Given the large similarity between AMD’s Barcelona (Family 10h) and K8 microarchitecture for a single core (compare instruction latencies and microarchitecture in [AMD05] and [AMD07c]), the amount of work required to get single-core performance similar between Barcelona and PTLsim seemed small.

To my surprise, the “intset” (see Section 5.1) benchmark from the TinySTM distribution<sup>1</sup> produced a large performance offset in comparison between native (Barcelona and even K8) cores and the supposedly similar PTLsim model. The native hardware would outrun<sup>2</sup> the simulated core, producing about 40% more throughput. Most notably, this difference was measured with a sequential version of the benchmark, without any involvement from STMs, atomic RMW instructions or interconnect networks.

After some dissection, it turned out that the code-snippet in Figure 4.1 was mainly responsible for the

---

<sup>1</sup>Available from <http://www.tinystm.org>.

<sup>2</sup>Of course, the simulation itself took much longer than that. Comparison is between real-time (native execution) and simulated-time (execution in simulator).

difference.

```
prev = set->head;
next = prev->next;
while (next->val < val) {
    prev = next;
    next = prev->next;
}
```

Figure 4.1: Problematic code sequence from intset benchmark.

This section of code is used by all benchmarked operations (add, remove, contains) to locate a specific element in the linked list. By my version of GCC 4, this loop was translated to the following assembler sequence<sup>3</sup>:

```
...
1:
mov    0x8(%rax),%rax    # load prev->next
mov    0xc(%rsp),%edx   # load val (parameter on stack)
cmp    (%rax),%edx     # compare next->val and val
jg     1b
...
```

Figure 4.2: Compiled result of the loop in Figure 4.1.

denoted here in AT&T syntax and AMD64 instruction set.

The loop consists of three loads from memory, an arithmetic operation and a conditional jump. The minimal latency for each load is three cycles, if the data is present in the L1 data-cache. The throughput of the loop is limited by the same latency, as the `mov 0x8(%rax),%rax` instruction is dependent on its own output. Amazingly, the native K8 core manages to execute one iteration of that loop (close to) every three cycles, hitting the minimal delay as noted above. This is possible, because the core aggressively unrolls the loop multiple times (thanks to correct branch prediction) and interleaves the instructions well on the available functional units. Of course, all accessed data (and code) has to be present in the L1 cache and the L1-TLBs must be filled as well. With sufficiently small working-set sizes, this is the case after running through the linked-list once.

PTLsim's tuned architecture is similar to the K8 / Barcelona pipeline, however, it produced substantially larger durations for the loop. Fortunately, it is possible to take a very close look at the pipeline of the simulated core with PTLsim, allowing detailed analysis of stalls and debugging the pipeline at every stage and microoperation. Scanning through the logs, I eventually found two flaws in PTLsim's pipeline implementation: Firstly, bypassing the register file did rarely work as expected, introducing an additional

<sup>3</sup>Which is obviously not compiled optimally, despite optimisation flags. Perhaps a bug in GCC.

cycle of delay between the producer of a value (for example the `mov 0xc(%rsp), %edx` instruction above) and the consumer of that value (`cmp (%rax), %edx` above). Secondly,  $\mu\text{ops}$ <sup>4</sup> were not scheduled optimally on the core's functional units, resulting in additional stalls due to unavailable functional units.

In order to increase the simulator's precision, I fixed the broken bypass engine and modified the scheduler to group the  $\mu\text{ops}$  better on the available clusters of functional units. These clusters group FUs, inside a cluster the FUs can broadcast results without additional delays, the delay for inter-cluster result forwarding is configurable. PTLsim uses these clusters to model delay between floating point units and integer units. The proposed model of a Barcelona core contains three integer clusters and one floating point cluster; the integer clusters can transfer results without additional propagation cost, transfers between integer clusters and the floating point cluster incur a penalty. Once an  $\mu\text{op}$  has been dispatched to such a cluster, it cannot be moved to another.

The original scheduler would try to put dependent instructions into the same cluster of functional units, thereby often exceeding the clusters maximum utilisation and leaving other clusters almost idle. My attempt was to trade off the cost it would take to propagate a result between clusters and the penalty for waiting in an overcrowded cluster. Luckily, the latency between the integer / memory clusters is zero, as specified in the original PTLsim description of the K8 core, making it possible for the scheduler<sup>5</sup> to spread  $\mu\text{ops}$  as much as possible among the clusters.

These two fixes bring PTLsim close to the native performance of the single-core benchmarks, as can be seen in Section 5.2.

## 4.2 ASF in PTLsim

Up to this point the simulator has emerged into a multi-core design modelling independent cores and a simplified cache coherency protocol with an idealised interconnect. As I want to analyse the applicability of an instruction set extension, it also has to be added, of course.

Given the proposed functionality and interface of ASF, different ways of implementing the extension in hardware can be envisioned. For my enhanced simulator core, I stick as close as possible to the original proposal outlined in section 3.2. This makes the simulator and possible future hardware more alike and has some benefits:

---

<sup>4</sup>Microoperations, see Section 2.3.

<sup>5</sup>This is a special case, the scheduler is generic and designed to work well with different latencies. Admittedly, more testing is required to validate its performance in more general settings.

**Applicability:** Given similar simulator and native hardware designs makes the application of modifications, tweaks and bug-fixes to the hardware easier.

**Feasibility:** Extending existing complex architectures is fairly difficult. An implementation proposed by a chip-vendor has at least passed some of the feasibility tests employed and is hence more likely to be implemented in silicon eventually.

**Plausibility:** Ensuring similar implementation in simulator and hardware is the only way to increase the significance of the results obtained with the simulator, as no real hardware to compare to is available.

Extensions to an existing CPU core can roughly be categorised as follows:

**Interface:** How is the extension presented to software using it? This is usually specified in the proposal, however, representation of instructions as binary opcodes on x86 can be intricate.

**Infrastructure:** Does the core need additional functional units, buses or other entities for the extension to function correctly?

**Behaviour / Interaction:** How does the extension affect the processing of other instructions and pipeline design? This is especially important if execution of instructions<sup>2.3</sup> in the core occurs out of order.

**State:** Does the architecturally-visible state have to be augmented?

For the thesis' scope of extending the enhanced out-of-order-execution core of PTLsim with the 'Advanced Synchronization Facility' extension, the following changes have to be made:

**Interface:** Extension of PTLsim's decoder to detect and translate ASF opcodes into the internal representation.

**Infrastructure:** With the similarity argument given above, we need a locked-line buffer and additional storage for two registers (RIP, RSP).

**Behaviour / Interaction:** ASF essentially adds another level of speculative execution to the already present speculative out-of-order execution of modern microprocessors. The speculative state of ASF becomes actual, as soon as the ASF critical section commits, *i.e.*, if the COMMIT instruction retires successfully. For this to happen, the instruction must have left the speculative execution of the out-of-order core (*e.g.* all preceding conditional branches have been resolved), additionally, ASF semantics require monitoring for coherency probes that hit in the LLB and would lead to an abort of the atomic phase. The COMMIT instruction hence has to check for such probes at the retire stage, before it can clear the LLB and conclude the atomic phase.

As the speculation introduced by ASF is backed up by different means (the LLB and provided

backup storage for the instruction- and stack pointer registers) than the standard speculative execution inside an out-of-order core, the original mechanisms remain mostly unchanged, *e.g.* the ROB (see Section 2.3). Care has to be taken however when these two mechanisms interact. As outlined above, speculative ASF state can only be resolved after control-flow speculation is resolved. Inversely, speculative ASF state has to be rolled back as soon as the instructions causing its creation (new ASF instructions) turn out to have been mispredicted by the underlying speculation mechanism (*e.g.* wrong branch prediction leading to spurious LOCK MOVes and ACQUIREs).

In order to keep the initial ASF implementation simple, I have added additional ordering constraints between ASF instructions, essentially they ensure that two ASF critical sections do not overlap and that cache lines must have arrived in the core, before the ASF critical section can commence. The detailed rules and reasoning will be discussed shortly, ways to drop some of them are explored in Section 6.3.

**State** The architecturally visible state does not have to be modified for ASF.

### 4.2.1 ASF in Out-of-order Cores

Section 2.3 introduced the principle of out-of-order execution, which reorders instructions from the instruction stream in order to make better use of functional units. Reordering relies on proper ordering between dependent instructions to ensure correctness of the shuffled sequence. In PTLsim these dependencies are tracked by the producer-consumer relationship, created by an operation which needs a value that is produced by another. If there is no such relationship, the ROB can freely permute the sequence of execution of these operations. Together with the fact that multiple operations are executed in the core simultaneously, either in different functional units or in other stages of the pipeline, the core of modern microprocessors is a complex and fragile machinery.

Simply adding the ASF primitives into the core is therefore doomed to fail: As the ASF instructions have hardly any register dependency, but in fact modify the entire context of execution, they would be executed in an entirely unconstrained order. This could, with prediction of conditional jumps, cause ASF instructions to be executed various loop iterations in advance, not protecting any memory references any longer.

It is henceforth mandatory to specify and enforce certain ordering constraints among ASF instructions and between them and other relevant instructions. The typical way of adding such additional constraints is by creating “artificial” producer-consumer dependencies and then relying on the present dependency detection mechanisms in the ROB. Another possibility would be to stall the pipeline’s front-end for each ASF instruction and let the pipeline drain. This ensures that only the ASF instruction is present inside the

core eventually. Stalling the pipeline that way has a large overhead (at least the number of pipeline-stages in processor cycles).

A general ASF critical section is a sequence of instructions that looks like this:

$$LLD, LLD, \dots, ST_1, ACQ, ST_2, COM, ST_3, \dots, LLD_2 \quad (4.1)$$

Where  $LLD$  are LOCK MOVes,  $ST_i$  are normal stores,  $ACQ$  is the ACQUIRE and  $COM$  the COMMIT primitive. The ordering constraints governing the execution of ASF instructions are for the current implementation:

$$ready_{addr}(LLD) \rightarrow issue(ACQ) \quad (4.2)$$

$$ready_{data}(LLD) \rightarrow issue(ACQ) \quad (4.3)$$

$$commit(ST_1) \rightarrow issue(ACQ) \quad \text{if } addr(ST_1) \in LLB \quad (4.4)$$

$$issue(ACQ) \rightarrow commit(ST_2) \quad \text{if } addr(ST_2) \in LLB \quad (4.5)$$

$$commit(ST_2) \rightarrow commit(COM) \quad \text{if } addr(ST_2) \in LLB \quad (4.6)$$

$$issue(ACQ) \rightarrow commit(COM) \quad (4.7)$$

$$commit(COM) \rightarrow issue(LD_1) \quad \text{if } addr(LD_1) \in LLB \quad (4.8)$$

$$commit(COM) \rightarrow issue(ST_3) \quad \text{if } addr(ST_3) \in LLB \quad (4.9)$$

$$commit(COM) \rightarrow issue(LLD_2) \quad (4.10)$$

Where  $\rightarrow$  is the “happens-before” relation defined by Lamport in [Lam86], such that  $A \rightarrow B$  means that event  $A$  happens entirely before  $B$ , and  $commit(X)$ ,  $issue(Y)$  are the events of instructions  $X$  and  $Y$  being at the respective pipeline stages.  $ready_{addr}(X)$  and  $ready_{data}(X)$  are the events when the data and address of a memory operation are available, whereas  $addr(X)$  simply returns the address of the memory instruction  $X$ . The  $LLB$  is the set of all memory locations specified by the ongoing ASF critical section.

