Energy-Aware Synchronization Techniques
For Multicore Embedded Systems

by

Cesare Ferri

B.Sc., University Of Bologna; Bologna, Italy, 2005
M.Sc., Brown University; Providence, RI, 2008

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in The School of Engineering at Brown University

PROVIDENCE, RHODE ISLAND

May 2011
© Copyright 2011 by Cesare Ferri
This dissertation by Cesare Ferri is accepted in its present form by The School of Engineering as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date_________________________  ________________________________

Iris Bahar, Ph.D., Advisor

Date_________________________  ________________________________

Maurice Herlihy, Ph.D., Advisor

Recommended to the Graduate Council

Date_________________________  ________________________________

Tali Moeshet, Ph.D., Reader

Date_________________________  ________________________________

Sherief Reda, Ph.D., Reader

Date_________________________  ________________________________

Samantika Subramaniam, Ph.D., Reader

Approved by the Graduate Council

Date_________________________  ________________________________

Peter Weber, Dean of the Graduate School
Vitae

Cesare Ferri was born in Teramo, Italy in 1979. He received his B.Sc. in Electrical Engineering from the University of Bologna, Italy, in 2005, and his M.Sc. in Electrical and Computer Engineering from Brown University in 2008.

His research interests concern the exploration of multiprocessing techniques for low-power embedded systems, devising reliability-driven optimizations for memory architectures, and the development of design methods to improve the yield of 3D integrated circuits.

Cesare_Ferri@brown.edu
http://www.lems.brown.edu/cesare

Brown University, RI, USA
Acknowledgements

I would like to express my sincere gratitude to my advisors, Prof. Iris Bahar and Prof. Maurice Herlihy, for their patience, encouragement, continuous support and guidance during this work. Without them, this project would not have been possible.

I am also grateful to my committee members, Prof. Tali Moeshet, Prof. Sherief Reda and Dr. Samantika Subramaniam, for the their invaluable feedback and the effort they have put in reading this thesis manuscript.

I also would like to thank all my friends and colleagues of the computer engineering laboratory. Thanks to them my staying in Brown has been an even more pleasant experience. I would like to thank Desta “Mickey” Tadesse, Kundan Nepal, Nuno Alves, Elif Alpaslan, Yiwen Shi, Rotor Le, Kumud Nepal, Octavian Biris and Marco Donato.

Last but not least, I deeply thank my family. My wife Elejdis, for her relentless encouragement, my son Lorenzo for helping me relax from work, and my parents and brother, Donatella, Arnaldo and Flavio, for their unconditioned support.
Abstract of “Energy-Aware Synchronization Techniques For Multicore Embedded Systems” by Cesare Ferri, Ph.D., Brown University, May 2011

Roughly ninety percent of all microprocessors manufactured in a year are intended for embedded devices such as cameras, cell-phones, or machine controllers. Like servers and PCs, also embedded platforms are turning into multicore architectures, since this kind of design offers the best power/performance tradeoff. However, managing concurrency in an efficient manner, both in terms of performance and energy, is very important if these devices are to be useful.

In this thesis we address two different approaches to synchronization. First, we optimize the design of conservative multiprocessing techniques based on locks. We compare a set of classical implementations (i.e., based on busy waiting, or on sleep states) of mutex semaphores, and propose a hybrid (wait/sleep) semaphore in which the sleep state is entered only after a number of busy-wait cycles. We show that our hybrid scheme provides the best overall energy-delay product with respect to previously proposed schemes.

In the second part of the thesis we focus on a hardware accelerator based on transactional memory for handling concurrency within an embedded system. We consider energy consumption and complexity to be driving concerns in the design of the and therefore adapt simple Hardware Transactional Memory (HTM) schemes in our architectural design. We propose several different cache structures, data versioning policies and contention management schemes to support transactional memory and evaluate them in terms of energy, performance, and complexity.

We finally present SoC-TM, an integrated HW/SW solution for transactional programming on embedded MPSoCs. Our proposal leverages a HTM design, based
on a dedicated HW module for conflict management, whose functionality is exposed to the software through a low-level transactional application programming interface (API). To further improve ease of programming, our framework supports speculative task- and data-level parallelism. Our SoC-TM system can be programmed to commit transactions in a given order, and OpenMP worksharing constructs have been enhanced to optimistically run in parallel threads which may be inter-dependent. Our experimental results confirm that SoC-TM is a viable and cost-effective solution for embedded MPSoCs, in terms of energy, performance and productivity.
Contents

Vitae iv
Acknowledgments v

1 Introduction 1
  1.1 Multicore Embedded Systems .......................... 1
  1.2 Energy Efficient Lock Synchronization .................... 2
  1.3 Transactional Memory For Embedded Architectures .......... 5
    1.3.1 Energy and Throughput Efficient HTM Design .......... 7
    1.3.2 Supporting HTM at the Compiler Level .................. 9
  1.4 Thesis Contributions ..................................... 11
  1.5 Organization of the Thesis ................................ 12

2 Background and Previous Work 13
  2.1 Conservative Synchronization: Locking ...................... 13
    2.1.1 Spinlocks ......................................... 15
    2.1.2 Alternative Approaches .............................. 17
  2.2 Speculative Synchronization: Transactional Memory .......... 20
    2.2.1 Handling Transactional Events .......................... 21
    2.2.2 Transactional Programming Benefits ..................... 23
    2.2.3 STM vs. HTM for Embedded Systems ...................... 24
  2.3 Transactional Memory Design Space ........................ 25
    2.3.1 Data Version Management ................................ 26
    2.3.2 Conflict Management Policies .......................... 30
    2.3.3 Target Design Space .................................. 31
    2.3.4 Extensions to Compiler ............................... 33

3 Energy-Efficient Locking Mechanisms 36
  3.1 Energy-Optimal Synchronization Primitives ............... 37
    3.1.1 Multi-Core Reference Platform ........................ 37
    3.1.2 Frequency Regulation .................................. 39
    3.1.3 Software ............................................. 40
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.4</td>
<td>3.1.4</td>
<td>Energy-Optimal Sleep-Based Semaphores</td>
<td>40</td>
</tr>
<tr>
<td>3.2</td>
<td>3.2</td>
<td>Experimental Results</td>
<td>44</td>
</tr>
<tr>
<td>3.3</td>
<td>3.3</td>
<td>Summary and Discussion</td>
<td>50</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Transactional Memory for MPSoCs</td>
<td>52</td>
</tr>
<tr>
<td>4.1</td>
<td>4.1</td>
<td>Architecture</td>
<td>53</td>
</tr>
<tr>
<td>4.1.1</td>
<td>4.1.1</td>
<td>The Transactional Cache</td>
<td>56</td>
</tr>
<tr>
<td>4.1.2</td>
<td>4.1.2</td>
<td>Size Limitations</td>
<td>58</td>
</tr>
<tr>
<td>4.1.3</td>
<td>4.1.3</td>
<td>Reducing Abort Rates</td>
<td>60</td>
</tr>
<tr>
<td>4.2</td>
<td>4.2</td>
<td>Experimental Results</td>
<td>64</td>
</tr>
<tr>
<td>4.2.1</td>
<td>4.2.1</td>
<td>Software</td>
<td>64</td>
</tr>
<tr>
<td>4.2.2</td>
<td>4.2.2</td>
<td>Hardware</td>
<td>66</td>
</tr>
<tr>
<td>4.2.3</td>
<td>4.2.3</td>
<td>Evaluating Locking vs. HTM using a TC</td>
<td>70</td>
</tr>
<tr>
<td>4.2.4</td>
<td>4.2.4</td>
<td>Evaluating the TC Schemes vs. L1+VC Scheme</td>
<td>71</td>
</tr>
<tr>
<td>4.2.5</td>
<td>4.2.5</td>
<td>Evaluating Conflict Resolution Schemes</td>
<td>74</td>
</tr>
<tr>
<td>4.3</td>
<td>4.3</td>
<td>Summary and Discussion</td>
<td>77</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>Design Optimizations</td>
<td>80</td>
</tr>
<tr>
<td>5.1</td>
<td>5.1</td>
<td>Data Versioning: The Transactional-log</td>
<td>81</td>
</tr>
<tr>
<td>5.1.1</td>
<td>5.1.1</td>
<td>Logging - No Filter cache</td>
<td>82</td>
</tr>
<tr>
<td>5.1.2</td>
<td>5.1.2</td>
<td>Static Filter Cache</td>
<td>83</td>
</tr>
<tr>
<td>5.1.3</td>
<td>5.1.3</td>
<td>Dynamic Filter Cache</td>
<td>85</td>
</tr>
<tr>
<td>5.1.4</td>
<td>5.1.4</td>
<td>Experimental Results</td>
<td>87</td>
</tr>
<tr>
<td>5.2</td>
<td>5.2</td>
<td>Decoupling Conflict Detection from Caches</td>
<td>89</td>
</tr>
<tr>
<td>5.2.1</td>
<td>5.2.1</td>
<td>The Bloom Module</td>
<td>90</td>
</tr>
<tr>
<td>5.2.2</td>
<td>5.2.2</td>
<td>Flexible Conflict Management</td>
<td>94</td>
</tr>
<tr>
<td>5.2.3</td>
<td>5.2.3</td>
<td>Performance of the Bloom Module</td>
<td>95</td>
</tr>
<tr>
<td>5.3</td>
<td>5.3</td>
<td>Summary and Discussion</td>
<td>98</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>Integrating HTM with Compiler: The SoC-TM Approach</td>
<td>100</td>
</tr>
<tr>
<td>6.1</td>
<td>6.1</td>
<td>Motivation</td>
<td>101</td>
</tr>
<tr>
<td>6.2</td>
<td>6.2</td>
<td>System Overview</td>
<td>102</td>
</tr>
<tr>
<td>6.2.1</td>
<td>6.2.1</td>
<td>Architectural Template</td>
<td>102</td>
</tr>
<tr>
<td>6.3</td>
<td>6.3</td>
<td>Hardware TM Design</td>
<td>104</td>
</tr>
<tr>
<td>6.3.1</td>
<td>6.3.1</td>
<td>Hardware Support for Ordered Commits</td>
<td>106</td>
</tr>
<tr>
<td>6.4</td>
<td>6.4</td>
<td>Software TM Support</td>
<td>112</td>
</tr>
<tr>
<td>6.4.1</td>
<td>6.4.1</td>
<td>Extensions for Ordered Commits</td>
<td>116</td>
</tr>
<tr>
<td>6.5</td>
<td>6.5</td>
<td>Experimental Results for Speculative parallelization</td>
<td>120</td>
</tr>
<tr>
<td>6.6</td>
<td>6.6</td>
<td>Summary and Discussion</td>
<td>125</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>Conclusions</td>
<td>127</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Overview of the system configurations. ........................................ 66
5.1 Hardware configurations. ..................................................... 95
6.1 Low-level Transactional API (LLTA) ....................................... 111
# List of Figures

1.1 Impact of Synchronization on Performance and Power for a synthetic microbenchmark. ........................................... 3

2.1 Example of Critical Section. .......................................................... 14

2.2 Example of a \textit{spinlock} synchronization routine, built on top of the \texttt{TestAndSet} operation. ................................................ 14

2.3 The \textit{TTS-lock} ........................................................................ 15

2.4 The \textit{DelayAfterRead-lock} ...................................................... 15

2.5 The \textit{DelayAfterAccess-lock} .................................................... 16

2.6 The \textit{Queue-lock}, implemented on top of the \texttt{ReadAndIncrement} primitive. .......................................................... 17

2.7 Examples of Master-Slave thread communication. ...................... 19

2.8 The transactional cache maintains two copies for each line, and its data set is exclusive to that stored in the L1 cache. .............. 22

2.9 Transactional Memory Design Space ............................................ 26

2.10 Modifications to the cache controller to handle logging of transactional writes. The control flow in the gray area is new. ............ 28

3.1 Distributions of $T_{idle}$ periods for the \textbf{DVS-sleep} implementation. . 44

3.2 Total cycles, energy and EDP as \texttt{MAXSPINS} varies for the DVS-sleep system configuration. Negative percentages indicate improvement relative to \texttt{MAXSPINS}=10. ................................................................. 46

3.3 Total cycles and energy normalized to HW-locks when \texttt{MAXSPINS}=10 and switch overhead=0. Negative percentages indicate improvement. 47

3.4 Total cycles and energy normalized to HW-locks when \texttt{MAXSPINS}=50 and switch overhead=1000. Negative percentages indicate improvement. 48

3.5 EDP normalized to simple locks when \texttt{MAXSPINS}=10 and switch overhead=0. Negative percentages indicate energy-delay improvement relative to locks. ................................................................. 49

3.6 EDP normalized to simple locks when \texttt{MAXSPINS}=50 and switch overhead=1000. Negative percentages indicate energy-delay improvement relative to locks. ................................................................. 49

4.1 Example configuration with 4 CPUs. ............................................ 54

4.2 a) Architectural configuration to support hardware transactional memory. Note the transactional cache holds all transactional data. b) New architectural configuration to support hardware transactional memory using a victim cache (VC). Note that the primary storage structure for transactional data is now the L1 cache. In case of conflict evictions, transactional data can be held in the VC. ............................................ 56
6.4 Design of a circular buffer to support ordered commits of dynamically scheduled transactional workload ................................................................. 108
6.5 Design of per-core priority buffers to support ordered commits of statically scheduled transactional workload ...................................................... 110
6.6 False sharing due to misaligned data ................................................................. 115
6.7 Speculative loop parallelization with static scheduling ............................... 118
6.8 Speculative loop parallelization with dynamic scheduling .......................... 120
6.9 Results for synthetic benchmark ................................................................. 122
6.10 Results for speculative parallelization ....................................................... 124
Chapter 1

Introduction

1.1 Multicore Embedded Systems

High-end embedded systems are increasingly coming to resemble their general-purpose counterparts. Embedded systems such as smart phones, game consoles, “net-tops”, in-car entertainment (ICE) systems, and home multimedia centers are becoming ubiquitous. In the same way that smart phones are gradually usurping many of the functions of laptops, specialized high-end embedded systems will eventually displace many general-purpose systems. Eventually, such devices will affect every aspect of modern life, and their energy consumption profiles will have a broad economic impact.

Multi-core architectures have become pervasive in large-scale digital integrated systems. Manufacturers of general-purpose processors have essentially given up trying to increase clock speed, and instead are now focusing on multiprocessor-system-on-a-chip (MPSoC) architectures, in which multiple processors residing on a single
chip communicate directly through shared hardware caches, providing increased concurrency instead of increased clock speed [13]. Integrated platforms for embedded applications are even more aggressively pushing core-level parallelism. SoCs with tens of cores are commonplace [58, 71, 73], and platforms with hundreds of cores have been announced [59].

In principle, multi-core architectures have the advantages of increased power-performance scalability (assuming on-chip interconnect scales well [8]) and faster design cycle time (by exploiting replication of pre-designed components). However, performance and power benefits can be obtained only if applications exploit a high level of concurrency. Indeed, one of the toughest challenges to be addressed by multi-core architects is how to help programmers expose application parallelism.