Rules (4.2) and (4.3) ensure that all addresses and all data is available for the snapshot created by the ACQUIRE instruction. (4.4) makes sure that modifications of lines in the speculative set before the ACQUIRE are carried out before the snapshot is taken, so that the modification is present in the snapshot. Stores to the speculative set have to be executed in the atomic phase, they are constrained by (4.5) and (4.6). Start and end of atomic phases is ordered through (4.7). (4.8) and (4.9) make sure that memory operations following the atomic-phase cannot read speculative results early. The last rule (4.10) finally ensures that ASF critical sections cannot overlap. This dependency is crucial for my implementation, as it simplifies the design of the LLB by allowing only one ongoing ASF-CS at a time. Extending ASF’s implementation to support multiple simultaneous ongoing ASF-CS would add complexity, but it would

also speed-up workloads with many short critical sections in close proximity. In Section 5.4 we will evaluate the performance penalty.

Rules (4.5), (4.6) and (4.7) are automatically enforced by the core ( $issue(X) \rightarrow commit(X)$ ), pipeline stage progression, and  $X, Y$  implies  $commit(X) \rightarrow commit(Y)$ , in-order commit stage). (4.8) and (4.9) are automatically enforced too: If the atomic phase is aborted when  $LD_1$  or  $ST_3$  have issued already, the pipeline would be flushed and the issued instructions would be discarded. If no abort occurs, the data  $LD_1$  and  $ST_3$  have operated on was actually valid.

The other dependencies are tracked by scanning the ROB during the issue of an ACQUIRE and building dependencies according to rules (4.2),(4.3) and (4.4). Dependency (4.10) is enforced by checking at every issue of a LOCK MOV that the core is not in “ASF-speculative” mode.

## 4.2.2 ASF with Multiple Cores

ASF aids multiprocessor synchronisation, simulation should hence also model a multiprocessor system. In Section 3.1 I described where PTLsim has deficiencies in this regard and proposed extensions to capture some effects of real hardware with multiple cores.

With multiple cores, data can be accessed simultaneously by normal loads and stores and those embedded in an ASF atomic phase. ASF has to maintain consistency in such cases by aborting atomic phases and reverting modifications made speculatively if necessary. In the present implementation of ASF this occurs by writing back the content of the LLB.

A store operation that has triggered such an abort will have to wait until the abort and roll-back of all ASF atomic phases containing the element in their read-set has completed. With MOESI implemented on HyperTransport [CH07] this can be achieved easily: Each core that has to abort the atomic phase will delay the acknowledgement (ACK) to the incoming invalidation probe until the LLB has successfully been written back. With the previously introduced simplified coherency model (see Section 3.1.2) this is not possible in PTLsim, as the model assumes that stores are not stalled in the main pipeline.

Loads suffer a similar problem: Once they try to read data that has been speculatively modified inside an ASF critical section but not committed, the respective critical section has to abort and the load must read the original, unmodified data.

Viewing the problem from the other side, we find that ASF’s original implementation proposal relies on probe monitoring to detect accesses to memory locations that are in use by the atomic phase, but again, neither the original simulated core of PTLsim, nor my proposed simplified coherent processor core model these probes explicitly. Hence, during execution of loads and stores on one core, not only the

caches of the other cores have to be checked (for invalidation and deciding on cache-to-cache latency), but also their LLBs have to be examined for any ongoing atomic phase and those atomic phases with conflicting speculative sets must then be aborted.

Checking for these inconsistent memory accesses immediately has the same issues as for caches, namely the reduced delay between the cause, the inconsistent concurrent memory access, and the effect, immediate abort of the ongoing critical section. This will not lead to increased abort rates due to small latency between inconsistent accesses, aborts and restarts, because cache line invalidation causes cache misses and associated (cache-to-cache) latencies during the specification phase of the restarting ASF critical section. These delay the restarting critical section, reducing the probability of live-lock.

Let me illustrate the problems with a small example, where a write on core A hits a line modified in an ongoing critical section on core B: As core A's store does not wait for any ACKs by core B, B cannot delay this probe acknowledgement until after it has finished writing-back all modified cache lines from the LLB.

In order to get the unmodified cache line from core B, the simulator directly extracts it from B's LLB and merges its content with the written data in core A. As A's write operation then produces the only logically valid copy of this cache line in the system, it implicitly discards any modifications made by B's critical section of that line. Care has to be taken to not accidentally undo A's write by the write-back of the original cache line from B's LLB during the abort of the critical section! This means that lines whose original content has already been forwarded from an LLB (causing an abort of the atomic phase on that core) to other cores must not be written back during the abort.

Although this behaviour is clearly an artifact of the shortcomings of the simplified coherency and interconnect model, this approach has some benefits for real hardware, too: Forwarding directly from the LLB detaches core A's memory operation from core B's abort process, henceforth reducing the delays for succeeding memory operations and thus removing pressure on the abort algorithm.

With cache line forwarding in such a way and potentially many copies of the same cache lines in various LLBs, the question arises which LLB should forward the data. I will prove shortly that this does not matter. Of course, once forwarding has taken place, all LLBs containing a copy of that line have to ensure that this line is not written-back during the inevitable abort. In context of the example above: If another core C's LLB had a copy of the cache line requested by core A, C would abort its ongoing critical section as well. If it then wrote back that requested line, it would overwrite A's modifications.

Therefore A has to probe all cores and their LLBs, and although it just needs the data forwarded from one of the cores (B or C in the example), all of them have to exclude this line from being written back

during abort.

We have just seen that both, loads and stores, can receive the original content directly from an LLB, without waiting for the roll-back operation to complete.

Once multiple LLBs contain the requested line, forwarding brings up the question which LLB to ask. I will prove that this decision does not matter, as all LLBs will contain the same data for a shared line. Furthermore, we can conclude that there exists still only single logically valid version of the data, even with ASF critical sections,.

**Lemma 4.1.** *If a cache line containing some specific memory location is present in multiple LLBs, all these entries contain the same value for said location.*

*Proof.* By induction on the number  $n$  of cache lines containing that element.

$n = 1$ : As there is just a single line, there is just a single version of the data.

$n = m + 1$ : From the induction hypothesis we know that  $m$  elements contain identical versions. Let us inspect the moment when some other LLB A adds a copy of that cache line. There are three cases to consider, relative to the time when the last entry in the  $m$  entries was added:

*Case 1* - The line has not been modified.

LLB A contains another identical copy, hence there are  $n$  identical copies in the system.

*Case 2* - The line has been modified non-speculatively.

If the line has been modified, there must be some store that happened after the last of the  $m$  elements was added and that has taken effect already. ASF guarantees that before this store can take effect in the system, *all* LLBs containing that cache line are cleared and their content is written back. In native hardware this can be realised by delaying the ACKs to the invalidating probes generated by the write operation until each LLB has finished this roll-back operation. The simulator discards the corresponding line immediately from the  $m$  LLBs, using one of the entries (they are identical) directly for store-merging.

This means that the old  $m$  copies are actually invalid or non-existent ( $m = 0$ ) and hence the new copy is the only one present.

*Case 3* - The line has been modified by a speculative write.

When the line has been modified speculatively in an ASF atomic phase (and not committed), it can only be present in the LLB associated to this ASF critical section, LLB B, because the modifying write instruction also waits until all other LLBs have finished rolling back, again either by waiting for (delayed) ACKs or by removing the entries from the *other* LLBs immediately. The locked load that tries to add the location to LLB A in turn triggers an abort

and write-back of B, which also removes the cache line in question before the load can read the data, either by delaying the ACK or removing the element directly from B (inside the simulator).

□

*Remark:* This proof relies heavily on the notion of stores “taking effect”. In the simulator, this is the point when the store operation reaches the commit stage and updates the (single) shared memory. In native hardware, the precise moment of “taking-effect” may be more difficult to determine, because data may be spread and replicated across different caches and memory nodes. The employed consistency model uses the coherency mechanisms to safely determine that point in time when a write actually has taken effect, at least for the used model of “processor consistency”.

**Theorem 4.2.** *There is always a single (logically) valid version of data in the entire system, including all ASF-related backup storage (LLB).*

*Proof.* Let us have a look at the number of present LLB lines for that given element of data:

1. If there is no LLB-entry for that cache line, the valid data can be found in memory (in the simulator) or is specified by the underlying cache coherency protocol.
2. There is exactly one LLB-entry for the line: The data backed up in that LLB entry is the correct data for all instructions not running in the ASF-CS owning that LLB entry. The version inside the LLB is the version used by the instructions inside that section, but that section is entirely speculative still, hence does not contribute to the logically valid state of the system yet.
3. There are multiple LLB-entries in the system containing a certain element of data: Lemma 4.1 states that all present copies of this element must have the same content. (2) then applies accordingly for these identical copies.

□

Theorem 4.2 states that forwarding from any of the LLBs which contain the needed line is possible. The proofs contain possible implementations for native hardware and specify the behaviour with the simpler interconnect model I implemented in the simulator.

### 4.3 Accelerating TinySTM

The analysis in Section 3.3.2 points out already that only few ad-hoc improvements to TinySTM [FFR08] are possible with ASF. Looking at the fast-path (Figure 4.3) in TinySTM’s transactional load, it is quite

```

stm_word_t stm_load(stm_tx_t *tx,
                   volatile stm_word_t *addr)
{
    ...
    lock = GET_LOCK(addr);
    /* Read lock, value, lock */
    l = ATOMIC_LOAD_MB(lock);

restart:
    if (LOCK_GET_OWNED(l)) {
        /* Locked: Check if by us, if not abort. */
        ...
    }
    value = ATOMIC_LOAD_MB(addr);
    l2 = ATOMIC_LOAD_MB(lock);
    if (l != l2) { l = l2; goto restart;}
    /* Check timestamp */
    version = LOCK_GET_TIMESTAMP(l);
    /* Valid version? */
    if (version > tx->end) {
        /* No: Revalidate read-set, if that fails abort. */
        ...
        /* Recheck lock, perhaps locked during validation. */
        l = ATOMIC_LOAD_MB(lock);
        if (l != l2) goto restart;
    }
    /* Good version: Add to read set and return value */
    if (tx->r_set.nb_entries == tx->r_set.size) {
        /* Enlarge read set */
    }
    r = &tx->r_set.entries[tx->r_set.nb_entries++];
    r->version = version;
    r->lock = lock;
    return value;
}

```

Figure 4.3: Simplified version of the original `stm_load` operation (TinySTM, write-through version).

clear that there are only few bottlenecks, mostly those due to handling of meta-data.

Monitoring the lock associated with the read memory location removes the additional load that is used to ensure that the lock has remained in the unlocked state. Listing 4.4 highlights the differences to the original version in Listing 4.3.

In Section 3.3.2, I have also proposed an extension with more drastic changes to TinySTM's handling of the read-set. Locations are not monitored with the STM until ASF's capacity limit is reached, Listing 4.5 shows the respective wrapper to the `stm_load` function.

The original STM uses read-set validation to ensure that the application is running only on consistent state. ASF did initially not contain a nondestructive `VALIDATE` instruction, but coupled validation to `ACQUIRE`, which validates the speculative set and does not allow further additions to it after successful

```

stm_word_t stm_load_asf(stm_tx_t *tx,
                       volatile stm_word_t *addr)
{
    ...
    lock = GET_LOCK(addr);
restart:
    /* Start monitoring the lock */
    l = lock_load64((stm_word_t*)lock);
    acquire(fail, asf_store, l);
    if unlikely (fail) {
        acquire_fail_clobber(asf_store);
        goto restart;
    }
    if (LOCK_GET_OWNED(l)) {
        ...
    }
    value = ATOMIC_LOAD_MB(addr);
    /* No need to re-read the lock here! */
    version = LOCK_GET_TIMESTAMP(l);
    if (version > tx->end) {
        ...
        /* No need to re-check the lock here! */
    }
    release(); /* Stop monitoring the lock. */
    /* Good version: Add to read set and return value */
    ...
}

```

Figure 4.4: Monitoring the lock during an `stm_load` with ASF.

validation. Putting new entries under ASF monitoring is only possible after the commit of the atomic phase, which clears the previously acquired list of locations.

This behaviour originates from the idea that data specified with locked-loads is not used in an “unsafe” manner before the ACQUIRE. Given the low complexity of validation in ASF, I have decided to introduce a simple VALIDATE instruction that validates the existing speculative set, yet still allows further extensions (up to the capacity limit). VALIDATE is then used to validate the entire speculative set on every call to `stm_load_asf` (Figure 4.5) to ensure that previously read locations are still valid. The cost for revalidation in the STM is reduced, because the part of the transaction’s read-set that is monitored by ASF can just be revalidated using the VALIDATE instruction, saving the cost of the original revalidation algorithm for these elements.

```
stm_word_t stm_load_asf(stm_tx_t *tx,
                       volatile stm_word_t *addr)
{
    stm_word_t res;
    ulong cache_addr = (ulong)addr & ASF_LINE_MASK;
    if unlikely (tx->asf_last & ASF_HINT_SOFTWARE)
        return stm_load(tx, addr);

    /* Aliasing on the last ASF line */
    if (tx->asf_last == cache_addr) {
        res = ATOMIC_LOAD_MB(addr);
        goto load_validate;
    }
    /* ASF capacity exceeded */
    if unlikely (tx->asf_entries >= ASF_ENTRIES) {
        tx->asf_last = ASF_HINT_SOFTWARE;
        return stm_load(tx, addr);
    }
    /* Check that the location is unlocked */
    res = lock_load64(addr);
    tx->asf_last = cache_addr;
    tx->asf_entries++;
    stm_word_t l = ATOMIC_LOAD_MB(GET_LOCK(addr));
    if unlikely ( LOCK_GET_OWNED(l) )
        return stm_load(tx, addr);
    /* Validate recent ASF read-set */
load_validate:
    long asf_inv;
    asf_validate(asf_inv, tx->asf_entries);
    if unlikely (asf_inv) {
        stm_abort_self(tx);
        return 0;
    }
    return res;
}
```

Figure 4.5: Transactional load, extended to use ASF to bypass the STM.



## 5 Experimental Evaluation

This chapter evaluates the design (see chapter 3) and implementation issues (refer to chapter 4) presented in the previous chapters. Let us quickly summarise what has been shown so far:

- An instruction set extension, called Advanced Synchronisation Facility (ASF), has been introduced. ASF allows programmers to specify sections of code to be executed atomically in a multi-core environment. (refer to Section 3.2)
- An existing integrated simulator, PTLsim ([You07]), has been introduced and extended to support ASF and multi-core execution. (See Sections 3.1, 4.1 and 4.2.)
- Lock-free implementations of concurrent linked lists using ASF have been proposed in Section 3.3.1.
- Two attempts at accelerating an existing modern Software Transactional Memory system (STM) have been introduced (Section 3.3.2) and discussed in more detail (Section 4.3).

This chapter first (section 5.1) describes the setup used for the benchmarks, including the modifications which have been made to existing benchmarks. In the following section (section 5.2) we will lay the foundation for the applicability of the simulated results by comparing simulated and native execution and estimating simulator precision.

Sections 5.3 and 5.4 will present results for the lock-free data structures and accelerated STM, respectively. Most results will be simulated, but native results will be shown for reference when available.

In each of the sections, results are discussed subsequently to the actual results, final conclusions can be found in Chapter 6.

### 5.1 Evaluation Setup

Given the high cost for building a new processor core, instruction-set extensions are initially evaluated with processor simulators. As already mentioned, I have chosen PTLsim and implemented ASF as described in Section 4.2. Additional modifications have been made to the simulator, partially bug fixing and architectural enhancements to bring its architecture more in line with current AMD processors. I

extended the work Yourst introduced in [You07] making PTLsim behave similar to an AMD K8 core.

For the native measurements, I used a dual-socket system, equipped with two AMD family 10h Opteron (Barcelona) processors running at 2.2 GHz. Each processor consists of four cores, each with private caches (L1D & L1I: 64 KByte, 2-way set-associative; unified L2: 512 KByte, 16-way set-associative) in an exclusive hierarchy, and a shared (between the four cores) L3 cache (2 MB, 32-way) with a mostly exclusive (sharing-aware) configuration. Both sockets are connected with HyperTransport links, making main memory at each socket available in a ccNUMA fashion.

Benchmarking is done using the well-known “intset” workload, a data structure that provides methods for insertion, removal, and query of a set of integers. Some implementations of intset and a test harness are included Felber’s TinySTM distribution<sup>1</sup> [FFR08], the other implementations are added and benchmarked in the same context. TinySTM itself is also extended to use ASF primitives, as sketched in Sections 3.3.2 and 4.3. Several small modifications have been made to the original test harness included in TinySTM, mainly to reduce variance (pinning of worker threads to physical cores), to use PTLsim effectively (switching to simulation mode after the initialisation) and to make the benchmark more robust (changed benchmark end-criterion from time-based to number-of-intset-operations based).

Because simulation is rather slow (depending on the number of simulated cores around 100,000 times slower than native execution), I have limited the number of operations on the intset to 5000 per thread, variance is reduced by pinning worker threads statically to CPU cores (avoiding the operating system’s load balancer) and maintaining a fixed seed for reproducibility. Other parameters of the benchmark from TinySTM remain at their default values (set with 256 entries, entries range from 0 to 65535, 20 % rate of updates) except noted otherwise.

Additionally to the implementations mentioned above (sorted singly- and doubly-linked lists using ASF) and those contained in TinySTM (sorted singly-linked list and red-black-tree using STM), I have added an implementation of Harris’ lock-free singly-linked list, as described in [Har01].

Another singly-linked list simply protected with a single spin-lock (test-test-and-set) and a single threaded implementation without any locks mark the limits of (bad) scalability and excellent single-thread performance.

---

<sup>1</sup>Available from <http://www.tinystm.org>

## 5.2 Simulator Precision

### 5.2.1 Motivation

Simulation is not an end in itself, but it is always a tool to predict results which are not feasible to obtain “for real”. If we want to use a simulator to predict the performance impact of a hardware extension, we actually want to know how native hardware containing this extension would perform. To even remotely be able to deduce such meaning from simulator results, we have to ensure a few things first: First of all, we would like to know that the unmodified simulator actually models the existing hardware correctly. Ideally, the simulation would match native execution results for all workloads and under all possible conditions. Realistically, we will have to live with good similarity for a set of workloads, because simulators are simplified models of the actual hardware for reasons of feasibility. For reasons of time I will concentrate on showing similarity for workloads which are similar to those I expect to make use of the hardware extension eventually.

Second, once we have established a link between the simulated results and what native hardware tells us, we can add the extension to the simulator. It is likely that the actual implementation of the extension inside the simulator varies from the way this extension would be implemented in hardware (if that is actually known in full detail yet). With care we can ensure that the implementation inside the simulator still captures the essence of the envisioned native one. This step is crucial, but can only be shown correct by persuasion, as native hardware for direct comparison is not available.

If we are confident about the accuracy of these two links (similarity of the original native hardware and simulator and similarity of the extension in both domains), we can assume that the results of the enhanced simulator have at least some correlation with results we expect from envisioned native hardware.

Ensuring simulator precision as a first step is therefore crucial for evaluating the extensions.

### 5.2.2 Single-Core Performance

Figure 5.1 compares throughput for different implementations of the intset interface on native hardware and inside the simulator. Simulator precision varies, dependent on the actual implementation, but is within about 20 % of native performance, except for the implementation using just a single big lock and the one without any synchronization. Both of them contain a short loop that traverses the list, inspected in Section 4.1. The simulator has been tuned using this loop, so the performance difference for Figure 5.1 was slightly surprising.

Quick testing revealed that increasing the number of operations on the intset executed for each bench-

mark run would raise the throughput to the simulated levels. In addition, the native testing environment could have had some impact: PTLsim uses Xen [BDF<sup>+</sup>03] to virtualise hardware and allow transparent switching between simulation and native execution. In simulation mode, PTLsim can simulate kernel execution, but cannot correctly simulate the hypervisor. Hence, native execution under Xen has an added system layer, which according to Barham *et al.* [BDF<sup>+</sup>03] is usually small. The timed portion of the benchmarks does not contain any I/O or traps into the hypervisor and Xen’s scheduling has been simplified by statically pinning virtual machines (called domains) to native cores 1:1.

These factors suggest that the performance impact of Xen’s added layer should be negligible, but comparisons between native executions with Xen and without (Linux on bare hardware) prove otherwise, as can be seen in Figure 5.2. It compares the implementation with a single lock (“biglock” or just “lock”) and the one without any present locks (“nosync”) on a single core for different numbers of intset operations per benchmark run (total 100 runs).

This measurement reveals that for the benchmark size used for this thesis, Xen is about 20 % slower than native hardware. The performance increase for larger sizes suggests that this penalty is amortized over time, maybe connected with Xen’s lazy page-mapping, called *shadow paging*. Page-faults that occur during the first traversal of the list could then need handling inside the hypervisor and become much more expensive than on native hardware.

Figure 5.3 shows the single-core throughput for all other tests compared inside the simulator and Linux. The simple cases (“lock” and “nosync”), which I have used to optimise the simulator, are now much closer in performance (from 45 % to 9 %). The other benchmarks do also profit from the leaner test environment and have larger throughput than simulated. The simulator obviously needs more tuning for these more complex workloads.

### 5.2.3 Multi-Core Performance

Multi-core results are shown in Figures 5.4 and 5.5, the differences between native execution in Xen and on plain Linux manifest also for multithreaded execution. Figure 5.6 compares the three environments, Xen, Linux and PTLsim, in more detail for each data structure. For the remainder of this chapter, all native measurements are made using plain Linux (unless noted otherwise).