Embedded applications in multimedia, imaging, and communication, have a high degree of exposed thread-level parallelism, and this is one of the main reasons for the rapid diffusion of multi-core architectures in embedded systems [20]. However, managing concurrency in order to guarantee correctness and consistency is not easy, and doing so in an energy efficient manner is even harder.

### 1.2 Energy Efficient Lock Synchronization

Although efficient software design for parallel systems has been widely studied for traditional multiprocessor architectures, the different performance impact of the micro-operations (e.g., cache misses) has not allowed the direct application of these techniques. For instance, in a multi-core architecture, where the main background memory is on-chip (be it an SRAM block or an embedded DRAM), the penalty of
a cache miss is often only a few cycles (as opposed to tens or hundred of cycles in a traditional architecture), making the dependency of performance on miss rate less critical. Furthermore, the appearance of energy as a relevant metric for the systems implemented by these multi-core architectures has ruled out many performance-oriented approaches that ended up being strongly energy-inefficient.

Besides these aspects, one often neglected feature of MPSoCs has to do with the class of applications they run (i.e., multimedia, filtering, encoding/decoding), which are characterized by a significant amount of data sharing. This data sharing results in complex synchronization patterns, and as a consequence the specific synchronization primitives employed can have a significant influence in determining the quality (performance and power) of the entire application.

Figure 1.1 quantitatively justifies the above claim; it shows the time and energy spent doing synchronization using conventional spinlocks (implemented by polling a variable stored in the shared address space) as a function of the number of processors.

![Figure 1.1: Impact of Synchronization on Performance and Power for a synthetic microbenchmark.](image)
The synthetic application generates various synchronization patterns, resulting into two levels of contention on the bus (label ‘high’ and ‘low’ in the figure). We notice that up to half of the time and energy consumption are due to synchronization.

Motivated by these results, the relevance of energy as a metric, together with the relative importance of synchronization, thus call for a light-weight and energy-frugal synchronization infrastructure, that possibly trades extra execution cycles for some additional energy efficiency.

When thinking of synchronization infrastructures for MPSoCs, we have to consider also that they often exhibit a very simple software architecture, sometimes even without the support of an operating system. This implies that synchronization cannot exploit high-level constructs such as monitors or message-passing primitives, and is rather done by conventional spinlocks.

Although several works have addressed the issue of light-weight synchronization primitives in chip multi-processor architectures ([40]–[25]), only a few have explicitly focused on their energy efficiency ([40, 27, 83]). The work of [27], in particular, provides an explorative analysis of the energy efficiency of various standard implementations of spinlocks, and proposes a sleep-based solution in which busy-wait is replaced with a low-power sleep state. The results support in part the intuition that sleeping is better, energy-wise, than busy-waiting, although the best choice of spinlock implementation depends on the access patterns to the critical section.

The analysis carried out in [27], however, did not consider several relevant issues. First, the MPSoC target architecture did not support frequency and/or voltage scaling; this additional dimension considerably complicates the analysis, since the energy cost of busy-waiting depends on the frequency. Second, the energy consumption in
the sleep state was assumed to be zero, assuming no leakage power is consumed. Third, only the most intuitive implementation of a sleep-based spinlock was presented, i.e., one where waiting is totally replaced by sleeping.

In this thesis, we present an explorative study of sleep-based spinlocks in a more realistic setting in which the multi-core architecture supports voltage/frequency scaling, and in which leakage is assumed to be non-negligible. The exploration analyzes two main low-level parameters: the state-transition overhead (sleep to active), and, more importantly, the number of sleep cycles. The latter dimension implies that we implicitly introduce a hybrid (wait/sleep) semaphore in which the sleep state is entered only after a number of busy-wait cycles. In other words, we use a timeout-based version of a spinlock.

We present simulation results on a set of benchmarks that clearly show that the timeout-based spinlock provides a solution that is more sensitive to the synchronization patterns of the application, providing overall better energy-delay savings, compared to scenarios in which DVFS or pure busy-waiting are applied.

1.3 Transactional Memory For Embedded Architectures

Although its use is pervasive, locking, whether employed in general-purpose or embedded systems, suffers from well-known limitations. To achieve high performance, fine-grained locking techniques may be used to minimize serialization of core activities, but this requires deep understanding of the target application and complicated debug processes. Deadlocks are in fact difficult to avoid, especially when systems
have multiple resources. Moreover, even when the system is operating correctly, conditions like lock spinning may impose a significant overhead in power and performance, since they tend to flood the interconnect with useless transactions. Coarse-grained locking is easier to use, but can not extract high degrees of parallelism.

In reaction to these problems, recent approaches for general-purpose systems have focused on Transactional Memory (TM) for handling concurrent accesses to shared memory [33]. The main idea behind transactional memory is to allow shared memory accesses to proceed optimistically, without requiring a processor to explicitly wait and obtain exclusive ownership of the data structure within the shared memory space first. If a conflict occurs (i.e., more than one active transaction tries to modify the same line in shared memory), some means of rollback and re-execution is required to retain correctness and consistency. While current research on transactional memory is extensive, it has focused predominantly on general purpose processors. Its applicability to embedded multi-core platforms, based on simple cores and memory system interfaces, has yet to be explored. Natural questions are then, “Is transactional memory an attractive alternative to locks for embedded systems? Is it energy-efficient?”

Here, we investigate how transactional memory can be adapted for embedded systems. Transactional memory for embedded systems makes different demands than transactional memory for general-purpose systems. The principal difference is the central importance of energy consumption: although embedded systems are becoming more sophisticated, they are and will continue to be energy-constrained, either because they run on batteries, or simply because energy consumption is increasingly a concern for systems at all levels. We are the first to use energy consumption as a guide for designing transactional memory mechanisms. While most earlier work on transactional memory has neglected the question of energy consumption, we claim
that it should be one of the driving concerns in transactional memory design. Taking energy into account requires revisiting and revising many widely-accepted assumptions, as well as widely-accepted architectures.

The need for energy-efficiency and simplicity makes software transactional memory (STM) unattractive. (Klein et al. [38] provide an analysis of the energy costs of a typical STM system.) For most embedded applications, it is unacceptable, both in terms of performance and energy consumption, to place a software “barrier” at each memory access. Indeed, embedded applications often run without an operating system. By contrast, we will see that a simple hardware transactional memory (HTM) can both enhance performance and conserve energy.

1.3.1 Energy and Throughput Efficient HTM Design

In this thesis, we evaluate HTM designs using three criteria: energy, performance, and complexity. Sometimes these criteria reinforce one another, and sometimes not. We investigate a sequence of HTM designs, starting from a simple baseline, and moving on to a sequence of redesigns, each intended to address a specific problem limiting energy efficiency and performance.

We first present Embedded-TM, a HTM that leverages the underlying cache coherency system. We investigate how a variety of cache designs affects the performance and energy consumption of a multicore embedded system that supports HTM. Prior work on HTM has focused on a simple cache architecture [24, 33] in which non-transactional data was stored in a large direct-mapped cache, and a smaller, fully associative transactional cache was used to store all data accessed within a transaction. This architecture has drawbacks. Namely, since the transactional cache is
the only place to store transactional data, any transaction that exceeds the size of this cache will overflow, forcing transactions to serialize, even if no data conflicts exist between them. So, although a small transactional cache is desirable for energy purposes, making it too small could hurt throughput significantly.

Here, we consider an alternative design suitable for embedded systems in which both caches are unified into a single L1 cache holding both transactional and non-transactional entries.

As often happens, alleviating one problem exposes another. Transactions can also be prevented from making progress by data conflicts that occur when two transactions access the same memory location, and at least one access is a write. The first approach we consider, called eager conflict resolution, works well when transactions have few data conflicts, but less well when transactions have many data conflicts.

We examine two approaches to the problem of high-conflict transactions: a brute-force approach where a transaction that fails to make progress is eventually restarted in serial mode, and a more complicated approach where conflicts are resolved in a “lazy” manner. Lazy conflict resolution postpones the decision on aborting transactions to a later time, when more data on detected conflicts is available, thus potentially increasing concurrency.

Each approach is successful in some circumstances. The brute-force approach is attractive for its simplicity and wide range of effectiveness, and it works moderately well most of the time. The lazy mode algorithm incurs a higher overhead cost, sometimes penalizing low-conflict transactions, but is effective for high-conflict transactions.
We also present a design optimization targeting the data versioning management of our TM system. We propose a “Transactional-log” to record transactional accesses to private memory. The goal of such technique is to reduce the number of transactional entries to store in the cache. We expect gains in throughput because of the fewer overflows. Note that our system is different from other log-based TM proposals [52, 81]. Our approach, in fact, does not incur in any performance penalty in case of abort, since the undo-log contains only thread-local data. Unlike other approaches, in our system transactions do not need to wait for the aborted transaction to restore the state of the shared memory. As further optimization, we extend our Transactional-log with a “Filter Cache” design. The Filter cache contains the addresses of the private data that have already been logged. If an address is already present in the Filter Cache, then current memory access can skip being logged. We show that even a small Filter Cache can improve the system performance because of the reduced number of logging operations required.

1.3.2 Supporting HTM at the Compiler Level

To further simplify the hardware subsystem and improve the flexibility of our design, we then introduce SoC-TM, an integrated HW/SW solution for transactional programming on embedded MPSoCs. At the heart of our proposal sits a HTM design that is decoupled from the cache coherency. An external module (i.e., the Bloom module), which is composed of a collection of Bloom filters [9], is responsible for keeping track of the history of each transaction and managing eventual conflicts. While other works have also proposed the use of Bloom filters as a means of managing conflicts among transactions (e.g., [67]), what we propose is a centralized unit that is fully implemented in hardware. This approach has the advantage of
being very lightweight and fast, requiring minimal changes to the cache coherency protocol logic, and removing the need for each core to make individual decisions on transaction conflict management. As with [67], it also allows for more flexibility in determining how to manage conflicts.

Ease of use and transparency of the internal functionality are first-class design constraints of the SoC-TM system. As such, application developers are not meant to cope directly with low-level transactional programming. Instead, in our system transactional features are triggered through the use of a custom set of compiler directives, implemented as an extension to the popular OpenMP programming model. Our enhanced compiler transforms the annotated program so as to interact with our runtime system, where low-level APIs are transparently used. To further improve ease of programming, our framework supports speculative task- and data-level parallelism. Specifically, loops which may carry cross-iteration dependencies (non-DOALL) can be annotated as parallel loops. Similarly, tasks which may possibly be inter-dependent can be annotated for parallel execution. The underlying TM system ensures that, in case a real dependence arises, the original sequential program semantics is preserved. In our SoC-TM system this can be achieved by forcing transactions to commit in program order, thanks to specific hardware support for prioritized commit that we provide. At the program level, this feature can be leveraged by the OpenMP worksharing constructs (parallel sections and parallel for), to generate parallel transactions holding speculative workloads (tasks or loop iterations). The compiler properly generates code that programs the Bloom module to commit speculative transactions in order.
1.4 Thesis Contributions

To summarize, this thesis makes the following contributions:

• We compare a set of classical implementations (i.e., based on busy waiting, or on sleep states) of mutex semaphores, and propose a hybrid (wait/sleep) semaphore in which the sleep state is entered only after a number of busy-wait cycles. The proposed scheme provides the best overall energy-delay product with respect to previously proposed schemes. Furthermore, we identify an optimal length of the busy-wait cycles, which is empirically shown to depend on the time required to switch from the sleep to the active state.

• We propose *Embedded-TM*, a new design for an energy-efficient hardware transactional memory (HTM) system for power-aware embedded devices. We propose several different cache structures and contention management schemes to support HTM and evaluate them in terms of energy, performance, and complexity.

• We present an alternative design of data versioning management for Embedded-TM. We propose a “Transactional-log” scheme to record transactional accesses to private memory. The goal of such a technique is to reduce the number of transactional entries stored in the cache, allowing the system to handle larger transactions and thereby causing fewer overflows. We extend our Transactional-log with a “Filter Cache” design, which improves the performance by eliminating many redundant log updates.

• We further simplify the design by introducing *SoC-TM*, a HTM architecture that is decoupled from cache coherency. We propose a signature-based centralized module (i.e., the *Bloom module*), providing lightweight support to conflict
resolution and ordered commits. We also present a low-level programming API for conveniently programming it.

- We transparently integrate \textit{SoC-TM} within the OpenMP programming model, extended with language features for transactional programming. We also provide extensions to the programming API, supporting speculative parallelization of non-DOALL loops and dependent task graphs.

1.5 Organization of the Thesis

This thesis is organized as follows. Chapter 2 provides background and related work on multiprocessor synchronization. Several multiprocessing techniques, based on locking and transactional memory are presented. In Chapter 3 we introduce our hybrid (wait/sleep) semaphore design, and show the energy/performance benefits of such approach. In Chapter 4 we describe \textit{Embedded-TM}, our HTM architecture for embedded systems. We explore different design choices and optimizations to meet the energy and complexity requirements of an embedded platform. Two design optimizations schemes targeting data versioning and conflict detections – namely the “Transactional-log” and the “Bloom module” – are presented in Chapter 5. In Chapter 6 we will introduce \textit{SoC-TM}, our complete HW/SW framework for transactional programming for MPSoCs. Finally, Chapter 7 presents our conclusions and future directions.
Chapter 2

Background and Previous Work

In this Chapter we will review various synchronization techniques that have been proposed in literature. We will also address specific optimizations for embedded systems. We will focus on shared-memory Multiprocessors-on-a-chip (MPSoCs), since they have the greatest degree of acceptance in real systems. In the first part we will describe some conservative approaches based on locking mechanisms. In the second part, instead, we will introduce a more advanced type of synchronization method, which is based on the concept of transactions.

2.1 Conservative Synchronization:

Locking

The memory model of a MPSoC defines the responsibilities and challenges of the programmer with respect to management of data. For distributed memory MP-
Figure 2.1: Example of Critical Section.

```
Lock(lck)
  critical section code
Unlock(lck)
```

Figure 2.2: Example of a spinlock synchronization routine, built on top of the TestAndSet operation.

```
Lock(int *lock) {
    while (TestAndSet(lock) == TRUE) ;
}
Unlock(int *lock) {
    *lock = FALSE ;
}
```

SoCs [18], where each core features private tightly coupled storage, programmers are responsible for orchestrating efficient data movement before computation can start. The shared memory paradigm is widely adopted in embedded MPSoC designs, since it provides an easy-to-understand abstraction of memory resources: a single address space, to which programmers are accustomed. Memory inconsistency due to concurrent operations on shared memory is avoided by adopting the concept of critical section. A critical section is a portion of code that is executed by at most one thread at a time, as shown in Figure 2.1. To support mutual-exclusion, the system must be extended with some hardware support, generally exposed by means of special instructions (abstractly termed TestAndSet), which atomically read and write data on a memory location (e.g., the CMPXCHG on the IA32 architecture).