Although PTLsim captures the general trend, simulation results scale better, especially at working thread numbers larger than four. This deviation is caused by the simulation’s interconnect model (see Section 3.1), which does not differentiate between local—between cores in the same socket—and cross-socket

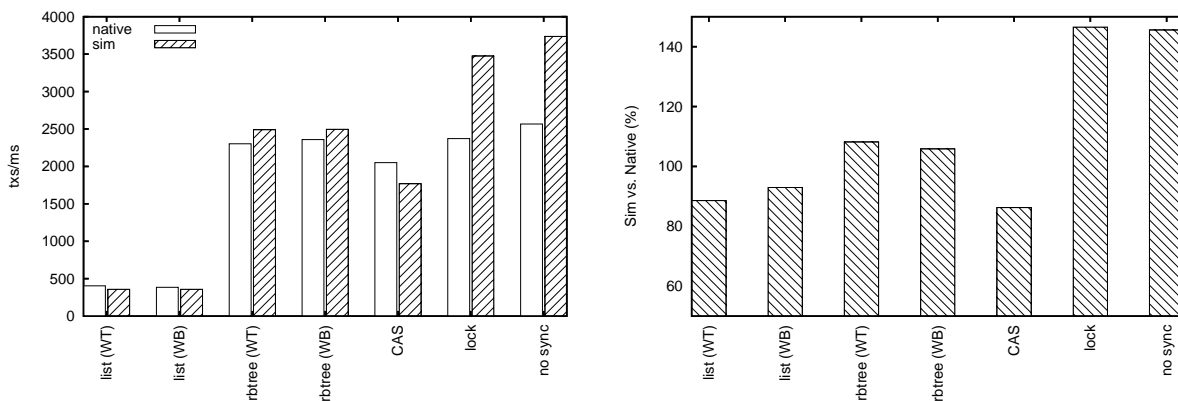


Figure 5.1: Single-core performance for native (Xen) and simulated execution. Throughput on the left, simulator results relative to native performance on the right (where values > 100 % denote higher simulated than native throughput).

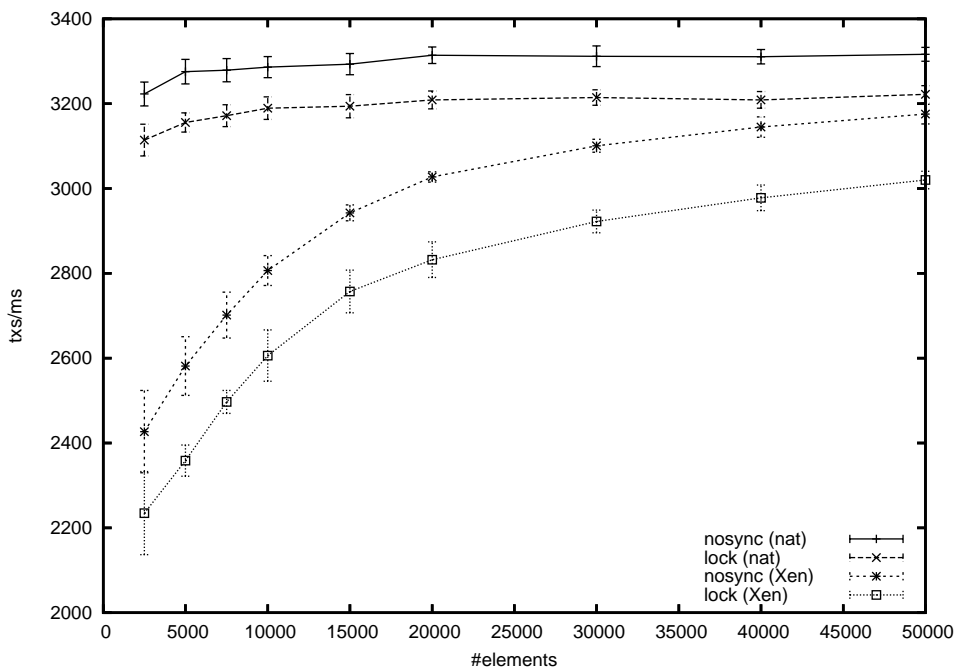


Figure 5.2: Single-core throughput of linked list implementations under Xen and plain Linux for different benchmark sizes.

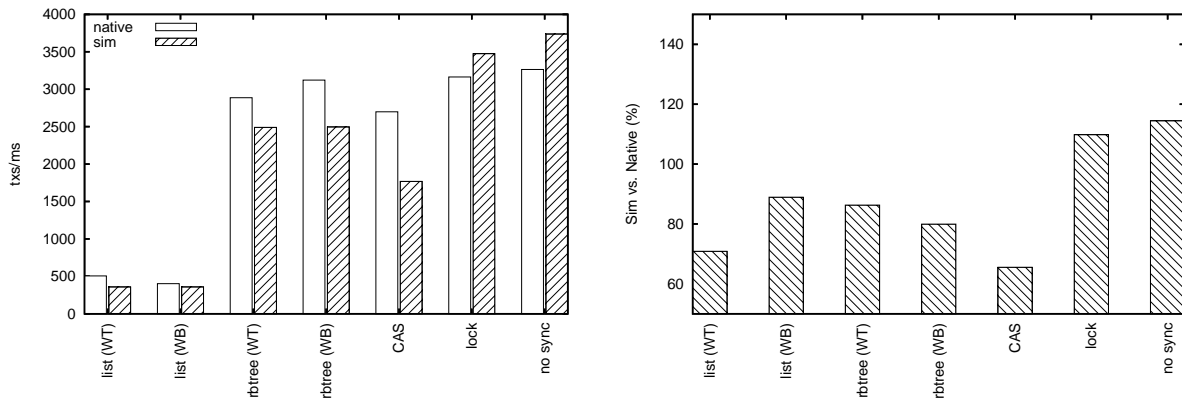


Figure 5.3: Single-core performance for native (plain Linux) and simulated execution. Throughput on the left, simulator results relative to native performance on the right.

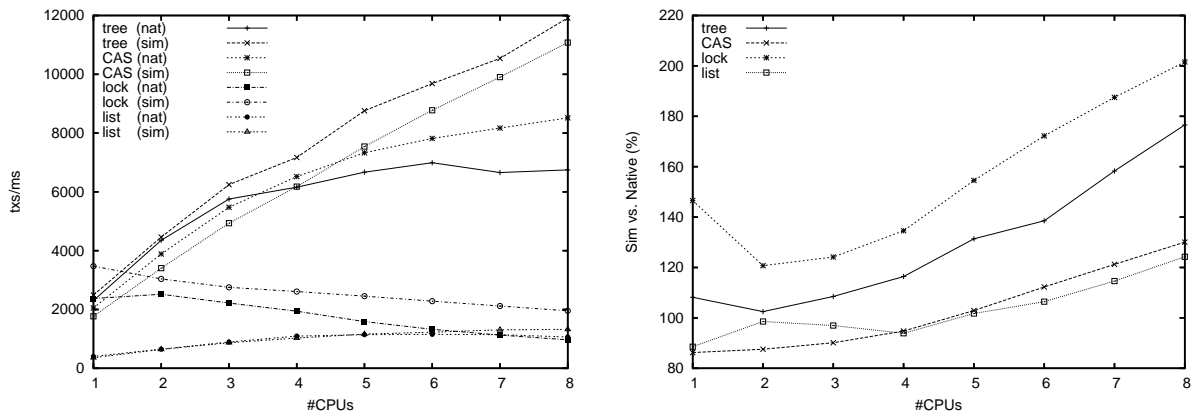


Figure 5.4: Multi-core performance for native (Xen) and simulated execution. Throughput on the left, simulator results relative to native performance on the right.

communication<sup>2</sup>. On native hardware, these links differ in both in latency and available bandwidth.

Additionally, PTLsim and native hardware differ in the way they treat atomic read-modify-write instructions: PTLsim simply grabs a simulator-internal lock for the affected memory location (without any delays), whereas native hardware executes the entire atomic read-modify-write (RMW) instruction speculatively and only then employs a costly locking scheme. This leads to highly different behavior for atomic RMW instructions on contended memory locations, such as spin locks (as used in the “big-lock” implementation).

<sup>2</sup>One might characterise the simulated CPU as an eight-core chip without shared caches.

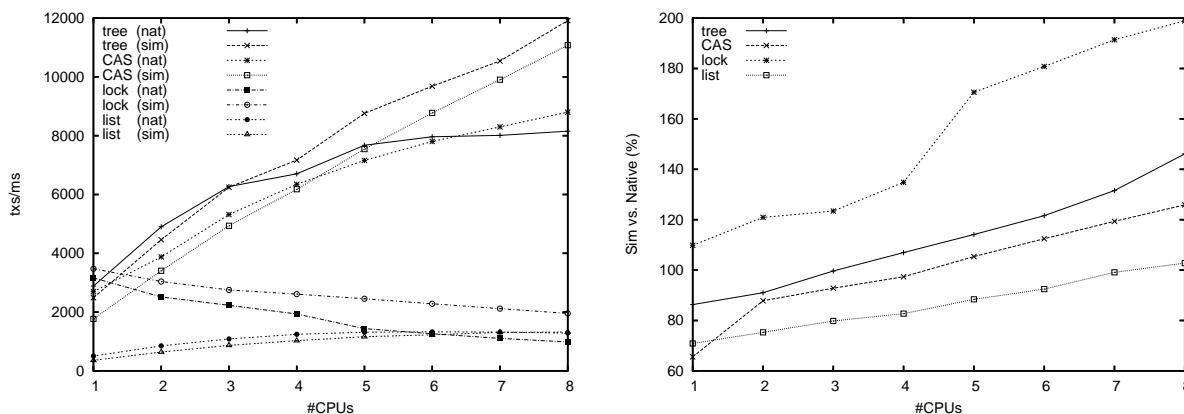


Figure 5.5: Multi-core performance for native (Linux) and simulated execution. Throughput on the left, simulator results relative to native performance on the right.

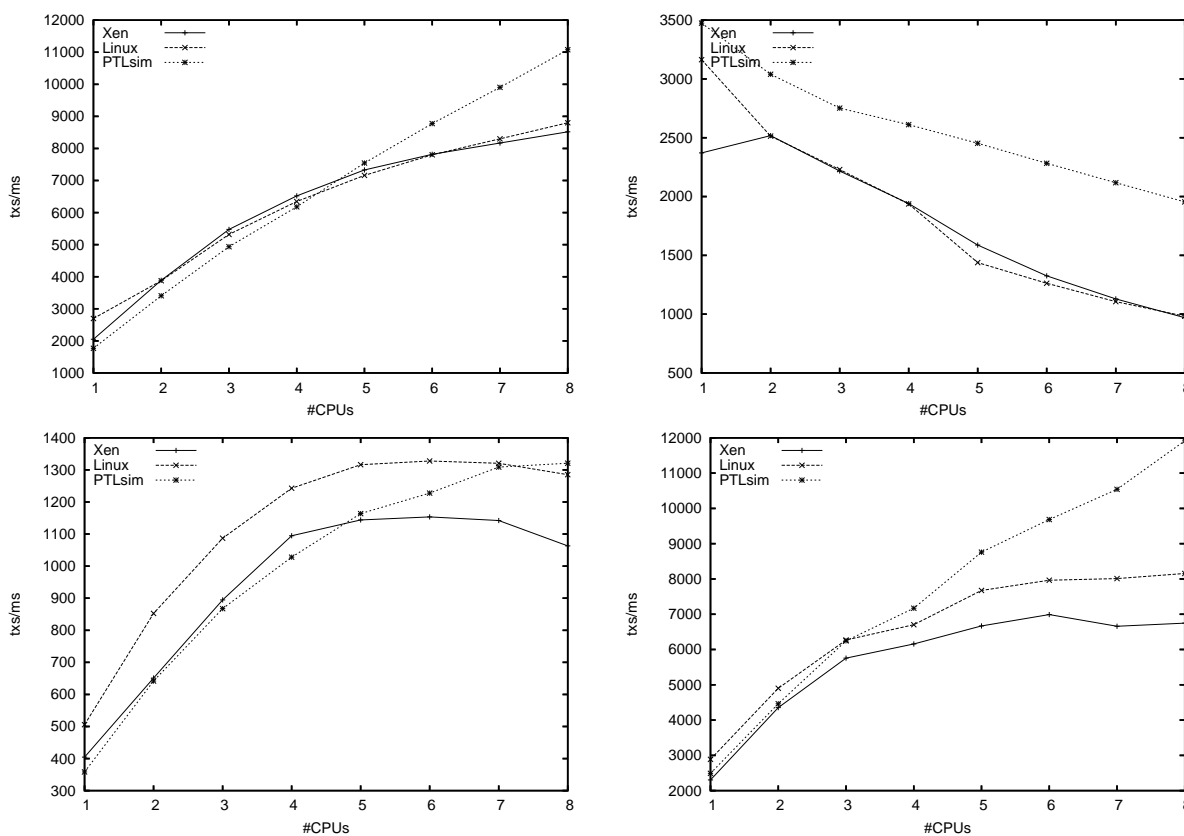


Figure 5.6: Multi-core performance for different implementations executed in Xen, Linux and PTLsim.  
 Top-left: Harris' lock-free CAS-based, Top-right: "biglock", protected by single lock  
 Bottom-left: linked list using STM, Bottom-right: red-black-tree using STM

### 5.2.4 Memory Latency

To a large extent, performance of a processor core is determined by the performance of the memory hierarchy. To test and adapt memory latencies, I have implemented a tool that times various ways of accessing memory, using the original idea from McVoy and Staelin's "Imbench" [MS96].

Figure 5.7 compares the simple read-latencies for the fully tuned simulator and native hardware. As the native system has NUMA memory access (two connected sockets with memory local to each socket), the access to the remote main memory takes slightly longer (about 17 cycles: 205 local and 222 remote). The graph also shows the different cache organisation: AMD's Barcelona core uses a (mostly) exclusive cache hierarchy which adds the usable capacity of caches. PTLsim in contrast has an inclusive hierarchy, as every line has to be stored in outer caches as well the total capacity is equal to that of the outermost cache.

Besides tweaking the cache's parameters, I have added a second-level TLB and made the L1-cache virtually indexed. The results of these additions can be seen in Figure 5.8, they help to bring PTLsim more in line with native hardware.

The simplified interconnect model introduced in Section 3.1 can specify a cross-cache delay which models the time needed for moving a cache line between caches of different cores. I have adapted the latency benchmark to measure this delay, results can be found in Figure 5.9. For transfers from L1 and L2 caches on the same socket, the latency is at around 140 cycles. As soon as the working-set starts to reside in the shared L3 cache, access proceeds with the normal L3 latency, because the L3 cache is shared among the four cores on each die. The latency for cache line transfer between the two sockets is roughly equivalent to the remote memory-access latency, irrespective whether data resides in one of the caches of the remote processor.

The performance for the socket connected only remotely to the memory where the working set resides is interestingly different from that of the local node. Although data is present in L1 and / or L2 cache on the same die on the remote socket, the high latency hints that data is not directly forwarded across on that same socket. A possible reason might be that the remote core that tries to access the data but does not have a copy in the caches sends a request to the socket that owns (is local to) the memory, by default. This cross-socket communication would explain why the latency is similar to the measured cross socket latency. The shared L3 cache could detect such an outgoing probe and reply to it if the cache contains the data. This explains the reduction of the latency back to L3 levels, once data is too large for L1 and L2 caches and has to be present in the L3 cache on that remote socket.

With the present results, I have set the cross-cache latency in PTLsim's simplified interconnect network

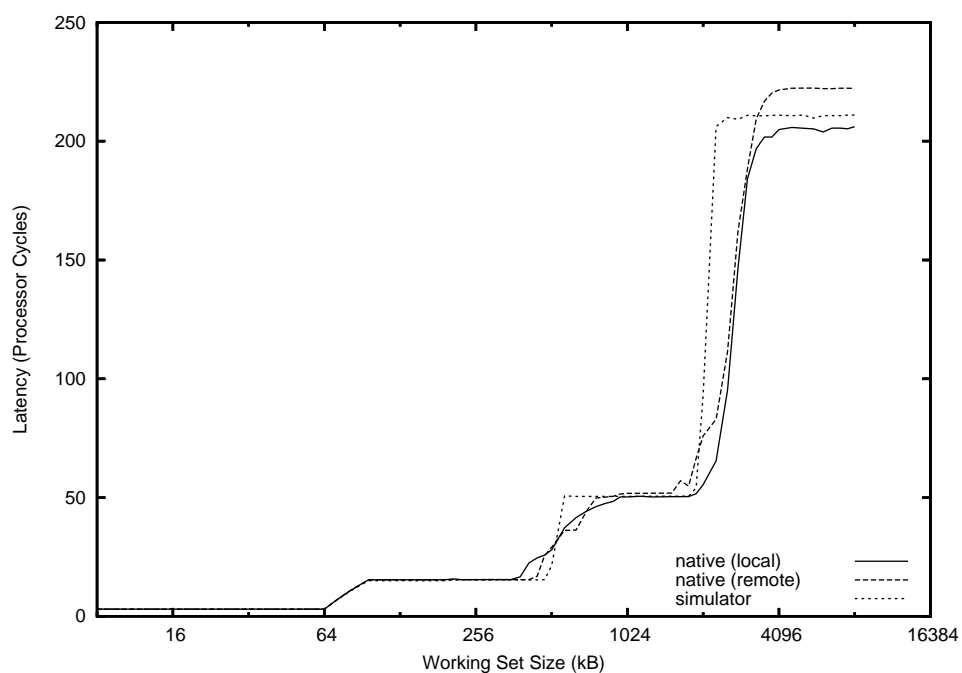


Figure 5.7: Memory read latencies for an AMD Barcelona core (on local and remote NUMA nodes) and PTLsim, tuned and with all enhancements (L2-TLB, virtually-indexed L1 data cache).

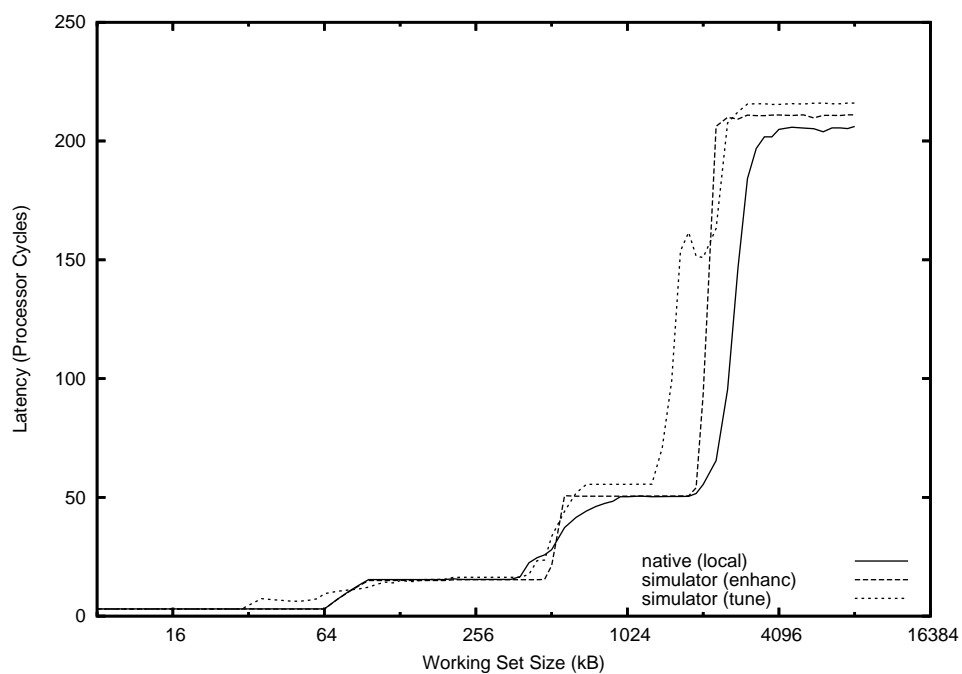


Figure 5.8: Memory read latencies highlighting effects of virtually-indexed L1D caches and L2-TLBs by comparing native (local NUMA node) performance and simulation results. “Tune”ed simulator with adapted cache parameters (size, associativity), “enhanc”ed version with virtually-indexed L1 data cache and L2-TLB.

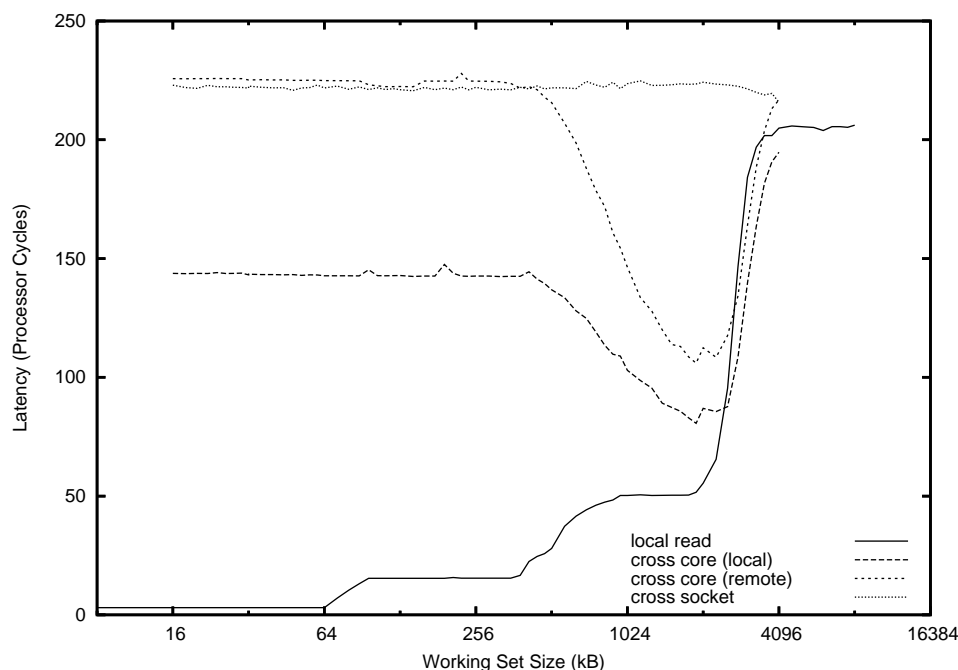


Figure 5.9: Cache-to-cache latencies on AMD’s Barcelona. “Cross core” accesses data in another core on the same die / socket, “cross socket” accesses data in another core in a different die / socket. “Local” experiments ran on the socket closest to the accessed memory, “remote” on the one furthest away.

to 140 cycles, approximating the delay between separated caches on the same die in native hardware.

Previously, I explained the different realisation of atomic RMW primitives. In Figure 5.10 an atomic compare-and-swap operation accesses the same element as a variable number of concurrent threads. It can clearly be seen that the time needed for each atomic compare-and-swap is proportional to the number of concurrent operations, with an offset between only local and only remote concurrent accesses.

PTLsim has very different implementation of atomic RMW instructions and has an interconnect network without any delay and bandwidth limitations. The same test executed inside the simulator yields the different results in Figure 5.11. Performance under read-contention does not degrade, as the RMW instructions simply grab an internal lock and delay the reads. With write-contention there is an offset, because the RMW instruction will not find the datum in the local cache, as it has been invalidated by the write-instructions. Once the data is present in the cache, the CAS can finish under the protection of the simulator’s internal lock.