Using such instructions, the basic synchronization routine consists of a loop (see Figure 2.2), where a task inquires a shared variable to obtain the ownership of the lock. The lock releasing is performed by simply restoring the shared variable to a “free” state. Next we will survey classical implementations of such synchronization
routines.

### 2.1.1 Spinlocks

The mechanism based on the spin-wait loop is usually referred in literature as *spin-lock* [3], and requires a bus access for each **TestAndSet** operation. This may cause the overloading of the bus, whose bandwidth may be totally saturated by the synchronization traffic, even with a small number of cores. In order to mitigate this problem, local copies of the lock might be stored into the cores’ private caches, which must therefore be kept coherent. In this case, many external accesses can be avoided by performing a preliminary test with a simple read (which is resolved in cache). Figure 2.3 shows a spinlock implementation based on such method (usually called as **TestAndTestAndSet-lock**, or TTS-lock).

**Figure 2.3:** The *TTS-lock*

```c
Lock(int *lock) {
    while (*lock==TRUE || TestAndSet(lock) == TRUE);
}
```

The **TestAndSet** (that still requires a bus access since other cores may need to update their local copies of the lock) is used only when the lock is found to be free. In such a way, the bus is accessed only when the actual lock acquisition is probed. Notice, however, that the bus may still be flooded with line refilling requests after the lock has been updated.

As a way to further reduce the traffic, a delay can be introduced between accesses to the synchronization variable. The delay can be introduced either the lock has been released (i.e., after each read), or after a **TestAndSet** bus access., as shown in
Figure 2.4: The *DelayAfterRead-lock*

```
Lock(int *lock) {
    while (*lock==TRUE || TestAndSet(lock) == TRUE);{
        while (*lock==TRUE);
        delay();
    }
}
```

Figure 2.5: The *DelayAfterAccess-lock*

```
Lock(int *lock) {
    while (*lock==TRUE || TestAndSet(lock) == TRUE);
    delay();
}
```

Figure 2.4 and Figure 2.5.

From an energy prospective, however, inserting a delay in the spin-wait loop can be counterproductive, as shown in [27]. In fact, the key feature of such mechanisms is to lower down the congestion of the bus by reducing the responsiveness of the system. On average, the cores will wait longer, with the overall effect of increasing the energy consumption. To alleviate this problem, the cores should perform the delay operation in a low-power state.

A different way to reduce bus accesses relies on more complex data structures, based on queues, provided that a particular version of the atomic read-write instruction (*ReadAndIncrement*) is available [3]. Figure 2.6 shows a possible realization of a spinlock based on queues. The goal of this approach is to filter out as many bus accesses as possible. Note that the variables *flags* and *IDX* are both shared. The overall effect of the ReadAndIncrement instruction on the IDX shared variable is to force each core to spin on a different slot of the queue. When leaving the lock, the core will notify only the successor by writing on its slot. All the spinning operations
Figure 2.6: The Queue-lock, implemented on top of the ReadAndIncrement primitive.

Init() {
    IDX = 0;
    flags[0] = OWNER;
    flags[1..P-1] = NOT.Owner;
}

Lock_Q() {
    my = ReadAndincrement(IDX);
    while flags[my % P] != OWNER;
}

Unlock_Q() {
    flags[my % P] = NOT.Owner;
    flags[(my+1) % P] = OWNER;
}

are internal to the caches. No bus accesses are required except those for refilling the cache lines. A further optimization could be mapping each slot in a different cache line (thus reducing the number of line invalidations). This approach can also be supported by special hardware devices aimed at managing the synchronization locally whenever possible [51].

2.1.2 Alternative Approaches

Other approaches use additional hardware to implements the queued-lock semantics in a distributed way: a special device is coupled to each core in order to keep track of requests and releases of locks [83]. A totally different approach statically encodes the semantically correct order of accesses to each shared variable but requires hardware support associated with each processor and the preliminary extraction of the dependency patterns from the application [80]. This method however is limited by the fact that both the CPU instructions and the SRAM blocks must be extended with two extra bits to encode the Read-after-Write and the Write-after-Read access
The authors of [75] proposed new techniques for efficient lock-based synchronization in FPGA-based multiprocessor system-on-chips (MPSocs) for real-time applications. They propose cache-like structures to contain the current state (e.g., free or busy) of each lock and barrier. In [39] the real-time performance of embedded Linux was improved by monitoring the lock hold times. In [84] the authors introduced energy-efficient producer-consumer communication by compiler-inserted write-through store insertions to update a cached memory location before exiting a synchronization region.

Others have investigated lock-free synchronization for embedded systems. The authors of [17] considered the benefits of lock-free synchronization for the multi-writer/multi-reader problem in embedded real-time systems. Other work showed how to exploit access pattern regularity in a single producer/single consumer synchronization pattern [80]. Their light-weight synchronization mechanism encodes dependence information within each memory access.

More structured synchronization schemes are based on barriers (eventually built on the top of lower level spin-locks) that are implemented through shared structures. Such shared data allow the architecture to keep track of the arrival of processors at the synchronization point so that each core waits for the arrival of the last one. A typical application for such construct is found in the master-slave communication model (see Figure 2.7). The computation usually proceeds in phases: in the “serial” phase only one core (i.e., the master) executes the code. In the “parallel” phase, the computation is split among a group of slave threads. At the end of the parallel phase, the master thread will collect all the partial results from the slaves.
In order to reduce the data sharing, and thus to improve the scalability of the barrier implementation, a tree structure can be used [48]. The “constrained” behavior of barriers can be exploited to implement an energy-efficient version. The *thrifty barrier* [40] achieves energy reduction by forcing waiting cores into a low-power state from which they are resumed when the barrier is released (the release event is also speculatively predicted using the past behavior of the cores involved in the acquisition of the barrier). Alternatively, the time spent waiting for a barrier can be used to decide the frequency setting of cores through a hardware/software system, rather than moving into a low-power state [41].

Such an approach actually can be extended by taking into account the time spent in any synchronization event. This time allows to estimate the workload of each processor and, thus, to set its frequency accordingly. A simple hardware device can perform such a job, provided that the lower level of the synchronization is implemented with the polling of a shared variable [44].
However, things become less clear when both a low-power state and frequency regulation are available. On one hand, the frequency regulation decreases the time spent waiting for a semaphore (since it tends to equalize the workloads of the cores); on the other hand, exploiting the low-power state decreases the penalty due to the waiting. Transition overheads and the increasing impact of the leakage (that decreases the benefit of the low power state) introduce additional complexity to the management of the synchronization. Our work tries to shed some light on the intricate interactions between a dynamic voltage scaling (DVS) policy, driven by the synchronization time, and the availability of a low-power state that cores can enter when idle. We propose a novel hybrid (i.e., a combination of busy waiting and sleep-based semaphore) synchronization primitive in which the sleep state is entered only after a number of busy-wait cycles. We will describe this technique in Chapter 3.

2.2 Speculative Synchronization:

Transactional Memory

Transactional memory (i.e., TM) has been extensively investigated as an alternative means of synchronization in general-purpose systems. The principle behind the transactional memory working model is simple: each critical section (referred in a TM system as transaction) is speculatively executed by the CPU, and, if no conflicts with another transaction are detected, its effects become permanent (that is, the transaction commits). Otherwise, if conflicts are detected, its effects are discarded (that is, the transaction aborts), and the transaction is restarted.

Until a transaction commits, its effects are not visible outside the transaction itself. To ensure such isolation, hardware transactional memory keeps tentative
updates in a thread-local cache. If the transaction commits, these modified entries may be written back to memory. Data conflicts with other transactions will be detected by the native cache coherence mechanism. If a conflict is detected, the transaction is aborted, and its effects are discarded.

Transactional memory requires a checkpointing mechanism to support roll-back and re-issue of transactions in case of a conflict. Various checkpointing mechanisms have been proposed for recovering processor state. Many of these techniques periodically create a system-wide logical checkpoint of the system, which includes the state of the processor registers, memory values, and coherence permissions [70, 55]. For our purposes, it is sufficient to checkpoint the local registers of the processor that started the transaction.

### 2.2.1 Handling Transactional Events

During the execution of a the transaction, two copies of the accessed data will be stored in the TC, as shown in Figure 2.8. Of these two physical lines maintained for each address, one stores a backup copy of the data, while the other contains data modified during the transaction. This scheme requires adding two extra bits to the cache coherence tag vector, allowing the cache to distinguish between the backup copy (marked with the `T_COMMIT` bit) and the modified data (marked with the `T_ABORT` bit). If neither bit is set, this indicates that the data in the line is not contained within a transaction and therefore can be managed in the usual manner by the snoop device. Note that data contained in the TC and L1 caches are mutually exclusive. This implies that, if during a transaction data is found in the L1 cache, it must be invalidated in the L1 cache first before placing the duplicate copies in the TC. A cache line with its `T_ABORT` bit set can be modified multiple times within
Figure 2.8: The transactional cache maintains two copies for each line, and its data set is exclusive to that stored in the L1 cache.

A single transaction. The process of writing back data from the L1 to the TC does not incur extra latency, since the data is returned to the CPU right away, and no stall is required.

In case of a data conflict, the snoop device notifies the CPU with the \textit{Abort\_Transaction} signal, causing all speculatively-updated lines to be invalidated. When it receives the \textit{Abort\_Transaction} signal, the CPU takes the following action:

1. halts execution of the transaction,

2. marks the modified lines in its TC as invalid and restores the backup copies by resetting their \textit{T\_COMMIT} bits,
3. restores the original register values by reading from the SPM, and
4. changes its status to a “low power” mode.

The range of the backoff period is tuned according to the conflict rate. The first
time a transaction conflicts, it waits an initial random period (< 100 cycles) before
restarting. If the transaction conflicts again, the backoff period is doubled, and will
continue to double each time until the transaction completes successfully.

If there is no data conflict and the transaction commits successfully, the trans-
actional cache invalidates the backup copies of the data (i.e., the ones with the
\texttt{T\_COMMIT} bits set). The modified data remain in the TC, and their associated
\texttt{T\_ABORT} bits are reset. These steps result in data lines that were modified within
a transaction to now become visible to the rest of the system. As a consequence of
this commit operation, the snoop device will now manage these lines according to
the native cache coherence protocol. That is, these lines can be read, modified, or
invalidated similar to lines in other caches.

\subsection{2.2.2 Transactional Programming Benefits}

Two are the major benefits introduced by a TM system:

1. \textit{Ease of programming}. TM guarantees transactional threads to run in isolation,
   and that is the key to eliminate all possible sources of race condition.

2. \textit{Increased performance}. Hardware transactional memory provides a level of
   concurrency at least equivalent to the finest granularity locking.
Transactional memory also has an advantage over locks in terms of energy consumption [54], due to reduced contention and the absence of lock acquisition overhead.

### 2.2.3 STM vs. HTM for Embedded Systems

Transactional memory can be implemented in hardware (e.g., [33, 52, 30]), in software (e.g., [66, 32]), or via hybrid mechanisms that combine hardware and software (e.g., [19, 68]). Software TM (i.e., STM) offers more flexibility, since it permits the implementation of complex algorithms that would be impractical to implement in hardware. Furthermore, STM allows natural “virtualization” of transactions, since transactions are not bounded either in time (e.g., a transactional thread can be safely swapped in case of context switch) or to hardware components like caches. However, software and hybrid solutions usually do not offer strong-isolation guarantee (i.e., transactional code is not protected by non-transactional accesses), which is a critical requirement to allow compatibility with legacy (i.e., non-transactional) code found in standard libraries. Also, STM and hybrid solutions are much slower than HTM schemes [50], since they require software barriers for bookkeeping operations for each read and write to memory. Software TM is also energy inefficient. The authors in [38] shows that, in an embedded platform, a state-of-the-art STM implementation (i.e., TL2 [21]), consumes in average more than twice the energy of a locking counterpart. Because of these energy and performance reasons, we strongly believe that TM for embedded architectures should be implemented in hardware.

While hardware transactional memory lowers the overhead associated to software transactional memory, limitations on cache size and associativity bound transactions’ sizes and durations. While proposals exist for “unbounded” transactional memory [2, 63] that allow transactions to survive certain kinds of resource exhaus-
tion (i.e., context switches and overflows), these schemes are much too complex to be considered for embedded systems. Virtualizing a transaction would in fact require important hardware and software changes to the virtual memory subsystem. Since virtual memory affects complexity (e.g., a Memory Management Unit is required for fast translation), performance (e.g., added extra translation stage for each memory access) and responsiveness (e.g., handling a page fault) of a system, many embedded applications opted to run without virtual memory support. Embedded applications usually run on top of a minimal software layer, consisting of a set of specific low-level libraries.

For most embedded systems, applications’ resource requirements are well-understood, and transactions that exceed those expectations are likely to be rare. Nevertheless, it is important to understand how to structure caches for HTM in embedded systems to maximize transaction sizes without compromising performance or increasing energy consumption. In Chapter 4 we will describe several such designs.

### 2.3 Transactional Memory Design Space

No matter which implementation (i.e., hardware, software or hybrid), Transactional Memory systems can always be categorized according to three key attributes: \textit{when} they update memory (i.e., data versioning), \textit{when} they detect conflicts (i.e., conflict detection), \textit{how} resolve conflicts (i.e., conflict resolution). Figure 2.9 shows the design space of a typical hardware transactional memory architecture. Each attribute is now discussed in more detail.
2.3.1 Data Version Management

Transactional Memories can update memory “in-place” (i.e., eager conflict detection) during the transaction. In that case, the TM system must guarantee a rollback mechanism (usually implemented by means of log structures) to restore the original content of the memory. Log-TM [52] and its variants (e.g., LogTM-SE [81], TokenTM [11]) are examples of TM architectures that fall into this category. They offer very efficient commit support – since no broadcast of data is required – but a rather complex roll-back mechanism in case of abort (i.e., the aborted transaction must walk the undo-log, causing other transactions to stall). Another disadvantage of a ‘eager-update’ system is the overhead introduced to update the log for each write operation.
The logging operations are transparent to the software. Some extra hardware
circuitry in the cache controller (i.e., the log-controller) is in charge to monitor all
CPU memory write operations. As shown in Figure 2.10, for each transactional
write, the log-controller has to 1) fetch the original value from memory (stalling the
write request), and 2) save the read value into the undo-log. Note that the original
write request and the write in the undo-log can be done in parallel only after the
log-controller has read the original value from memory (either from the cache in case
of hit, or from the external memory in case of miss).

To filter out unnecessary log updates, Log-TM [52] proposes an extra “W” state
bit in the cache. If the W bit is set, then the memory location has already been
logged, and no further actions are required. However, having only one W bit per line
introduces another performance problem: an entire cache line must be logged for each
write. In case a line contains 8 words (which is a typical value), that would require
16 writes to the log (i.e., 8 writes to for the addresses and 8 for the data). Since
reads are more frequent than writes (i.e., subsequent memory locations are more
likely to be read and not written), this choice could negatively impact performance.
In Chapter 5, we will describe a technique that allows tracking single words, thus
providing all the filtering benefits of the W bit, without incurring in the overhead of
saving the entire cache line.

The major benefit of log-based architectures is the fact they are not bounded
to a particular transaction size, since the entire memory hierarchy can be exploited
by the transaction. However, pure logging solutions (e.g., Log-TM [52]) still need
caches to track down read and write transactional sets. In case a transactional line
must be evicted, in fact, a per-core overflow bit (i.e., the “O” bit) must be set in
the directory line. New coherency signals (i.e., the “NACK” signal) are required
to handle overflowing lines, thus complicating cache coherency and penalizing the
Figure 2.10: Modifications to the cache controller to handle logging of transactional writes. The control flow in the gray area is new.

system performance (since each external request to an overflowed line requires the owner of the line to walk its own overflow buffer). Signature-based logging schemes (e.g., LogTM-SE [81]), instead, avoid all problems related to line evictions by storing transactional read/write sets in per-core signatures. Other logging approaches (like TokenTM [11]) require adding extra physical bits (i.e., “metadata” bits) for each word in DRAM. Since modifying commodity resources – such as DRAM modules – is both unrealistic and unpractical, we opted for a design that did not require any change to such components.

With a “deferred-update” (or lazy-update) policy, cores update local copies of the shared data, and propagate the modifications only at commit time. This type of policy leverages caches to store local copies of the shared data, and cache coherency
to propagate the read and write transactional sets to the other cores at commit time. Unlike the “in-place” scheme, restoring an aborted transaction is quite fast, since the transactional copies can be invalidated locally. However, committing a transaction could potentially be slow, in case the adopted protocol would require broadcasting the transactional entries to the other cores [31].

To reduce serialization at commit time, newer architectures (e.g., BulkTM [15]) propose broadcasting signatures instead of the entire transactional sets. A signature summarizes in a compact way the addresses touched during a transaction. However, broadcasting big signatures (which are required to avoid many false conflicts) may require multiple bus transactions (e.g., broadcasting two R/W signatures of 4KBits each would require 128 bus transactions on a 64bit bus). Hence, short transactions would be greatly penalized.

Other ‘lazy-update’ schemes (e.g., Helihy&Moss [33], VTM [63], LTM [2]) handle efficiently also the commit phase. They leverage the cache coherency system to keep the data ‘naturally’ consistent at the end of the transaction. The major drawback of such architectures is however that they are bounded to the cache size. If the transaction does not fit into the cache, then some overflowing mechanism is required.

The original HTM proposal [33] implements transactional memory by means of a transactional cache (TC), an extra fully-associative cache that is accessed in parallel to the L1 cache. We will see in Chapter 4 how such a design is particulary power-inefficient when applied to an embedded context. Also, adding another cache structure complicates the design of the cache controller, since new inter-cache line operations (i.e., moving a line from L1 to TC) must be performed in hardware.
2.3.2 Conflict Management Policies

Transactional memory architectures differ also for the type of conflict management. Conflict management tackles two separate problems:

- Detecting Conflicts
- Resolving Conflicts

As for data versioning, conflicts can be detected at access time (i.e., eager conflict detection) during the execution of the transaction, or deferred at commit time (i.e., lazy conflict detection). With an eager approach, the system has less wasted work – since the conflicting transaction is stopped when the conflict is detected – at the expenses of no progress-guarantee (i.e., restarted transactions may abort committing transactions).

Note that, by construction, “lazy” conflict detection is incompatible with “eager” resolution. Also, a scheme with “eager” versioning and “lazy” conflict detections is not practical in a HTM architecture, since transactions could potentially violate each other’s transactional sets, and still have to wait until the commit phase to abort. The overall effect is to increase the likelihood of starvation.

A deferred conflict detection scheme, instead, features a progress guarantee (since at least one transaction has to commit), at the risk of wasted cycles (because all the transactions continue executing until commit time). This is an issue also for power, since the system may consume energy for carrying out meaningless work.

It has been shown that resolving conflicts at commit time has advantages in terms of throughput [46]. Both FlexTM [68] and EazyTM [74] leverage cache coherency to
detect conflict at access time, and exploit special hardware and software structures to resolve conflicts at commit time. Both architectures require important changes to the CPU (i.e., new instructions), to the cache coherency protocol (i.e., new states), and to the bus interconnect (i.e., new core-to-core wires). In Chapter 4 we will describe a lazy conflict resolution scheme that tries to minimize the hardware requirements without penalizing the performance.

2.3.3 Target Design Space

We evaluate HTM designs using three criteria: energy, performance, and complexity. Sometimes these criteria reinforce one another, and sometimes not. Here, we investigate a sequence of HTM designs, starting from a simple baseline, and moving on to a sequence of redesigns, each intended to address a specific problem limiting energy efficiency and performance. Structuring the presentation of our HTM architecture (i.e., Embedded-TM) as a sequence of redesigns makes it possible to quantify the contribution of each incremental improvement.

We take as the baseline Embedded-TM an HTM based on a simple cache architecture \([24, 33]\) in which non-transactional data is stored in a large L1 cache, and a smaller, fully-associative transactional cache stores the data accessed within a transaction. The principal drawback of this architecture is that the transactional cache consumes too much energy (i.e., more than 30% of the system energy, as we will show in Chapter 4). As a first line of defense, we consider how to conserve energy by powering down the cache without adversely effecting performance.

Another drawback of the baseline Embedded-TM architecture is the limited size of the transactional cache. Any transaction whose data set cannot fit in that cache
cannot complete, and must continue in a less-efficient serial mode described below. To alleviate this problem, we consider an alternative design in which both transactional and non-transactional data are kept together in the L1 cache (as proposed also by [2, 63, 31]). The L1 cache is substantially larger than the transactional cache, and eliminates the need to maintain coherence across two same-level caches. This architecture adopts lazy versioning and eager-eager conflict detection and resolution, since this type of conflict management requires less changes to the cache coherency protocol.

Our approach to deal with overflows is however different from [2, 63, 31]. We propose an efficient yet simple mechanism that does not require any extra hardware structures. In case of overflow, an interrupt is sent to all the other non-overflowing cores. The service routine associated to the interrupt sets the core into a low-power mode, with the overall effect of stopping the execution. Only at the end of the overflowing transaction the other cores may resume executing.

As a lazy conflict resolution guarantees more throughput [46], we devised a mechanism that, still based on cache coherency like FlexTM [68], detects conflicts at access time and defers resolution at commit time. For heavily conflicting workloads, we present an effective and yet simple method that reduces contention by serializing transactions. As for overflow, the mechanism is entirely SW based, and it does not require any extra bus wires or changes to the CPU.

To further reduce the probability of overflow, we investigated a logging solution similar to Log-TM [52]. However, we exploited an important peculiarity of our target architecture: the lack of virtual memory. This fact allows a fundamental optimization: the log-controller can easily identify a memory access that belongs to the private address space with a simple decoding operation. As we mentioned
earlier, logging also accesses to shared memory requires longer roll-back times (i.e., an aborted core must restore more entries in memory). Even worse, the other cores must stall for the entire duration of the roll-back operation (since the shared memory locations must be restored to the original state), with the overall effect of serializing the system. Our proposal, instead, adopts a parallel abort scheme. Since we are logging private addresses, rolling-back is a core-local operation (i.e., the other cores can safely continue executing). Hence, our method still reduces the number of transactional entries to save in the L1 cache, without incurring in any serialization in case of abort. Also, as opposed to LogTM that requires saving an entire line into the log, we propose a method (based on a filter cache) to track single words. Beside the static usage of the “W” bit introduced by LogTM [52], we adopt a more aggressive dynamic solution that prevents logging of obliviously-written memory locations (i.e., private memory locations whose initial content is always re-initialized within the transaction). We will discuss all these policies in more detail in the first part of Chapter 5.

Finally, at the end of Chapter 5 we show a more flexible solution for conflict management, decoupled from cache coherency, and based on an external hardware module (i.e., the Bloom module). The module is programmable with simple memory operations, and allows not only the adoption of different conflict resolution policies, but also an efficient mechanism to prioritize transactions.

2.3.4 Extensions to Compiler

As the last part of this work, we add compiler features to interact with our hardware TM module, with the goal of simplifying the programmability of a multiprocessor embedded system.
We propose a HW/SW framework in which transactional features are triggered through the use of a custom set of compiler directives, implemented as an extension to the popular OpenMP [79] programming model. OpenMP offers the programmer a set of parallel constructs with a level of abstraction higher than simple critical sections. Inter-thread synchronization is transparent in OpenMP, since the programmer has just to declare the parallel blocks without worrying of the synchronization details, as opposed to a pthreads-based synchronization [14], for example. Furthermore, a program compiled in OpenMP is more scalable and portable, since the number of parallel tasks is always tied to the number of cores (i.e., no modifications to the source code are required when porting the application to a platform with different degree of parallelism).

It should be noted that a similar approach has been proposed by the authors of the OpenTM compiler programming framework [5], which extends the OpenMP API with compiler directives to express TM programming patterns. However, key features such as the ordered clause in OpenMP (which allows speculative code parallelization) were not implemented for transactional execution, and no discussion about how this could be practically supported was provided. Indeed, this feature requires specific support from the underlying TM system to order commits. For these reasons, we decided to use the OpenTM programming interface and compiler as a starting point.

Our enhanced compiler transforms the annotated program so as to interact with our runtime system, where low-level APIs are transparently used. To further improve ease of programming, our framework supports speculative task- and data-level parallelism. Specifically, loops which may carry cross-iteration dependencies (non-DOALL) can be annotated as parallel loops. Similarly, tasks which may possibly be inter-dependent can be annotated for parallel execution. Several embedded ap-
applications exhibit a similar pattern [37], thus making support for speculation very valuable in this domain. Note that other approaches based on TM and ordered commits have been proposed in the general purpose domain [47] [76]. However, previous TM-based approaches are unlikely to be beneficial for embedded systems, due to the software (and hardware) complexity, which induces high overhead, and thus also high energy consumption.

In Chapter 6 we will describe how we enabled speculative parallelization of target applications by enhancing our Bloom module design with the capability of ensuring a predetermined commit order for concurrent transactions. The underlying TM system ensures that, in case a real dependence arises, the original sequential program semantics is preserved. In our TM system this can be achieved by forcing transactions to commit in program order, thanks to specific hardware support for prioritized commit that we provide. At the program level, this feature can be leveraged by the OpenMP worksharing constructs (parallel sections and parallel for) to generate parallel transactions holding speculative workloads (tasks or loop iterations). The compiler properly generates code that programs the Bloom module to commit speculative transactions in order.
Chapter 3

Energy-Efficient Locking Mechanisms

Synchronization among tasks accounts for a sizable fraction of the energy consumption and execution time of applications running on Multi-Processor Systems-on-Chips platforms. In order to achieve fast and energy-efficient operations, it is therefore essential to implement efficient and power-frugal synchronization primitives. The design of such primitives is complicated by several software and hardware issues, such as: processors running at different speeds, different implementations of the waiting phase upon entering the critical section, and the ratio between static and dynamic power. Taking into account all these issues, which are normally not considered in synchronization primitives for traditional multi-processor systems, is a complicated task; in fact, no solution has been proposed which is consistently superior to others.

In this chapter, we compare a set of classical implementations (i.e., based on busy
waiting, or on sleep states) of mutex semaphores, and propose a hybrid (wait/sleep) semaphore in which the sleep state is entered only after a number of busy-wait cycles. The proposed scheme provides the best overall energy-delay product with respect to previously proposed schemes. Furthermore, we identify an optimal length of the busy-wait cycles, which is empirically shown to depend on the time required to switch from the sleep to the active state.

3.1 Energy-Optimal Synchronization Primitives

3.1.1 Multi-Core Reference Platform

Hardware

Our underlying architecture is built off MPARM [43] and consists of:

- A configurable number of ARM7-like processors, with private instruction and data caches;
- The main memory, split in equally-sized banks;
- Snoop devices to enforce cache coherence;
- A shared bus interconnection structure;
- A frequency divider, that allows each processor its own frequency;
- A hardware device dedicated to synchronization;
- Cycle accurate power models, related to each component.
The synchronization device provides a TestAndSet feature with which it is possible to implement a conventional spin-lock. The device consists of a bank of registers, mapped on the address space of cores, which can be written as a normal variable. However, reading from one of those registers will return its content, and simultaneously store a “1” into it. The content of such registers is never cached, to avoid the burden of keeping their consistency among different processors, resulting in an extremely simple synchronization mechanism.

However, the use non-cacheable locks can quickly end up saturating the bus, in particular when the coupling among tasks increases. In order to support sleep-based locks, as described in [27], we have therefore modified the simulation platform by implementing the following:

- **An atomic TestAndSet instruction that can handle cached variables.** The TestAndSet instruction has to interact with the bus and the snoop devices because it can modify shared data. The cores, hence, must be aware of its behavior to have a chance to update (or invalidate) their local copies of the data. Of course, if the write policy of the caches is write through (as it is in the platform), the main memory must also update its copy of data when the TestAndSet results in some changes.

- **A sleep/wakeup mechanism that allows cores to enter and exit a low-power state.** The low-power state is entered through a special sleep instruction. When executed, this instruction halts execution and brings the core to a low-power state where it only consume static energy due to leakage currents. We also introduced a wakeup instruction: when executed by a processor, all the halted cores are brought back to the normal state and their execution is resumed.
Using these two instructions, therefore, it is possible to wait for a semaphore to be unlocked with a reduced energy consumption (at the cost of reduced responsiveness).

### 3.1.2 Frequency Regulation

Since the synchronization is based on the polling of shared variables (either in main memory or in local caches), it is possible, by inspecting the access patterns of a processor to the data, to determine how much time is spent by each core while doing synchronization. Using this information, it is possible to evaluate the past workload of each processor, which can be used as an estimation of the future workload and in turn used to set the frequencies of the cores accordingly. We refer the reader to the original paper [44] for a more detailed description of the algorithm.

For this purpose, we added a hardware device to the simulation platform that monitors the interfaces between each core and its private cache. The device identifies a series of repeated accesses to the same shared memory location as an attempt to acquire a lock; when this situation occurs, it starts counting these repeated accesses. Eventually, the amount of time that a processor has spent in the low-power state is also added to the total count of the repeated accesses. The value of the count is called *synchronization time*.

After the expiration of a fixed time window ($TW$), the device uses, for the $i$-th core, the measured synchronization time ($T_{Si}$) and the core speed ($S_i$) to compute the past workload $W_i$; workload is measured as number of executed instructions excluding synchronization, that is

$$W_i = \frac{TW-T_{Si}}{S_i}.$$  

For each processor, the workload for the next time window is assumed to be equal
to the past one which is a reasonable assumption, in case the time window is small compared to the overall execution time); the device then tunes the core speeds ($S_i$) to reduce (ideally nullify) the future synchronization times ($T_{S_i}$).

This frequency regulation is fully handled by the hardware, thus it is totally transparent to the software whose synchronization behavior does not need any changes.