Contention for these locks can be seen, when multiple atomic RMW instructions on different cores process the same cache line. Then the performance of the single CAS is reduced proportionally to the number of competing cores.

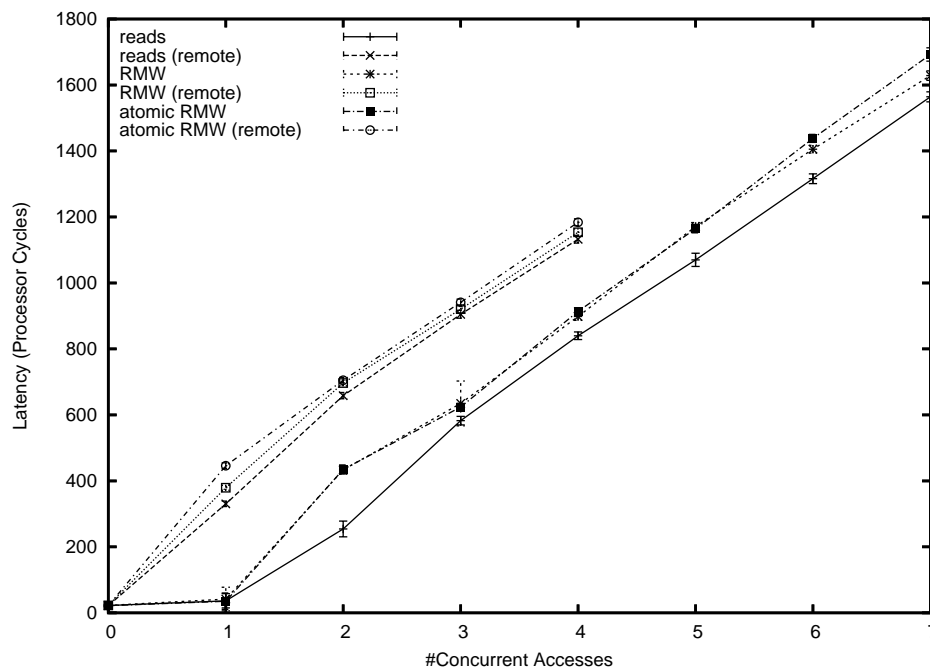


Figure 5.10: Processor cycles needed for an atomic compare-and-swap instruction under varying levels and types of contention on native hardware. “Read”, “RMW” and “atomic RMW” are the instructions executed simultaneously by threads on different cores. “Remote” indicates threads are running on a different die / socket than the atomic compare-and-swap. Otherwise, concurrent threads populate the cores on the same die as the atomic CAS instruction first. All instructions access the same memory location.

These differences explain why the “biglock” benchmark has less degrading performance under PTLsim. The atomic RMW instruction used to acquire the lock is not slowed down by the large number of contending readers also trying to enter the critical section.

### 5.2.5 Discussion

Linking results of native and simulated execution is crucial for predicting results for which no native measurement for comparison purposes can be made. This section has shown that the simulator is reasonably correct for “intset” benchmarks and memory latency from one to four cores. For core counts larger than four it deviates, but the cause for this deviation is well understood. In a sense, PTLsim marks the upper bound of scalability if the interconnect is optimised as much as possible. With that in mind we can use the simulator to make helpful predictions about the performance of ASF.

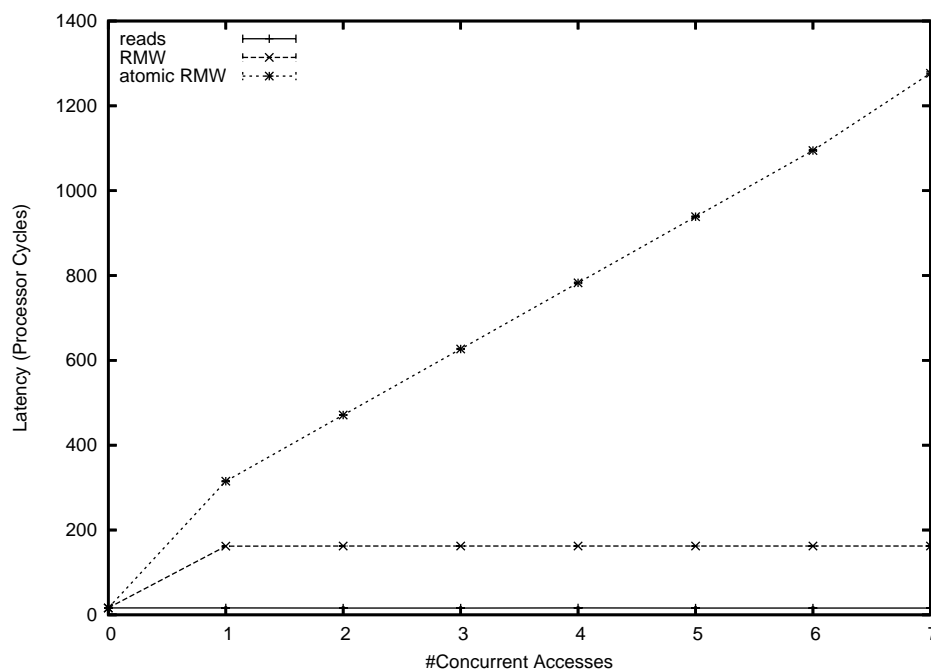


Figure 5.11: Simulated processor cycles needed for an atomic compare-and-swap operation under varying levels and types of contention in PTLsim. In contrast to Figure 5.10 there is no notion of dies or sockets in PTLsim. Therefore “remote” measurements make no sense.

### 5.3 Lock-free Data Structures

Well designed lock-free data structures usually offer superior performance to those implemented with coarse-grained locks and STM, because of increased parallelism and reduced overhead. Figure 5.12 shows the results for the various implementations of the intset interface: singly- (ASF 1x) and doubly-linked (ASF 2x) lock-free list using ASF from Section 3.3.1, Harris’ CAS-based lock-free implementation (CAS) and the version that uses a single lock (lock).

It can be seen clearly that both ASF implementations outperform the CAS-based implementation. The singly-linked ASF list is faster than the doubly-linked ASF list, because the latter eagerly checks the back references in elements, needing more cycles during traversal.

The performance advantage over the CAS-based implementation comes from three facts: Firstly, Harris’ CAS-based implementation marks deleted elements and removes them later, because removing them from the list immediately is not possible with CAS. ASF lists, to the contrary, do not keep deleted elements in the list for later clean-up. Immediate removal keeps the list short and saves the overhead of marking and cleaning up later. Secondly, ASF does not guarantee progress and thus does not need a bus-based protocol when contention on the same memory location occurs. Finally, ASF does not serialise the other memory accesses in the core, leaving more potential for parallel and out-of-order execution.

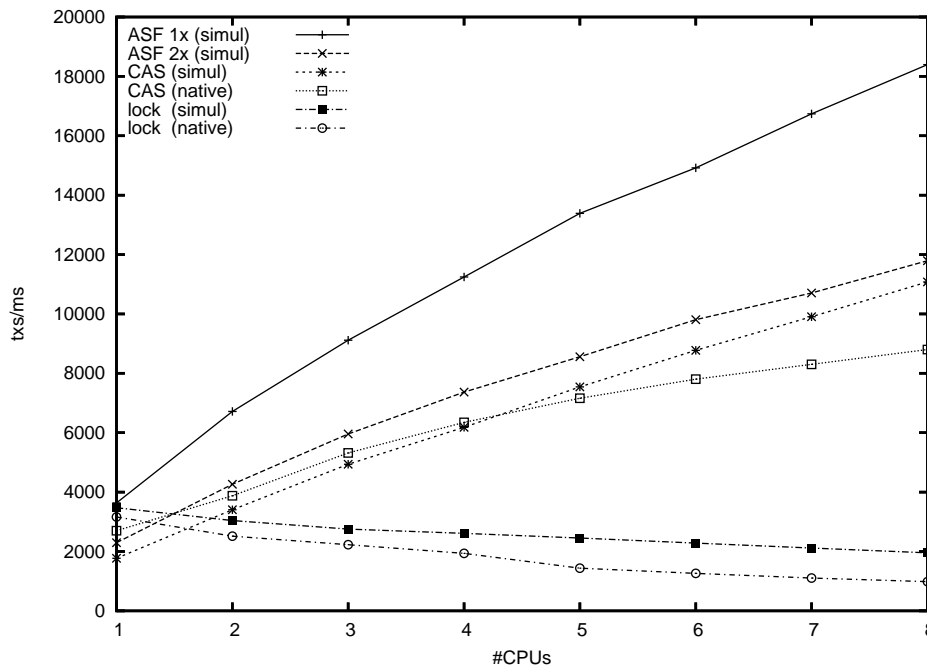


Figure 5.12: Different implementations of lock-free lists (Harris’ “CAS”-based and singly- / doubly-linked lists with ASF, “ASF 1x” and “ASF 2x”) and single “lock” implementation. “Native” results displayed for reference where available, “ASF” implementations without native reference, because of outstanding hardware support for ASF.

## 5.4 Acceleration of STM

Section 3.3.2 introduced several ideas for speeding up TinySTM with ASF. Figure 5.13 shows the results for the naive approach, proving that this approach is not effective at accelerating TinySTM with a throughput penalty of about 40 %. This is likely because of the large number of ASF critical sections (one for each `stm_load`) and the associated ordering constraints between them: The current implementation does not allow multiple ongoing ASF-CS as explained in Section 4.2. This causes reduced ILP for the traversal loop and consequently, bad overall performance. If ASF-CS could be interleaved or a clever compiler/linker could combine several `stm_loads` to execute within a single ASF-CS, performance should increase. This is demonstrated by manually combining the two `stm_loads` (value and next-pointer) for the list traversal (see Figure 4.3 in Section 4.3) and shown in Figure 5.13 as “ASF2”. The throughput is still reduced, but the penalty has reduced from 40 % to about 29 %.

We can conclude that the naive attempt at improving TinySTM’s performance is not useful and that the implementation of ASF without allowing any overlap carries large performance penalties.

In order to improve TinySTM more substantially, I proposed a more extensive change to TinySTM in Section 3.3.2 where ASF would track parts of the read-set, thereby reducing validation and book-keeping

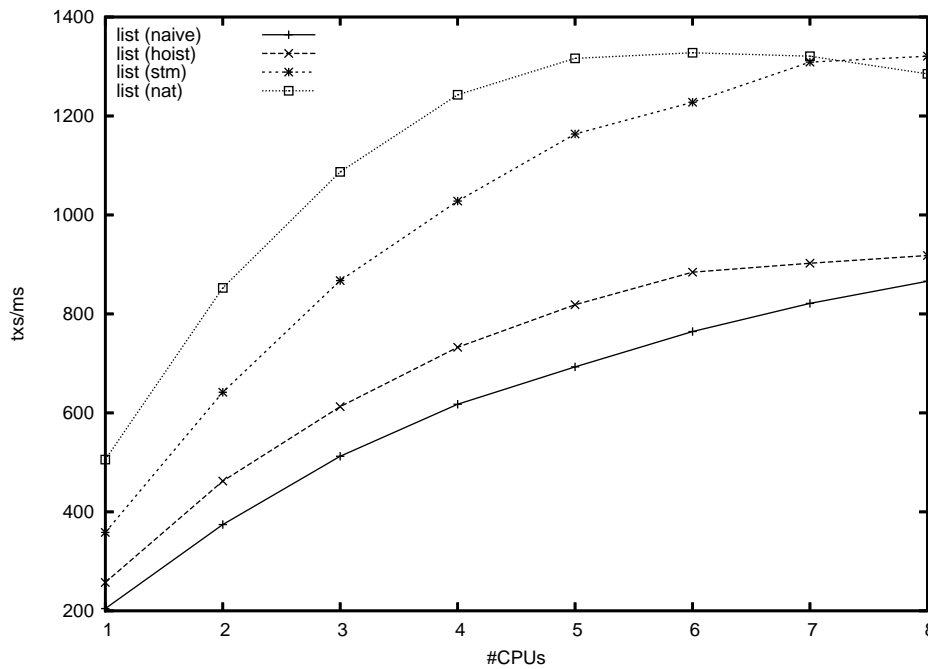


Figure 5.13: Throughput of linked lists using a naively accelerated implementation of TinySTM.