### 3.1.3 Software

Since we focus on the behavior of the synchronization, we removed all the software levels between the user level application and the low level synchronization routines. In particular we removed the whole operating system to avoid the noise in measurements that such a software layer would inevitably introduce. However, stripping the OS moves the burden of task mapping onto the programmer. Thus, we are forced to establish a static one-to-one mapping between cores and tasks when writing the software. Such an approach has the further benefit of avoiding the overhead caused by the switching and the migration of tasks that, in this analysis, is not related to synchronization.

### 3.1.4 Energy-Optimal Sleep-Based Semaphores

Since we are interested in analyzing the interactions among busy-waiting, the availability of a low-power state, and the possibility of having cores running at different speeds, we have implemented a small subset of the many variants of spinlocks proposed in [27]. More specifically, we considered four versions of the `WAIT()` function used by a core to acquire a semaphore:
**HW-locks**: The acquisition of the semaphore is performed by indefinitely polling a shared variable using the TestAndSet instruction. The system does not use any DVS policy (all processors run at the same speed) and cores never enter a low-power state.

**DVS-policy**: As in HW-locks, but with the DVS subsystem enabled. Therefore the times spent for synchronization are monitored and the cores’ frequencies are tuned accordingly.

**HW-locks-sleep**: As in HW-locks, the semaphore is acquired by polling the hardware device and DVS is not enabled. However, after a fixed number \( \text{MAXSPINS} \) of iterations, the core enters the low-power state through a `sleep` instruction. It will be resumed by the `wakeup` instruction issued by a core that releases the lock.

**DVS-sleep**: Both a low-power state and DVS are part of the scheme. The busy waiting lasts \( \text{MAXSPINS} \) iterations at most; after that time, the core executes a `sleep` and it falls in the low power state. Concurrently, the DVS block settles the processors’ frequencies as determined by the synchronization times collected in the chosen time window.

Figure 3.1 shows the pseudo-code of the `WAIT` function as used by DVS-sleep and the HW-locks-sleep semaphore types; the code is the same since the difference between these two solutions is only in the activation or deactivation of the DVS subsystem, which, as we said, is transparent to the software.

A core spins on the semaphore (Line 3) until the semaphore becomes free. In Line 4, the expiration of the timeout, defined by \( \text{MAXSPINS} \), is checked; if expired, the core enters the low-power state.
Notice that some overhead (on the order of few µs [26] [72]) is introduced when switching from the active to the idle mode. In general, entering the low-power, idle state may be convenient from an energy point of view depending on two parameters: i) the duration of the semaphore’s busy-period, and ii) the state-transition overhead. In case a DVS device is present in the platform, the clock rate and the voltage of each CPU may vary during the busy-waiting operation. Hence, it is possible to save even more energy by allowing the CPU to scale down its voltage and frequency before entering the low-power state.

**Figure 3.1:** The WAIT() function.

```c
void WAIT(int semaphore_ID) {
    int ntimes = 0;
    while(isBusy(semaphore_ID)){
        if (ntimes >= |MAXSPINS|){
            GO_IDLE();
            ntimes = 0; //reset
        }
        ntimes++;
    }
}
```

From the pseudocode of Figure 3.1 it is evident that the choice of the MAXSPINS parameter is crucial, and may significantly affect the time and/or energy efficiency of the WAIT() function. An analytical derivation of an optimal value of MAXSPINS is in principle possible. The existence of an optimal timeout value for a power management policy has in fact been introduced by Wu and Xiong in [78], in which they formally show that when the distribution of idle times has heavy tails (as opposed to an exponential distribution, which falls very quickly), an optimal timeout
value does exist. Their analysis, however, was carried out on uni-processor systems, without DVS support, and optimality was referred to dynamic energy. Extension of their analysis to multi-processor systems, with processors running at different speeds, while considering total energy as a metric, has not been considered and it has been left for future work.

In order to understand whether their basic assumption applies to our case, we have collected, for a set of benchmarks, the same type of data used in [78], namely, the distribution of the idle times. Since we are dealing with a sleep-based semaphore, idle time $T_{idle}$ is defined in this case as the sum of three components:

1. time spent spinning on the semaphore ($T_{spin}$, Line 3);
2. CPU state-transition time ($T_{overhead}$, Line 4);
3. time spent in the low-power state ($T_{sleep}$, Line 5).

Figure 3.1 shows these distributions, computed using a value of $T_{overhead} = 1000$ cycles. It is quite evident that tails are relatively thin, and seem to match better the shape of an exponential distribution. Moreover, increasing the value of MAXSPINS from 10 to 100 seems to make tails even thinner, contrary to expectations (idle times values should tend to increase because $T_{spin}$ increases).

Based on this empirical evidence, we decided to resort to an explorative analysis of a pre-defined set of MAXSPINS values, described in the following section.
3.2 Experimental Results

In this section we evaluate the impact of the proposed solution on both performance and energy consumption. In order to have a better understanding of the different tradeoffs that may arise, we tested our architecture under a variety of hardware and software configurations.

We focused on two hardware parameters: the state-switching overhead ($T_{\text{overhead}}$) and \textsc{Maxspins} variable. For the state-switching overhead we considered two extremes: the ideal case (i.e., no overhead), and a conservative value (i.e., $T_{\text{overhead}} = 1000$ cycles). On the other hand, \textsc{Maxspins} varied on a range of four points (i.e., 10, 20, 50 and 100 maximum loop iterations).

Finally, we stressed the platform with a mixture of well-known kernel applica-
tions, that should cover a representative set of possible workload scenarios. Our benchmark suite is composed of:

- **Eratost**: A prime number search on a shared vector. It implements a parallel version of the *Eratosthenes* algorithm. Each processor scans a disjoint portion of a shared vector, and cores are synchronized using barriers.

- **Mergesort**: A parallel version of the mergesort algorithm; it sorts a pool of randomly generated vectors stored in shared memory. The sorting phase is parallelized using barriers while the merging phase uses pipelining. The workload allocation among the cores is not uniform and varies during the execution.

- **DES**: A Data Encryption Standard algorithm parallelized in a master-workers-slave fashion. The master core sends the stream to two workers that encode it and send it to the slave core. The resulting workload is not uniform among cores but it is static.

- **FFT**: a variant from the SPLASH-2 [69] suite, modified to perform two concurrent FFTs on two distinct vectors of different sizes. In order to obtain a non-uniform and non-static workload allocation, The computation is repeated several times and the role of the processing elements is changed after each iteration. Half of processors are in charge to perform the computation of the first vector, while the other cores handle the second vector.

During our experimentations, we also considered a mix of the aforementioned applications, that we named ‘Application Mix’. Our empirical discussion on the experimental data will proceed as follows. First, we show that an optimal MAXSPINS value exists depending on the application and the state-switching overhead. Second,
Figure 3.2: Total cycles, energy and EDP as MAXSPINS varies for the DVS-sleep system configuration. Negative percentages indicate improvement relative to MAXSPINS=10.

we plug this optimum value into the platform, and compare the results against the other synchronization schemes.

Figure 3.2 reports the execution cycles, the system energy and the Energy-Delay Product (EDP) data of various applications for the DVS-sleep policy with increasing MAXSPINS values, for a state-switching overhead of 0 cycles and 1000 cycles respectively. All the values are normalized to the case of MAXSPINS=10.

In the ideal case (i.e., no overhead), letting the CPU spin for only a small amount of time before falling into the sleep state is in general preferred for reducing the execution cycles. We see, in fact, that increasing MAXSPINS from 10 to 100 hurts
Figure 3.3: Total cycles and energy normalized to HW-locks when MAXSPINS=10 and switch overhead=0. Negative percentages indicate improvement.

significantly the overall performance (e.g., up to 20% for MAXSPINS=100 in the Application Mix). However, this trend is not followed when considering energy. In fact, though the energy-optimal MAXSPINS value is still 10, in general the energy curves are not following any clear trend. For example, MAXSPINS=50 gives overall better energy savings than MAXSPINS=20.

On the other hand, with a more realistic value for state-switching overhead ($T_{\text{overhead}}$=1000 cycles), new tradeoffs in energy-performance are defined. Depending on the duration the semaphore’s lock period, it may be convenient to busy-wait for a longer time period (by increasing the value of MAXSPINS) before entering the power-idle state. In fact, we can see that the best performance and energy savings are achieved with MAXSPINS=20 and MAXSPINS=100 respectively.

When considering the Energy-Delay Product, we found the optimal MAXSPINS values to be 10 and 50 (the latter by around 5%), for $T_{\text{overhead}}$=0 and $T_{\text{overhead}}$=1000 respectively.
Figure 3.4: Total cycles and energy normalized to HW-locks when MAXSPINS=50 and switch overhead=1000. Negative percentages indicate improvement.

Next, we compared the DVS-sleep policy with the optimal MAXSPINS values against HW-locks, DVS-policy and HW-locks-sleep. Figure 3.3 and 3.4 show the execution cycles and system energy of various applications, normalized to HW-locks, for switching-state overhead of 0 and 1000 cycles respectively. As before, negative percentages indicate improvement relative to HW-locks. Overall, the DVS-sleep scheme guarantees the best performance in terms of execution cycles, regardless of the switch-overhead. On the other hand, system energy results vary and while the DVS-sleep policy is always the best choice when there is no switch-overhead, it is not always the best option when the switch-overhead is set to 1000 cycles.

The EDP data in Figure 3.5 shows that with no switch overhead the DVS-policy may obtain an average improvement in EDP of about 35% (against the 11% of DVS-policy and 18% of HW-locks-sleep). Notice that with no overhead, even the simple sleep-based version of the lock outperforms the DVS-policy configuration.

With an overhead of 1000 cycles, Figure 3.6 shows that the DVS-sleep scheme
**Figure 3.5:** EDP normalized to simple locks when MAXSPINS=10 and switch overhead=0. Negative percentages indicate energy-delay improvement relative to locks.

![Energy-Delay Product](image)

**Figure 3.6:** EDP normalized to simple locks when MAXSPINS=50 and switch overhead=1000. Negative percentages indicate energy-delay improvement relative to locks.

![Energy-Delay Product](image)

with an optimal MAXSPINS value does a better job at balancing both energy and performance than the other schemes (e.g., the overall EDP is improved by 25% against the 11% of DVS-policy, and 2% of HW-locks-sleep). The results confirm that both energy and performance need to be taken into account when choosing an
appropriate synchronization policy.

### 3.3 Summary and Discussion

In this chapter we have presented an explorative analysis of various implementations of mutex semaphores suitable for multicore architectures, with the purpose of analyzing their energy and performance efficiency. Besides widely used versions, we also propose a novel hybrid (i.e., a combination of busy waiting and sleep-based semaphore) synchronization primitive in which the sleep state is entered only after a number of busy-wait cycles (essentially, a timeout value). The latter quantity is the main parameter evaluated in our exploration, since energy and performance of an application will be strongly affected by this timeout value. Our experiments provide two main conclusions: First, we show that the proposed timeout-based semaphore provides overall higher energy savings (30% on average), compared to previously proposed semaphores. Second, although each application will in general have its own optimal timeout value, it is possible to identify an average optimal length of the timeout value, whose value is empirically shown to be dependent on the time required to switch from the sleep to the active state.

The analysis proposed in this Chapter is mostly of explorative nature. As a future research direction, it would be interesting to analytically derive an optimal value for the sleeping interval of the semaphore. Such an optimal value will in general be application-specific, although an average optimal value can be defined for an application mix. The possibility of the existence of an optimal timeout value has been described by Wu and Xiong in [78]. Their analysis was carried out on a single processor machine, and optimality was referred to dynamic energy. The extension of
this idea to multi-processor systems while considering total energy (i.e., active and static) as a metric is still an open research issue.

In this chapter we presented an energy-driven optimization technique for a conservative synchronization mechanism (i.e., locking). However, as we explained in Chapter 2, locking has inherent parallelism issues, since threads result serialized when executing critical sections. Furthermore, using lock is prone to errors and failures (e.g., deadlocks, livelocks etc.). Transactional Memory has been proposed in the past as an speculative alternative to locking for general-purpose architectures (see Chapter 2). In the next Chapter we will study its applicability to an embedded platform, and explore the energy implications of various design choices.
Chapter 4

Transactional Memory for
MPSoCs

In the previous Chapter we presented various energy-efficient locking solutions, and studied their performance. However, despite their widespread use, locks have serious limitations. Coarse-grained locks, which protect relatively large amounts of data, simply do not scale. Threads block one another even when they do not really interfere, and the lock itself becomes a source of contention. Fine-grained locks, which protect smaller regions of memory, may appear more scalable, but they are difficult to use effectively and correctly. In particular, they introduce substantial software engineering problems, since the conventions associating locks with objects become more complex and error-prone. Locks also cause vulnerability to thread failures and delays: if a thread holding a lock is delayed by a page fault, or context switch, other running threads may be blocked. Locks also inhibit concurrency because they must be used conservatively: a thread must acquire a lock whenever there is a possibility of a synchronization conflict, even if such a conflict is actually rare. Transactional
Memory [33] has been proposed as a speculative alternative to locks, as discussed in Chapter 2.

In this Chapter we present *Embedded-TM*, a hardware transactional memory architecture specifically tailored for MPSoCs. We start from a baseline design consisting of an extra Transactional Cache (TC) attached to each core, and show how such simple design can be beneficial in terms of performance and energy consumption. In the second part of the Chapter we introduce several optimizations to the baseline configuration, each tackling different limitations of the proposed transactional memory architecture.

### 4.1 Architecture

We developed and tested our Embedded-TM designs using the MPARM simulation framework [4, 43], a cycle-accurate, multi-processor simulator written in SystemC. MPARM models any simple instruction set architecture with a complex memory hierarchy (supporting, for example, caches, scratch pad memories, and multiple types of interconnects). MPARM also includes cycle-accurate power models for many of its simulated devices. The power models reflect a 0.13µm technology provided by STMicroelectronics [71], and the energy model for the fully associative caches is based on Efthymiou and Garside [23].

Figure 4.1 shows the basic system configuration. It consists of a variable number of ARM7 cores (each with a 4KB 4-way L1 data cache and 4KB 4-way L1 instruction cache), a set of private memories (256KB each), a single shared memory bank (256KB), and one bank (16KB) of memory-mapped registers serving as
Figure 4.1: Example configuration with 4 CPUs.

hardware semaphores. The interconnect is an AMBA-compliant communication architecture [1]. A cache-coherence protocol (MESI) is also provided by snoop devices connected to the master ports. Platforms featuring such cache-coherency subsystems are common (e.g., the ARM11 MPCore Multiprocessor [28]). While the private and shared memories are sized arbitrarily large (256KB each), they do not significantly impact the performance or power of our system (see Section 4.2).

Each transaction is speculatively executed by the CPU. If no conflicts with another transaction are detected, its effects become permanent, and the transaction commits. Otherwise, if conflicts are detected, its effects are discarded, and the transaction aborts and is later restarted.

What does it mean to detect a conflict with another transaction? Like most HTM schemes, our Embedded-TM architecture relies on an underlying cache coherence protocol to detect when data cached by one CPU is invalidated by another (we use a
conventional MESI protocol). A transaction can cache a data item either for reading or for writing. A conflict occurs when two transactions concurrently attempt to cache the same data item, and at least one caches that item for writing.