“Naive” implementation with one ASF-CS per `stm_load`, “hoist” statically combines two `stm_loads` and protects them with one ASF-CS. “Stm” is the unmodified implementation of TinySTM. All results obtained in PTLsim, except “nat”, unmodified TinySTM on native hardware for reference.

overhead. Figures 5.14 and 5.15 compare a standard TinySTM to this accelerated version of TinySTM. The large tree and the linked list both benefit from the reduced re-validation overhead. The number of re-validations grows with the level of concurrency (frequent validity interval extensions in TinySTM) and thus the accelerated version benefits more at higher CPU counts.

This can also be seen in Figures 5.16 and 5.17 that show scalability with the accelerated TinySTM. ASF improves scalability for the linked-list by 17 %. The red-black-tree scales similarly for both implementations, the performance improvement of 13 % with eight cores is achieved by a lower overhead: The single-core experiment shows a 15 % higher throughput for the accelerated STM. Surprisingly, the small tree does not profit from the acceleration, although its entire read-set should fit into ASF. I believe this is caused by the small read-set, which makes the validation in the standard STM still reasonably fast. This reduces the advantage of ASF’s fast VALIDATE and brings out some unknown overhead. Investigation where this overhead of the accelerated STM actually comes from and how it can be avoided is matter of further research.

Figures 5.14 and 5.15 also contain the performance of the native execution using the unmodified STM for reference. As I have shown previously in Section 5.2, the simulator does not yet model the limitations

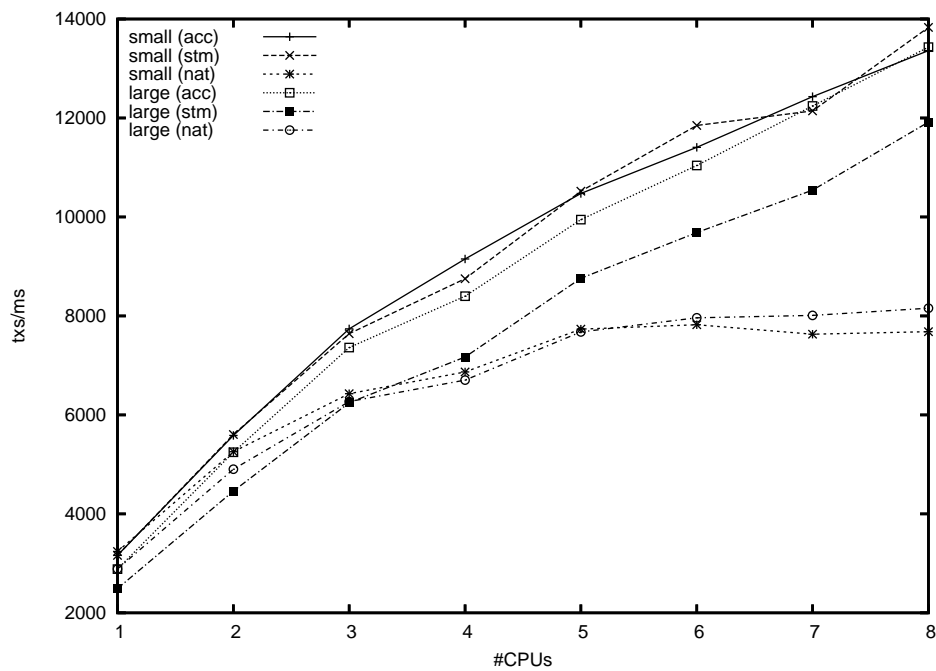


Figure 5.14: Comparison of ASF-accelerated (“acc”) and standard TinySTM (“stm”) with red-black-trees containing 128 (“small”) and 256 (“large”) initial elements. All measurements done in PTLsim, except “nat”, standard TinySTM on native hardware for reference.

of the interconnect between the two sockets in the system, which obviously limits performance for the red-black-tree on native hardware for CPU counts greater four, when cross-socket communication is necessary.

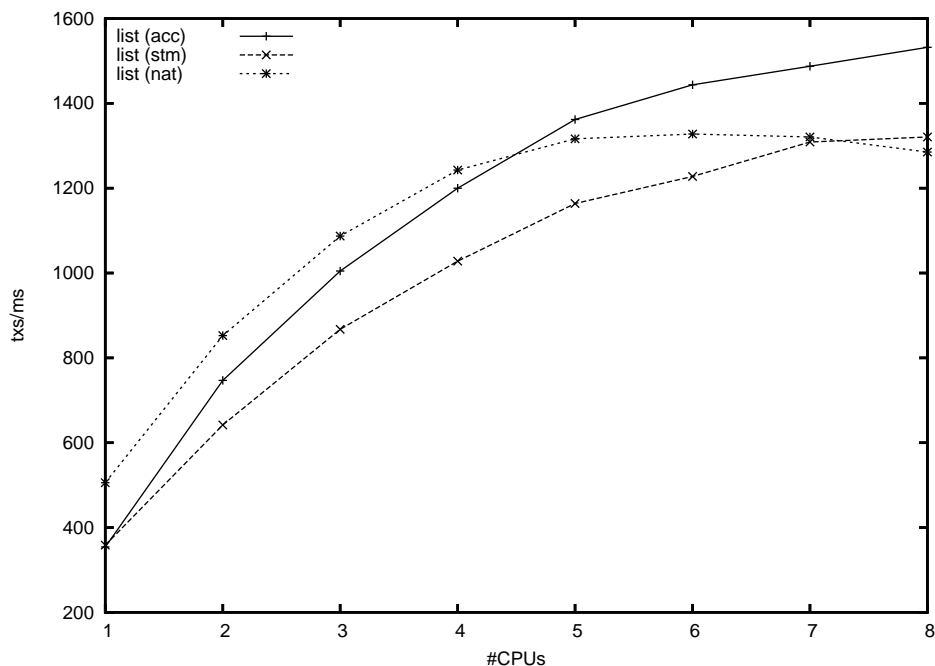


Figure 5.15: Accelerated version of TinySTM (“acc”) increasing throughput for a linked list compared to unmodified TinySTM (“stm”). Both experiments executed in PTLsim, unmodified TinySTM on native hardware (“nat”) provided for reference.

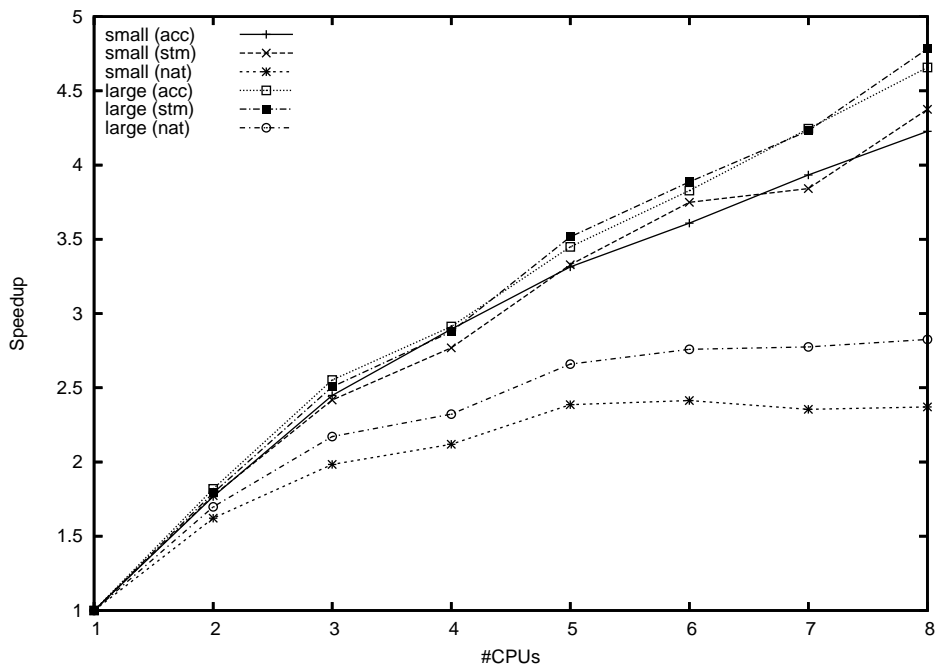


Figure 5.16: Resulting speed-up for the measurements in Figure 5.14, red-black-trees with ASF-accelerated TinySTM.

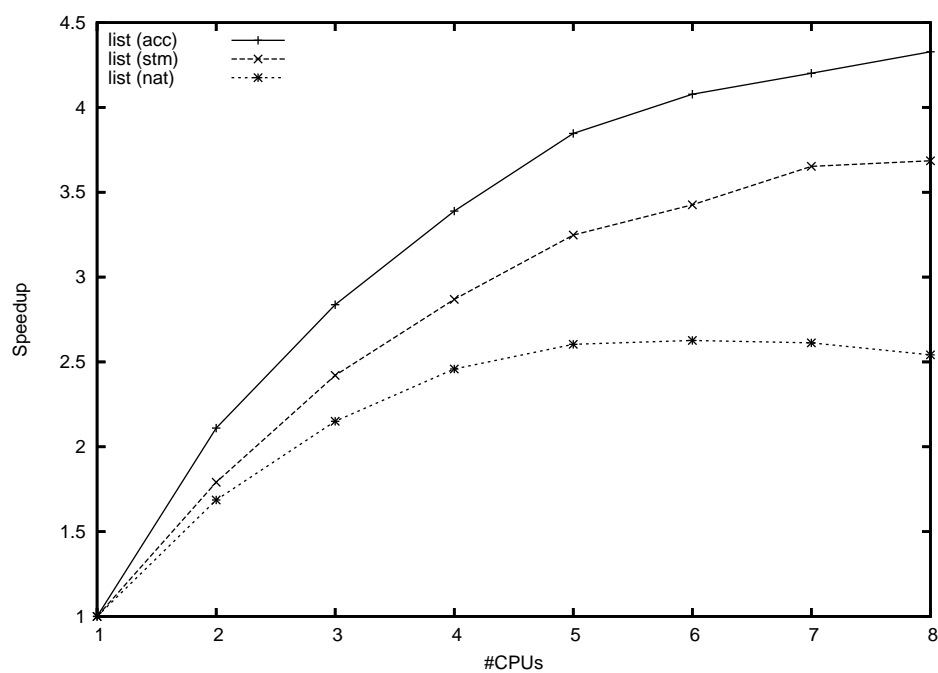


Figure 5.17: Resulting speed-up for the measurements in Figure 5.15, linked lists with ASF-accelerated TinySTM.