Cache coherence protocols are notoriously complicated and difficult to prove correct. Our goal of simplicity means that we consider only designs that make few, preferably not any, changes to the native cache coherence protocol. While it is easy to propose radical protocol changes on paper, our reliance on a cycle-accurate simulator “keeps us honest” in the sense that we cannot ignore implementation difficulties. We are also concerned that radical changes to highly-optimized protocols limit portability and broader acceptance.

Hardware support for transactional memory is organized around these components:

1. A cache coherence protocol for detecting data conflicts,
2. A buffer for storing and modifying transactional data, and
3. A rollback mechanism for re-executing aborted transactions.

What happens when a transaction overflows its buffer? Given our need for simplicity, we take a brute-force approach. The transaction continues in serial mode: all other CPUs are suspended, and the privileged transaction runs in isolation. It uses the entire memory hierarchy, and runs non-speculatively to completion.

Figure 4.2a) shows an overview of our Embedded-TM architecture. To start a transaction, the CPU creates a local checkpoint by saving the register contents to a small (128B) scratch pad Memory (SPM) [7]. (The pipeline is stalled while the CPU
Figure 4.2: a) Architectural configuration to support hardware transactional memory. Note the transactional cache holds all transactional data. b) New architectural configuration to support hardware transactional memory using a victim cache (VC). Note that the primary storage structure for transactional data is now the L1 cache. In case of conflict evictions, transactional data can be held in the VC.

writes to the SPM.) The SPM must be large enough to hold all the CPU registers.

4.1.1 The Transactional Cache

The Transactional Cache (TC) is a small (512B) fully-associative memory. Each transaction keeps two copies of the data it accesses in the TC: a working copy and a backup copy. If the data is found in the L1 cache, it must be invalidated there before being placed in the TC. The transaction modifies the working copy. If there is no data conflict, the transaction completes successfully, backup copies are invalidated, and the working copies become visible. On a data conflict, the snoop device notifies the CPU, invalidating the working copies, and restoring the backup copies. The CPU enters a low-power mode, and after a random backoff, re-executes the transaction. Starvation is practically avoided by adopting an exponential distribution for
the backoff periods, and also by setting a threshold for the maximum number of consecutive aborts the system allows before switching the execution in serial mode (see Section 4.1.3). Note that our model includes a realistic state-switching overhead (i.e., idle to active).

When reading or writing data from memory, the TC is checked first (1 cycle). On a TC miss, the rest of the memory hierarchy (starting with the L1 cache) is searched. To save power, the caches are accessed sequentially. Since TC misses are rare (we found the average TC miss rate to be about 5%), this serial access has a negligible impact on performance.

Because the TC is fully associative, it makes effective use of limited space, but it is expensive in terms of energy and power. As shown in Figure 4.3, the TC accounts for about 30% of total energy consumption when running a sample benchmark. \(^1\)

\(^1\)Figure 4.3 shows energy distribution for the matrix-microbenchmark with 5% of the time spent inside transactions, as described in Section 4.2.1.
Making the TC set-associative would save power, but makes it more likely that a transaction may overflow a set. Restarting such a transaction in the TC would probably be futile, since the same overflow is likely to recur, forcing the overflowing transaction to run in serial mode.

It is not strictly necessary to keep valid data in the TC after a transaction commits. We in fact show that it is often more energy-efficient to write back the modified lines to the traditional cache hierarchy after the commit, allowing the TC to be powered down when not in use. This approach is called *aggressive shutdown* mode. The CPU is stalled while the TC is flushed. Powering up the TC at the start of each new transaction incurs a $0.2 \mu s$ (40 cycles) overhead. Powering down the TC improves energy efficiency most of the time, although is not a good strategy when there are back-to-back transactions, since it results in unnecessary traffic between the TC and the rest of the memory hierarchy. We evaluate these issues in more detail in Section 4.2.3.

### 4.1.2 Size Limitations

The baseline Embedded-TM architecture with aggressive shutdown is simple, and it provides good performance and energy efficiency for transactions with data footprints small enough to fit in the TC. If the transaction’s data footprint is too large to fit in the TC, then the transaction must be run in serial mode, which is unattractive because there is no concurrency. The absence of concurrency is not troubling for transactions that have data conflicts, because such transactions must be serialized in any case. Lack of concurrency does matter for large, non-conflicting concurrent transactions. We face a dilemma: a larger TC means larger transactions, but substantially increases power consumption.
It seems wasteful to have transactions overflow the small TC when the L1 cache, normally much larger than the TC, may have plenty of room. Here is an alternative design that uses the L1 cache for both transactional and non-transactional data. Because there is much more memory to hold transactional data, the likelihood of overflow should be substantially reduced. Unfortunately, because it is impractical to make the L1 cache highly associative (especially in a power-constrained embedded platform), we have replaced one problem with another: a set-associative combined cache introduces the danger that an unlucky transaction may overflow a set while the rest of the cache still has room.

Our solution is to place a victim cache (VC) between the L1 cache and main memory to catch transactional data evicted from the L1, as shown in Figure 4.2b). The notion of a victim cache is not new (e.g., [36, 77, 6]), but our application of this technique is novel because we are the first to use this notion to improve the energy and performance of HTM.

Normally, all transactional data resides in the L1 cache. The victim cache is used only when a transactional entry is evicted from the L1 cache. Here, too, transactions access the caches sequentially, testing first the L1, and only after an L1 miss testing the VC. This strategy saves energy without hampering performance since most accesses hit in the L1 cache. Because the VC is used only when a set in the L1 overflows, the VC can be designed to be smaller than the TC used in the baseline design.

Transactions execute concurrently while the VC is in use. If, despite everything, the VC overflows, then the transaction continues in serial mode. Because the combination of the L1 and VC provides much more room than the baseline TC, overflows are much less common, improving both performance and energy consumption. In
terms of high-level architecture, the core design remains virtually identical to the baseline design except that the TC has been replaced by a smaller VC.

Unlike the TC, the L1 and VC caches do not hold backup copies of transactional data. When a transaction aborts, the CPU refills the line from memory, which increases bus traffic and reduces both performance and energy efficiency. However, this cost should be negligible if the abort rate is low enough. Because the VC is utilized only when transactions overflow the L1 cache, it makes sense to keep the VC powered down unless needed. Like the TC, powering up the VC costs on the order of tens of cycles (i.e., 40 cycles). The experimental results in Section 4.2.4 show that, overall, the L1+VC design is better than the baseline TC design, both in terms of energy and performance.

4.1.3 Reducing Abort Rates

Reducing transaction overflow exposes more parallelism, which in turn exposes more data conflicts. All Embedded-TM designs described so far use eager conflict detection and resolution, meaning that conflicts are detected and resolved when a transaction accesses a location, rather than waiting until a transaction is ready to commit. Eager detection is attractive because it requires minimal modifications to the original cache coherence protocol. In particular, neither new bus states nor new coherence signals are needed. Suppose a transaction at one CPU is about to write to an address whose data has been modified by another transaction at another CPU. As part of the cache coherence protocol, the writer broadcasts an invalidate signal for that address on the bus. The other snoop devices monitor the bus, and detect the data conflict. The snoop device for the CPU whose cache holds that data informs the CPU, which aborts and restarts the transaction, typically after a backoff period.
This approach is lightweight, and fits quite well with the hardware restrictions of an embedded platform. A design that relies on limited hardware modifications not only eases verification, but also significantly increases the portability of such methods to other invalidation-based cache coherence schemes (e.g., MOESI, MSI).

Nevertheless, applications with high data conflict rates tend to perform poorly under an eager scheme. For example, long-running transactions could be repeatedly aborted by shorter conflicting transactions [12], yielding high abort rates, lower performance, and higher energy consumption. We next consider two alternative conflict resolution schemes. The first is simply to revert to serial mode (forced-serial) as soon as the abort rate exceeds a threshold [53]. This approach is attractive for its simplicity: only a few simple abort counters and control signals need to be added to the existing design, since hardware is already in place to handle serialized execution in case of an overflow.

The second, more complex alternative is a lazy conflict resolution scheme. As before, data conflicts are detected as they occur, but instead of resolving them eagerly when they are detected, they are (lazily) left unresolved until commit time. This scheme requires more substantial modifications to our embedded platform, but understanding the effectiveness of such trade-offs is essential to understand how to adapt HTM to embedded platforms.

We are not the first to consider hardware support for lazy conflict resolution.

Both FlexTM [68] and EazyTM [74] leverage cache coherency to detect conflict at access time, and exploit special hardware and software structures to resolve conflicts at commit time. Both architectures require important changes to the CPU (i.e., new instructions), to the cache coherency protocol (i.e., new states), and to the
bus interconnect (i.e., new core-to-core wires). Our method tries to minimize the hardware requirements without penalizing the performance. Our contribution is also to explore the energy implications of such an approach for mechanisms appropriate for embedded systems.

Here is how the lazy scheme works. Each CPU keeps two $N$-bit bitmaps, where $N$ is the number of CPUs. The \textit{RW} bitmap tracks which CPUs wrote data that the local CPU had previously read (see Figure 4.4 for an architectural overview). The \textit{WR} bitmap tracks which CPUs read data that the local CPU had previously written. Both bitmaps track which CPUs wrote data that the local CPU had previously written. See Figure 4.5 for a flowchart depicting how bits are set in the MS and MS bitmaps.

Our lazy scheme effectively serializes transaction commits, somewhat like TCC [30]. When a transaction is ready to commit, it asserts a signal to notify the other CPUs that a commit protocol has started. When the other CPUs detect this signal, they suspend what they are doing to participate in the protocol. When all CPUs have
Figure 4.5: Flowchart for setting bitmaps to support lazy conflict resolution.

paused, the committing CPU broadcasts its WR bitmap and ID on the bus. Each CPU currently running a transaction checks whether its ID is in the committing transaction’s WR bitmap, or whether the committing ID is in its local RW bitmap. If so, that CPU aborts, invalidates cached transactional entries, and broadcasts its ID. All transactions remove the IDs of the committing and aborting transactions from their local bitmaps. The commit protocol ends when all transactions have been heard from.

We would have preferred to use a non-blocking lazy protocol that did not serialize commits. Unfortunately, we found that any such protocol would require pervasive changes both to the underlying hardware (for example: extra bus wires, more kinds of interrupts, etc.), and to the cache coherence protocol (there are many subtle race conditions). Such changes would take us too far from mainstream embedded system architectures.

We found that lazy conflict resolution improves both the performance and energy efficiency of the high-conflict benchmarks that fared poorly under eager conflict
resolution. However, although the serialization overhead of lazy conflict resolution is low, it tends to penalize low-conflict applications. These results are described in detail in the next section.

4.2 Experimental Results

In this section we evaluate our proposed Embedded-TM design using a mix of applications. We first describe the benchmarks used in our experiments as well as our experimental setup, followed by a detailed discussion of our results.

4.2.1 Software

To test our ideas, we chose a range of different applications. Four of these applications were taken from the STAMP benchmark suite [50] and one from the MiBench suite [29]. Note that we selected a representative subset of the STAMP benchmark suite. Porting the entire STAMP suite is non-trivial due to inherent limitations of the simulation platform, and is left for future work.\(^2\)

- **Vacation** (STAMP): implements a non-distributed travel reservation system. Each thread interacts with the database via the system’s transaction manager. The application features large critical sections.

- **K-means** (STAMP): a partition-based program (commonly found in image filtering applications). The number of objects to be partitioned is equally

\(^2\)For example, the lack of OS requires a workaround for handling I/O, and no CAS instruction support means implementing locks via a semaphore-based Test&Set.
subdivided among the threads. Barriers and short critical sections are both used to provide synchronization.

- **Genome** (STAMP): a gene sequencing program. A gene is reconstructed by matching DNA segments of a larger gene. The application has been parallelized through barriers and large critical sections.

- **Labyrinth** (STAMP): the program finds the shortest-distance paths between pairs of starting and ending points in a 3D maze using Lee’s algorithm. Synchronization is provided by a mix of short and large critical sections.

- **Redblack, Skiplist**: applications operating on special data structures (i.e., redblack-trees and skip-lists). The workload is composed of a certain number of atomic operations (i.e., inserts, deletes and lookups) to be performed on these two data structures. Redblack-trees and skip-lists constitute the fundamental blocks of many memory management applications found in embedded applications.

- **Matrix Microbenchmark**: implements a filtering method on a matrix of data and is representative of image-processing applications typically found in plotters, printers, and digital cameras. The data is partitioned into overlapping sub-matrices, each mapped to a particular thread. The workload can be parameterized in terms of time spent executing critical sections. In our simulations we set the workload as 5%, 20%, 60% and 85% of the time spent within critical sections.

- **Patricia** (MiBench): uses a Patricia Trie data structure to quickly perform IP address prefix matching for applications such as an IP subnet, or network or routing table lookups. The computation has been equally distributed among the cores, so that each thread operates on a subportion of the original input
data. Note that in the lock-based version a global lock must be acquired by each thread before starting a new operation.

For each set of applications we also considered an average of the results, called the “Application Mix”.

### 4.2.2 Hardware

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Configuration(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>ARMv7, 3-stage in-order pipeline, 200Mhz</td>
</tr>
<tr>
<td>L1 cache</td>
<td>4KB 4-way Icache, 4KB 4-way Dcache</td>
</tr>
<tr>
<td>Cores</td>
<td>{4, 8}</td>
</tr>
<tr>
<td>Tx Policies</td>
<td>TM-vanilla, TM-shutdown-L1WB, TM-victim</td>
</tr>
<tr>
<td>Contention</td>
<td>eager, lazy, forced-serialization</td>
</tr>
<tr>
<td>Management</td>
<td></td>
</tr>
<tr>
<td>TC, VC</td>
<td>{Fully Associative}, {64B,512B}</td>
</tr>
<tr>
<td>Bus</td>
<td>Amba AHB</td>
</tr>
</tbody>
</table>

Table 4.1: Overview of the system configurations.

For convenience, in Table 5.1 we report the principal system parameters and relative configurations. We considered the following alternative Embedded-TM implementations.

- **TM-vanilla**: the original transactional memory implementation [33, 24], consisting of an additional transactional cache (TC). The transactional data resides exclusively in the fully-associative TC. The TC is never turned off.

- **TM-shutdown-L1WB**: same as vanilla TM, except that the TC is shut down after each commit, as described in Section 4.1.1. Before turning off the TC, the CPU writes back the modified TC lines into the L1 cache. The overhead
of turning the TC back on is $0.2\mu s$ (i.e., the CPU will stall for 40 cycles when reactivating the TC).

- **TM-victim:** the combined L1+victim cache configuration. The fully-associative VC contains the lines that were evicted from the (transactional) L1 cache because of conflict misses. Similar to TM-shutdown-L1WB, the VC is shut down at the end of each transaction. The modified VC lines will be written back into the main memory (i.e., SRAM).

In addition, we consider the following contention management schemes:

- **eager:** the system resolves a conflict as soon as it is detected (i.e., when a transaction tries to access a modified line in shared memory). The requesting transaction always wins the conflict, effectively aborting all other conflicting transactions. This is the default scheme.