## 6 Conclusion

### 6.1 Summary

In my thesis “Hardware Acceleration of Software Transactional Memory” I have introduced the “Advanced Synchronization Facility”, a proposal for hardware extensions supporting atomic lock-free blocks in hardware. In the thesis’ introduction I raised the questions whether ASF could reduce the overhead of software transactional memory, and whether it could simplify lock-free programming.

ASF’s fitness for both purposes has been evaluated by implementing ASF in a microprocessor and full-system simulator called PTLsim. For that I have extended PTLsim to allow multi-core simulation with a simplified cache coherency model and created a model of ASF in the simulator.

ASF has showcased its potential for simplifying lock-free programming with a novel lock-free implementation of singly- and doubly-linked lists. This implementation is not only simple, but has speedups over a common lock-free implementation of about 66 % and 6 % respectively.

To answer the question, if ASF had potential for accelerating software transactional memory, I adapted a modern high-performance STM implementation, TinySTM, to use ASF and achieved performance improvements up to 17 %.

During the experimental evaluation, the enhanced PTLsim has produced results within 10 - 20 % of those obtained on the native platform for benchmarks running on up to four cores.

### 6.2 Experience and Review

I think that the obtained results are satisfactory, although I had hoped for larger possible speed-ups of TinySTM. But given the amount of time that was spent on fixing, tuning and extending the simulator, and the already high performance of TinySTM, I am happy to have achieved this much.

Most impressive in my opinion is the simplicity of designing lock-free algorithms with ASF. Simplifying Harris’ CAS-based linked list with the help of ASF has been very straightforward and produced amazing results. More complicated structures, such as trees or hash-tables are likely to also benefit from the

increased flexibility offered by ASF's primitives.

Working with the simulator gave me a chance to take a closer look “under the hood” of an architecture similar to that of today’s microprocessors. I am amazed at how this complex functionality can be implemented using just a “fistful” of transistors. Unfortunately, the amount of time spent on the simulator was higher than anticipated. The false estimation was caused by wrong assumptions about the functionality and precision already provided in the simulator and the steep learning curve for understanding out-of-order effects.

## 6.3 Outlook

### 6.3.1 PTLsim

During my thesis I have only scratched the surface of many of the involved problems. A large amount of work is still needed inside the simulator: Cache hierarchies should be more flexible, so that caches can be shared between cores and for example connected in “exclusive” fashion. A generalisation of flexible connections between caches and cores is the idea of a proper interconnect network model, with selectable consistency and coherency protocols, bandwidth limitations and latencies. In fact such a model exists already: The “Opal” component of GEMS [MSB<sup>+</sup>05] provides such functionality. Making the interfaces of PTLsim and Opal compatible is an interesting thing to be looked at in the future.

Several improvements I made to PTLsim’s core have to be polished and evaluated, such as prefetchers and the improved scheduler for microoperations. Their correct operation and utility have to be shown for a larger variety of benchmarks.

PTLsim uses Xen to provide an initial separation between the stack of applications and kernel which are to be tested, and the native hardware. This separation helps PTLsim when switching between native (on Xen) execution and simulation in the core model. The drawback of this technique is that PTLsim is not a normal application but part of a hypervisor which complicates debugging. Modifying PTLsim such that it is for example based on QEMU [Bel05], would help because full-system simulation could then run inside a “normal” user-space application. “Native execution” would then mean execution in QEMU and simulation would execute in PTLsim “next to” QEMU, *i.e.*, inside a user-space application, too.

### 6.3.2 ASF

The static layout of critical sections in ASF makes the efficient use of the additional hardware primitives sometimes difficult. In particular for the purpose of accelerating STM, a more flexible setup would

enable more efficient optimisations. The `VALIDATE` instruction, for example, was easily integrated into the simulator and is the key element that actually enables the presented effective acceleration idea.

Several concepts for making ASF more flexible came up during the thesis, simulation of some of these concepts and comparing the results for different workloads would be helpful in deciding how this flexibility could be implemented natively.

Experiments with the STM acceleration have shown that the current implementation of ASF limits performance, because critical sections cannot be executed concurrently. Such serialisation of critical sections can cause bad exploitation of instruction-level parallelism for small, frequent critical sections. An idea to allow overlap of ASF critical sections that hardly increases complexity is to add identifiers to ASF-CS inside the core and use these to tag entries in the LLB. That way the LLB can accommodate multiple smaller sections and decide on inconsistent incoming probes which sections should be aborted.

Finally there is still a large amount of formal verification needed for ASF. Lamport's formalism and similar notions have sparked my interest in deriving formal relations for ASF primitives and out-of-order execution. Being able to prove implementation extensions correct might have saved several weeks of pipeline debugging caused by unexpected pipeline interactions.

### 6.3.3 STM

During the benchmarks of TinySTM on native hardware, I obtained preliminary results with performance peculiarities. These artifacts suggest that lock-based STMs may behave poorly when concurrent applications are executed simultaneously. With the original proposal of using ASF to remove indirections from non-blocking STMs, it would be interesting to analyse if ASF could help them to reach performance of the current locking ones. I speculate that the non-blocking implementations would be less prone to adversarial scheduling conditions.

In addition to the mechanisms I introduced in this thesis, a few other acceleration attempts for TinySTM were envisioned, but could not be tried out because of time constraints, for example using ASF also for stores in TinySTM. With the simulator working well now, these modifications can be implemented and evaluated easily.



---

## Bibliography

- [AAK<sup>+</sup>05] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA*, pages 316–327. IEEE Computer Society, 2005.
- [AG96] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.
- [AMD05] AMD Corp. *Software Optimization Guide for AMD64 Processors*, 3.06 edition, October 2005.
- [AMD07a] AMD Corp. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, 3.14 edition, September 2007.
- [AMD07b] AMD Corp. *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors*, 3.00 edition, September 2007.
- [AMD07c] AMD Corp. *Software Optimization Guide for AMD Family 10h Processors*, 3.05 edition, January 2007.
- [BC95] Jean-Loup Baer and Tien-Fu Chen. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.*, 44(5):609–623, 1995.
- [BDF<sup>+</sup>03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In Michael L. Scott and Larry L. Peterson, editors, *SOSP*, pages 164–177. ACM, 2003.
- [BDH<sup>+</sup>06] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.
- [Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46. USENIX, 2005.
- [BJ00] R. Bhargava and L. K. John. Issues in the design of store buffers in dynamically scheduled processors. In *ISPASS '00: Proceedings of the 2000 IEEE International Symposium on*

- Performance Analysis of Systems and Software*, pages 76–87, Washington, DC, USA, 2000. IEEE Computer Society.
- [BP04] James R. Bulpin and Ian A. Pratt. Multiprogramming performance of the pentium 4 with hyper-threading. In *Third Annual Workshop on Duplicating, Deconstructing and Debunking (WDDD2004)*, pages 53–62, 2004.
- [CH07] Pat Conway and Bill Hughes. The amd opteron northbridge architecture. *IEEE Micro*, 27(2):10–21, 2007.
- [CLL<sup>+</sup>07] Lawrence Crowl, Yossi Lev, Victor Luchangco, Mark Moir, and Dan Nussbaum. Integrating transactional memory into c++. In *Proceedings of The Second ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 07)*, Portland, Oregon, August 2007.
- [DFL<sup>+</sup>06] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. *SIGPLAN Not.*, 41(11):336–346, 2006.
- [DMMJ01] David Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. Lock-free reference counting. In *PODC*, pages 190–199, 2001.
- [DMSS07] Luke Dalessandro, Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Capabilities and limitations of library-based software transactional memory in C++. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*. ACM, Portland, OR, Aug 2007.
- [DSS06] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.
- [Enn06] Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, intel, 2006.
- [FFM<sup>+</sup>07] Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, and Heiko Sturzrehm. Transactifying applications using an open compiler framework. In *TRANSACT*, August 2007.
- [FFR08] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.
- [GAG<sup>+</sup>92] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, 1992.

- [GC96] Michael Greenwald and David R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *OSDI*, pages 123–136, 1996.
- [Goo89] J. R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherence Interface Working Group, 1989.
- [Har01] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. *Lecture Notes in Computer Science*, 2180:300–??, 2001.
- [Her90] M. Herlihy. A methodology for implementing highly concurrent data structures. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 197–206, New York, NY, USA, 1990. ACM.
- [HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In Ron Crocker and Guy L. Steele Jr., editors, *OOPSLA*, pages 388–402. ACM, 2003.
- [HLMS03] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, pages 92–101, July 2003.
- [HM93] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.
- [HP02] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 3rd Edition*. Morgan Kaufmann, 2002.
- [Int07] Intel Corp. *Intel® 64 Architecture Memory Ordering White Paper*, 1.0 edition, August 2007.
- [IS05] William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In Marcos Kawazoe Aguilera and James Aspnes, editors, *PODC*, pages 240–248. ACM, 2005.
- [Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *SIGARCH Comput. Archit. News*, 18(3a):364–373, 1990.
- [KCJ<sup>+</sup>06] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of Symposium on Principles and Practice of Parallel Programming*, March 2006.
- [KM03] D. Koufaty and D.T. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2):56–65, April 2003.

- [LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, mar 2004.
- [Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.
- [Lam86] Leslie Lamport. On interprocess communication. part i: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.
- [LMN07] Yosef Lev, Mark Moir, and Dan Nussbaum. Phtm: Phased transactional memory. In *Second ACM SIGPLAN Workshop on Transactional Computing*, 2007.
- [MBM<sup>+</sup>06] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. LogTM: log-based transactional memory. *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 254–265, 11-15 Feb. 2006.
- [MCE<sup>+</sup>02] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.
- [MHW02] Carl J. Mauer, Mark D. Hill, and David A. Wood. Full-system timing-first simulation. In *SIGMETRICS*, pages 108–116. ACM, 2002.
- [Moo98] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, jan 1998.
- [MS96] Larry W. McVoy and Carl Staelin. Imbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.
- [MSB<sup>+</sup>05] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.
- [MTC<sup>+</sup>07] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. *SIGARCH Comput. Archit. News*, 35(2):69–80, 2007.
- [PP84] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA '84: Proceedings of the 11th annual*

- international symposium on Computer architecture*, pages 348–354, New York, NY, USA, 1984. ACM.
- [RFF06] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In Shlomi Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 284–298. Springer, 2006.
- [RFF07] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based transactional memory with scalable time bases. In Phillip B. Gibbons and Christian Scheideler, editors, *SPAA*, pages 221–228. ACM, 2007.
- [RHL05] Ravi Rajwar, Maurice Herlihy, and Konrad K. Lai. Virtualizing transactional memory. In *ISCA*, pages 494–505. IEEE Computer Society, 2005.
- [SATJ06] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.
- [SPA92] SPARC International Inc. *The SPARC Architecture Manual - Version 8*. SPARC International Inc., 535 Middlefield Road, Suite 210, Menlo Park, CA 94025, 1992.
- [Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.
- [Val95] John D. Valois. Lock-free linked lists using compare-and-swap. In *Symposium on Principles of Distributed Computing*, pages 214–222, 1995.
- [WAFM07] Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Mechanisms for store-wait-free multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 266–277, New York, NY, USA, 2007. ACM.
- [YBM<sup>+</sup>07] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 261–272, Washington, DC, USA, 2007. IEEE Computer Society.
- [You07] M.T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 23–34, April 2007.