- **lazy:** as with eager, detection happens when the conflicting data access occurs; however, conflicts are not resolved until a transaction completes. The first transaction to complete “wins”, and effectively aborts all other conflicting transactions.

- **forced-serial:** can be run on top of eager or lazy contention management. The system reverts to serialized execution if a transaction has been aborted more than once. Once the transaction completes, the system reverts back to its original conflict resolution policy (i.e., eager or lazy).

Finally, we also considered two different locking schemes for comparison:

- **locking:** a conventional Test&Set, or spin-locking mechanism. The spin-lock
implementation consists of a repeated test on a variable in shared memory, and therefore requires an access to the bus for each access to the lock. The Matrix microbenchmark uses fine-grained locks, the rest of the benchmarks use coarse-grain locks.

- **locking-sleep**: a more energy-efficient locking scheme that tries to avoid wasted energy due to “busy-waiting” until the lock is free. At the first attempt to access a busy lock, the core switches into a low-power state and is woken up as soon as the lock is released by another core. The state-transition overhead when going between active and sleep mode is set to $2\mu s$ (40 cycles). We assume the power consumed by a “sleeping” CPU is 10% of a full-running core (e.g., [34]).

All Embedded-TM configurations incur a penalty of $2\mu s$ whenever a core wakes up from the power-idle state because of an abort due to a data conflict. This specific value was chosen since it is consistent with that found in real embedded systems (e.g., [72, 26]).

Note that while prior work used the TC along with a direct-mapped L1 cache, in this study we increased the associativity of the L1 to 4-way (which is a common degree for the associativity of data caches in embedded platforms, e.g., [72]). Our initial experiments showed this configuration to be the best in terms of energy-delay product for all benchmarks. Therefore, all experimental results shown in this paper assume the 4-way configuration for the L1 cache.
Figure 4.6: Cycle, Energy, and Energy-Delay Product for various benchmarks. All results are relative to locking.
4.2.3 Evaluating Locking vs. HTM using a TC

For each application run, we measured both the total execution cycles and the consumed energy. Then we quantified the energy/performance trade-off by considering the Energy-Delay Product (EDP).

Figure 4.6 shows three graphs, reporting the cycles, energy, and EDP data for the different applications. Note that all results are reported relative to the simple locking scheme. The energy includes the energy used by processor cores, transactional caches, L1 caches, scratch pad memories and RAM. The matrix microbenchmark results show the effect of increasing the proportion of time spent inside a critical section when different number of cores are available. Experiments with longer proportion of time spent in critical sections have a larger total number of instructions executed per transaction. This figure shows that for most benchmarks the aggressive shutdown policy has a better EDP than the vanilla configuration. In some cases, such as Vacation, the overhead of turning the TC off and back on results in a notable increase in the number of execution cycles, which is more than compensated for by increased energy efficiency. Conversely, the aggressive shutdown policy is disadvantageous for the Redblack and Skiplist benchmarks. These benchmarks show a negligible abort rate, and spend close to 90% of their execution inside critical sections. For these reasons, shutting down the TC between transactions does not result in overall energy savings. Note also that $TM_{-shutdown}$-$L1WB$ performs better in Patricia than in Redblack, because of the different workload configuration of the two applications. In Patricia, not only do threads spend less time executing transactions (80% vs. 90% in Redblack), but also conflict more often due to the higher number of object insertions (17% vs. 9% in Redblack).
Figure 4.7: Energy-Delay Product for various benchmarks comparing transactional memory using a traditional transactional cache (with and without shutdown) with a unified L1+victim cache scheme. All results are relative to EDP for the transactional cache with shutdown (TC-shutdown-L1WB).

4.2.4 Evaluating the TC Schemes vs. L1+VC Scheme

The next set of experiments focused on evaluating the potential benefits of using a unified L1+VC scheme. As mentioned in Section 4.1.2, an HTM scheme that relies on keeping all transactional data in a small, fully-associative transactional cache may be severely limited in the type of applications that can efficiently run on the embedded system. That is, applications with large transactions may overflow the TC and therefore will be forced to run serially, even if data conflicts with other transactions do not exist. The TM-victim scheme attempts to reduce the occurrence of overflow by allowing transactional data to reside primarily in the (much larger) L1 data cache. Furthermore, the VC serves as additional storage in case of a conflict miss in the L1, thereby providing additional defense against overflows.

Figure 4.7 compares the energy-delay product for the STAMP and Patricia applications when using a traditional TC scheme (both with and without shutdown) and an L1+VC scheme. Note that all results are reported relative to the TC with
shutdown scheme (TM-shutdown-L1WB) since this was the overall winner from the results reported in the previous section. We focus on these applications since they were the ones that suffered from high overflow rates under the TC-vanilla and TC-shutdown-L1WB schemes and therefore stand to gain the most from the L1+VC scheme. We see that, in most cases, the TM-victim configuration offers the best EDP. The only exceptions are K-means and Labyrinth. Labyrinth shows an interesting behavior. Our experiments reveal that, though TM-victim is capable of boosting the performance by more than 30% compared to TM-aggressive-L1WB, this improvement is not sufficient to compensate for the non-linear increase in energy consumption. A more detailed analysis shows that the L1 cache is responsible to a large extent for the energy overhead. This is due to the fact that now the L1 is used for both transactional and non-transactional data, and also because more instructions must be fetched from the Icache and re-executed in case of abort (since TM-victim allows larger transactions to be aborted). K-means, as previously noted, exploits a barrier type of thread-synchronization. Since transactions are quite rare and small (i.e., less than 5% of time is spent within transaction), shutting down the TC is the most effective policy.

In contrast, other workloads benefit from the higher throughput guaranteed by the TM-victim solution. For example, for the Genome benchmark, we see that TM-victim has a 5X improvement in EDP compared to TM-vanilla for the 8 core 64B configuration. Using the TM-shutdown-L1WB scheme improves EDP a little relative to TM-vanilla, mainly by avoiding accesses to the TC when not executing transactional code. However, since the TM-shutdown-L1WB scheme does not address the problem of overflows, it will not be sufficient in the case of large transactions.

The general trend to note here is that when a system is executing large transactions, the victim configuration can often lead to a significantly better energy-delay
**Figure 4.8**: Overflow rates for various benchmarks. Note that using the L1+VC scheme completely eliminates overflow cases for all benchmarks.

<table>
<thead>
<tr>
<th>TC size (Bytes)</th>
<th>STAMP</th>
<th>Patricia</th>
<th>App. Mix</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>4core genome</td>
<td></td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>8core genome</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>4core kmeans</td>
<td></td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>8core vacation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>4core labyrinth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>8core Patricia</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

product compared to a *TM-vanilla* configuration.

Figure 4.8 shows the transaction overflow rate for STAMP, Patricia and the Application Mix. Using the TM-victim scheme reduces the overflow rate to zero for all these benchmarks. The overflow data for Redblack-Skiplist and Matrix-Microbenchmark has been omitted, since for these benchmarks, either the transactions overflowed 100% of the time (with a 64 Byte *TM-vanilla* or *TM-shutdown-L1WB* configuration), or never overflowed (with all other configurations).

As expected, we see that 1) the overflow rate is very high for *TM-vanilla* and *TM-shutdown-L1WB*, except when the TC is large, and 2) even these high overflow rates can be reduced to zero with a victim configuration. We can also notice that EDP and overflow rates are correlated; usually better EDP corresponds to low overflow rates. For example, as shown in Figure 4.7, on average, *TM-victim* offers the best EDP for a 64 Byte VC configuration. For the same VC size, Figure 4.8 shows the overflow rate dropping by the largest absolute amount when switching from a *TM-vanilla* to a *TM-victim* configuration (i.e., from 90% to 0%).
4.2.5 Evaluating Conflict Resolution Schemes

As mentioned earlier, another important parameter affecting the performance of a transactional memory system is the transaction abort rate. Recall that when a transaction detects a data conflict with another transaction, one of the transactions needs to abort, causing the core executing that transaction to go into a low-power state for some random backoff period while the other transaction continues to execute. Eventually, the core executing the aborted transaction is woken up so it can attempt to re-execute that transaction. This whole process consumes extra energy and cycles, but is required in order to properly synchronize the two transactions. As with the overflow case, the overall effect on the system is to serialize the execution.

As discussed in Section 4.1.3, ideally a “lazy” conflict resolution scheme offers some advantages in terms of throughput, since it is capable of filtering out some false conflicts that would be impossible to detect with an “eager” approach. In general, however, the price to pay for such improvements is an increased complexity of the contention management scheme. An all-hardware approach, like ours, would require non-trivial modifications to the cache coherence protocol as well as the bus protocol.

Next, we quantify the impact of the eager and lazy policies on energy consumption, and then propose an alternative lightweight solution for workloads characterized by high-conflict rates. More precisely, we propose a “forced-serial” mode that can be run on top of eager or lazy contention management, and that is triggered whenever the number of consecutive aborts reaches a certain threshold. As we explained before, once the transaction completes, the system reverts back to its original conflict resolution policy (i.e., eager or lazy).

Figure 4.9 reports the abort rate and the energy-delay product (EDP) for an
Figure 4.9: Abort Rate and Energy-Delay Product for an 8 core TM-victim configuration, comparing three different contention management schemes (ie, eager, lazy and forced-serial). Note that \( \{2, 4, \infty\} \) indicate the number of times a transaction is allowed to abort before reverting to serialized execution (\( \infty \) means never enter forced serialization mode). For the EDP graph, all results are relative to the eager scheme with no forced serialization (i.e., eager with \( \infty \) retries).

8 core TM-victim configuration for different benchmarks. The light and the dark colored bars represent, respectively, the data for the eager and the lazy contention policies. On the x-axis we report the maximum number of retries that are allowed before forcing the system to re-execute the transaction in serial mode. The EDP values are relative to the eager scheme with no forced serialization (i.e., eager with \( \infty \) retries).

As expected, the lazy contention policy reduces considerably the number of aborts. On average, we see that with no forced serialization the abort rate drops from 29% with eager to 9% with lazy. The overall effect of the increased throughput is an average improvement in EDP of about 14%. The forced-serial policy has a
Figure 4.10: Performance (in terms of execution cycles) using different contention management schemes. All the values are relative to the eager scheme with no forced serialization. Results for ideal lazy assume no overhead in synchronizing the reading/writing of the bitmaps when committing a transaction.

<table>
<thead>
<tr>
<th>Application</th>
<th>eager</th>
<th>lazy</th>
<th>lazy (ideal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>genome</td>
<td>100%</td>
<td>79%</td>
<td>76%</td>
</tr>
<tr>
<td>vacation</td>
<td>100%</td>
<td>97%</td>
<td>95%</td>
</tr>
<tr>
<td>kmeans</td>
<td>100%</td>
<td>99%</td>
<td>99%</td>
</tr>
<tr>
<td>labyrinth</td>
<td>100%</td>
<td>94%</td>
<td>94%</td>
</tr>
<tr>
<td>Matrix-Microbench</td>
<td>100%</td>
<td>86%</td>
<td>85%</td>
</tr>
<tr>
<td>(85% in CS)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>redblack</td>
<td>100%</td>
<td>117%</td>
<td>96%</td>
</tr>
<tr>
<td>patricia</td>
<td>100%</td>
<td>59%</td>
<td>52%</td>
</tr>
<tr>
<td>App.Mix</td>
<td>100%</td>
<td>90%</td>
<td>85%</td>
</tr>
</tbody>
</table>

We also see that the improvements in EDP are more conspicuous in Genome, Matrix-Microbench and Patricia. These applications are characterized by: 1) a high percentage of time spent in critical sections, and 2) a high abort rate. In contrast, the improvements are more constrained whenever the abort rate is small (i.e., Vacation), or the time spent in critical sections is low (i.e., K-means). Note that the reason for the high abort rate in K-means is that the majority of the transactions are executed right after leaving the barriers. Therefore, since the cores start the transactions simultaneously, the likelihood of conflict increases. The Redblack benchmark exhibits an interesting result (Skiplist has been omitted since it exhibits similar characteristics, EDP and abort rates). Since there are no aborts, ideally we would expect
the same EDP results, regardless of the particular contention policy. However, we see that the EDP data for the lazy scheme is about 30% worse. This is due to the overhead introduced in the commit phase, when cores must synchronize for reading/writing the conflict bitmaps. Figure 4.10 quantifies this overhead by comparing the execution cycles of the proposed lazy scheme with an “ideal” version of the lazy scheme that instantaneously pauses all cores and serializes commits. The values in this Figure are relative to eager with no forced serialization. In general, we see that lazy and ideal-lazy show similar results (on average the difference is about 5%), confirming the efficiency of the proposed implementation.

4.3 Summary And Discussion

Like general-purpose systems, today’s embedded systems are adopting multicore architectures. In the medium term, advances in technology will provide increased parallelism, but not increased single-thread performance. System designers and software engineers can no longer rely on increasing clock speed to enable ever more ambitious applications. Instead, they must learn to make effective use of increasing parallelism. Transactional memory is an attractive way to structure concurrent programs.

Nevertheless, transactional memory for embedded systems is different from transactional memory for general-purpose systems. The principal difference is the importance of energy efficiency. While performance remains important, we are no longer willing to consider high-performance algorithms that consume excessive amounts of energy.
We have shown that hardware transactional memory is a viable way to structure synchronization on power-constrained embedded systems, as long as it is designed with energy efficiency in mind. A straightforward implementation of Embedded-TM, using a dedicated, fully-associative transactional cache (as in [33]), was too power-hungry to be applicable in an embedded context, even when it can be selectively powered down. Instead, we found that we could devise a better design by mixing and matching known techniques, such as placing transactional entries in the L1 cache (as in LTM [2], VTM [63] and TCC [31]), backed up by a small victim cache (with the novelty of being powered down when not in use).

Our approach to deal with overflows is however different from LTM [2], VTM [63] and TCC [31]. We propose an efficient yet simple mechanism that does not require any extra hardware structures. In case of overflow, an interrupt is sent to all the other non-overflowing cores. The service routine associated to the interrupt sets the core into a low-power mode, with the overall effect of stopping the execution. Only at the end of the overflowing transaction the other cores may resume executing.

Surprisingly perhaps, some design decisions that individually consume more energy than their simpler alternatives yield overall energy savings. For example, in some cases employing forced serialization of transactions actually results in lower energy than attempting to maximize parallelism. We also show that a small, fully associative victim cache suffices to drastically reduce the number of overflow cases compared to a conventional HTM architecture. Given limited storage, we prefer the TM-victim scheme since it makes the most flexible use of available memory, a critical concern in resource-constrained embedded systems.

As a lazy conflict resolution guarantees more throughput [46], we devised a mechanism that, similarly to FlexTM [68] and and EazyTM [74], detects conflicts at ac-
cess time and defers resolution at commit time. However, as opposed to FlexTM and EazyTM, we propose a method that does not require changes to the CPU (i.e., no new instructions), and to the bus interconnect (i.e., no new core-to-core wires). For heavily conflicting workloads, we present an effective and yet simple method that reduces contention by serializing transactions. As for overflow, the mechanism is entirely SW based, and it does not require any extra bus wires or changes to the CPU.

There are still open questions. Our Embedded-TM design switches to serial mode both for transactions that overflow the hardware cache, and for transactions that repeatedly abort due to data conflicts. Further work is needed to evaluate strategies for switching aborted transactions: should one switch right away, on the grounds that data conflicts probably prevent the current transaction mix from executing concurrently, or is it more sensible to try several times, hoping that the conflicts are transient?

Also, our Embedded-TM design leverages the cache coherency system to detect conflicts. That requires modifications to some of the internal states of the coherency protocol. We would like to remove this constraint, by implementing a mechanism that is decoupled from cache coherency (thus making the design even more applicable). We will show in the next Chapter how we made the conflict detection scheme independent from caches, by implementing a signature-based module that monitors the transactional accesses of the core.
In this Chapter we discuss two optimizations for our Embedded-TM design. They both target fundamental aspects of transactional memory: versioning and conflict detection (see Chapter 2 for details). In particular, we devised a log-based versioning scheme (i.e., the *Transactional-log*) for private data within a transaction, and we proposed a specialized hardware module (i.e., the *Bloom module*) that decouples the conflict detection from the cache coherency.

To reduce the probability of overflow, we investigated a logging solution similar to Log-TM [52]. However, we exploited an important peculiarity of our target architecture: the lack of virtual memory. This fact allows a fundamental optimization: the log-controller can easily identify a memory access that belongs to the private address space with a simple decoding operation. As we mentioned in Chapter 2.2, saving into the log also accesses to shared memory requires longer roll-back times (i.e., an aborted core must restore more entries in memory). Even worse, the other cores must stall for the entire duration of the roll-back operation (since the shared
memory locations must be restored to the original state), with the overall effect of serializing the system. Our proposal, instead, adopts a parallel abort scheme. Since we are logging private addresses, rolling-back is a core-local operation (i.e., the other cores can safely continue executing). Hence, our method still reduces the number of transactional entries to save in the L1 cache, without incurring in any serialization in case of abort. Also, as opposed to LogTM that requires saving an entire line into the log, we propose a method (based on a filter cache) to track single words. Beside the static usage of the “W” bit introduced by LogTM [52], we adopt a more aggressive dynamic solution that prevents logging of obliviously-written memory locations (i.e., private memory locations whose initial content is always re-initialized within the transaction). We will discuss all these policies in more detail in the first part of this Chapter.

In the second part of the Chapter we discuss how we simplified conflict detection, by decoupling our Embedded-TM scheme from cache coherency. We introduce an external signature-based module (i.e., the Bloom module) that is in charge of detecting conflicts among transactions. Note that a similar centralized approach has been proposed in software by Mehrara et al. [47]. However we are the first to propose a full-hardware solution, and explore the performance and energy implications in an embedded context.

5.1 Data Versioning: The **Transactional-log**

In the current transactional scheme, all transactional data is stored on the transactional cache. The transactional cache, however, consumes roughly 30\% of total energy (as we showed in Chapter 4). The transactional cache is also subject to a
large amount of overflows, which can contribute to increased cycles spent in a transaction because the execution is serialized once an overflow occurs. To combat the energy consumption and overflows of the transactional cache, we implemented the transactional logging scheme which logs private transactional data and stores only shared transactional data in the transactional cache.

The transactional log is located in the scratchpad memory. Since the scratchpad memory has a limited size, we also implemented several methods of preventing unnecessary information from being stored in the log. The Static Filter Cache prevents us from saving unnecessary data in the log by saving all addresses that memory writes to. The Dynamic Filter Cache prevents us from saving unnecessary data by saving data that is read and only writing that data to the log if it is written.

5.1.1 Logging - No Filter cache

The traditional logging scheme uses a log stored in the scratch pad memory to store private data. Figure 5.1a) shows the new control flow of the cache controller. For
each transactional write to private memory, the cache controller has to:

1. read the old value from memory
2. update the log with the old value
3. write the new value in memory

The log operates as a stack. Each time we need to save private data, we save the data in the scratch pad memory at a saved index, and increase the index. When using the logging scheme, we only write private transactional data to the log.

The shared transactional data is still written to the transactional cache, as in the original data scheme. We save the original value of the private data to the log only when the data is modified. In case of a commit, we discard the data in the log by resetting the log index to 0. In case of an abort, we write all of the data in the log back to memory, starting with the most recently written private data.

### 5.1.2 Static Filter Cache

Applications that have a large amount of private, transactional data can easily overw ow the scratchpad memory. If an address is already written to the log, however, we do not need to write the address and corresponding data to the log again because we only need an address and its original data in case of an abort. If an application writes to a private address multiple times, we could potentially save space in the log by only saving an address once.

Figure 5.2 shows an address distribution for the genome benchmark. Since
Figure 5.2: Address Distribution for the private writes in Genome

Genome Address Distribution

Average Writes per Transaction

The Filter cache is a cache that only stores addresses that we write to the log; it does not store any data. For that reason, a Filter cache contains many more entries than a equally sized L1 Cache (e.g., the Filter cache has $5 \times$ the number of entries of a Direct Mapped Cache with 16Bytes data lines). Also the line granularity is different: the Filter cache works at word granularity (as opposed to the multi-word granularity of regular L1 caches).

We use the Filter cache to 'Filter' all of the log entries. The overall effect is similar to the usage of the “W” bit proposed in LogTM [52]. However, as discussed in Chapter 2.2, our method is more efficient, since we require saving single words into the log, while LogTM requires saving an entire line. Much fewer cycles are hence wasted. In the next section, we will show a novel and more advanced technique that reduces even further the number of log updates. When we store an address in the log, we also store that address in the Filter cache. Each time we want to update the log with an address and data, we check the Filter cache for that address. If the address is not in the Filter cache then we update the log. If the address is in the
Filter cache we do not update the log. Since the original data is the data that we will save to the log the first time the address is written to, we do not need to save the address and its modified data multiple times. If the transaction commits or aborts, we invalidate all lines in the Filter cache.

5.1.3 Dynamic Filter Cache

The dynamic Filter cache only saves addresses that are read from memory. This method prevents logging of obliviously-written memory locations (i.e., private memory locations whose initial content is always re-initialized within the transaction).

Therefore, we only need to store addresses that have been read and are then written \textit{later on} in the transaction. The scheme for saving addresses that are read and then written to the log is slightly more complicated than the static Filter cache.

In the dynamic filter cache scheme, we use two bits to identify the data stored in a cache line, as shown in Figure 5.3. A dirty bits state of 01 indicates that a line is invalid. A state of 00 indicates that the line has been read, but not written. A
Figure 5.4: The new control flow to handle the Dynamic Filter Cache.

If we read an address, we need to check if the address is in the Filter cache. If it is, then we do not need to do anything else. If the address is not already in the filter cache, we must check to see if the line in the Filter cache that the current data will be replacing contains valid data.

If the line in the Filter cache contains valid data, then we must check the line state. If the line state is 00 we must force update the log: write the current address and data already in the Filter cache to the log before we replace it with the new data. We need to save the data to the log because if the data that we have just evicted is written to and then read again, we will save incorrect data to the log. If the line state is 10, then we can overwrite the line and save the new address in the Filter cache, with state 00. If we write an address, we also need to check if the address is in the Filter cache. If we have a cache hit, then we need to save the address and data to the log and change the line state in the Filter cache to 10. If we have a cache
miss, then we either had to force update the log with the data or we have not read the data, so we do not need to save it.

5.1.4 Experimental Results

In this Section we present some preliminary results. Due to instability issues we encountered in our simulations, we report data only for the Vacation benchmark. A more exhaustive evaluation of hardware and software configurations is left for future work. The adopted system configuration is \textit{TM-victim}, with a small 1KB L1 data cache. The L1 cache is artificially small to emphasize the potential benefit of this technique for larger benchmarks.
We compared two policies:

- **no-log**: the regular TM-victim configuration.
- **txlog-private**: our logging proposal for private data.

As shown in Figure 5.5, the logging scheme greatly reduces the amount of overflows (i.e., in average by 30%). As we mentioned in Chapter 4, reducing overflows increase the probability of conflicts (because more transactions are computing concurrently). We can see this behavior in Figure 5.6.

Figure 5.7 shows the execution cycles. The graph clearly shows that the performance penalty (due to the log updates) of a 1 core configuration is offset by the increased throughput offered by the logging scheme. The same trend is seen for sys-
tem energy (Figure 5.8), where the 1core energy overhead gets reducing when more cores are present in the system.

Finally, Figure 5.9 shows the impact of the filter cache size on the total number of log updates (measured in Bytes). We see that a simple filter cache of 64Bytes (which contains only 16 entries), achieves an hit rate of 63%, which in turns helps reducing significantly (i.e., by 3x) the number of log updates. We expect the dynamic filter cache scheme to further reduce this number. The evaluation of the dynamic filter cache is left for future work.

5.2 Decoupling Conflict Detection from Caches

Hardware transactional memory as discussed in Chapter 4 requires modifications to the MESI protocol because of the addition of a transactional state to the four possible MESI states of cache lines. Each core is responsible for determining when it has a data conflict with another core by snooping the bus and aborting when a bus request conflicts with the status of its own cache. Requiring each core to
handle its own aborts makes it very difficult to exercise any control over the conflict resolution policy used by the system. It is also risky to make any modifications to the transactional memory system, because it is so closely linked with the cache coherence protocol that changes can easily cause coherence to break down.

In this section we present the Bloom module, an external module that maintains a read- and a write-signature for each core in the system, and that detects conflicts by comparing signatures with each other. Because of its separation from the cache coherence protocol of the cores, the Bloom module configuration requires much less complexity in its logic. Next we discuss the implementation details of the proposed architecture.

5.2.1 The Bloom Module

As explained earlier, one of our key design goals was to decouple the transactional memory system from the cache coherency hardware. Signature-based transactional memory architectures have been proposed in the past [15, 81, 67] as an effective solution to disentangle conflict detection from caches. They all share the concept of adopting Bloom filters [9] to keep track of the read and write sets of transactions.

Bloom filters allow two main operations: insertion and membership. Inserting a key to the set results in one or more bits in the filter being set to one. The indices of the bits to flip are given by performing a simple hash function on the key. Checking the membership of a key requires hashing the key and checking if all the corresponding bits are set to one. Bloom filters have the property of potentially producing false positives when checking if a key has been entered into the set, but never false negatives.
Previous works have proposed per-core hardware signatures internal to each core. However, bringing the signatures inside the cores has some drawbacks. There are potential issues in terms of performance (e.g., when broadcasting the signatures at commit time [15]), complexity (e.g., it may be needed to change the bus protocol [67]), and flexibility (e.g., no priority system, and fixed conflict management [81]).

For such reasons, we developed the Bloom module as a single hardware device with access to snoop the shared bus. The physical proximity of the signatures facilitates conflict detection and priority management, while the bus snoop approach uses the existing serialization of the shared bus provided by the bus arbiter.

Note that for the bloom filter module to detect all conflicts, every memory location that a core uses during a transaction must translate into some kind of bus traffic, or else the bloom module will miss conflicts. Because the memory hierarchy satisfies memory requests from cache whenever possible, only using the external memory when necessary, there were some cases that didn’t produce bus traffic that would
alert the bloom module that an address was being used. Reads from addresses that were already in the L1 cache, and writes to addresses that were already exclusive or modified in the L1 cache would not normally require bus accesses. The data would be moved from the L1 cache to the transactional cache and used in the transaction. The solution was to add extra bus accesses for these cases. Since the cache already had the required data, these extra bus accesses were pure overhead. We will see in Section 5.2.3 what impact they have on performance.

The internal details of the Bloom module are shown in Figure 5.10. The design consists of four main components: the control logic, the memory mapped registers, the Bloom Filter Units (i.e., BFUs), and the hash functions.

The control logic is responsible among other things for forcing the commit priority, and for coordinating the cores in case of special events (e.g., overflows, aborts). Note that the handshaking between the cores and Bloom module is entirely managed with commodity resources (e.g., interrupts and read/write memory operations), and no extra wires are required.

Some of the functionalities of the Bloom module (e.g., commit priority) can be programmed at run time with simple writes on its memory mapped registers. The memory space reserved for programming the registers is quite small (only 256Bytes).

As shown in Figure 5.11, each core has a BFU that consists of K read-write pairs of simple Bloom filters. In our case K=4, since empirically we found that number to give the best power/performance tradeoff. To limit the hardware requirements, we adopted a parallel Bloom filter design [64], which sets a single bit in K small bloom filters, instead of setting multiple bits in one big filter.
A signature is updated as follows. When a core writes to some address within a transaction, the address is hashed in K different ways, the R/W is set to W for write, and Lookup/Insert is set to Insert. This causes a single bit in each of the write filters to be set to 1. (If a given bit is already 1, it remains 1 until the clear signal, CLR, is asserted). The bit that is set is selected by the corresponding hashed address.

The hashed addresses also go to the BFUs of the other cores. For all other BFUs, the R/W is set to Read and the Lookup/Insert is set to Lookup. The corresponding bits in the K read filters for each BFU are checked to see if they are already 1. If all K of the bits checked in a given BFU are found to be 1, a conflict is possible and is indicated on the output. Depending on the conflict resolution scheme, one or more transactions may need to be aborted.

Finally we designed the hash functions with delay and power as major concerns. An ideal hash function should be designed for small fan-out, small fan-in, and use few levels of gates to keep the critical path short. Because of the peculiarities of our target system (no OS, no virtual memory, and fixed shared address space) the addresses added to the Bloom filter are expected to show some spatial locality. As a
result, useful hashing can focus on the lower order bits of the addresses. In particular, we decided to model the hash function as a single level of two input XORing of lower order address bits.

### 5.2.2 Flexible Conflict Management

An additional benefit of the centralized nature of the bloom filter module is that it can use any algorithm to decide which cores to abort. It can give priority to the transaction that has been running the longest, the transaction that has been aborted the most times previously, or just the transaction with the lowest CPU number, if that is what turns out to be most effective. This kind of flexibility should make it relatively simple to implement features like lazy conflict resolution, in which conflicts are detected when they occur but aborts don’t occur until cores try to commit. Additionally, it makes it possible to add a very useful feature: the ability to impose an ordering on commits. In this scheme, a core that tries to commit will be forced to wait for other transactions to commit if necessary to carry out a pre-programmed order.

Ordered commits are much easier to implement with the addition of the bloom filter module, because it acts as an authority that possesses information about all the commits that occur. Cores performing ordered transactions wait for an acknowledgment message from the bloom filter module, and the bloom filter module imposes the correct ordering on these acknowledgments, so the transactions commit in the desired order. We will show in Chapter 6 that ordering is the key feature that allows speculative parallelization of loops, even if it were not certain that the loop iterations were free of dependencies on one another.
5.2.3 Performance of the Bloom Module

We tested our architecture with several hardware and software configurations. For convenience, we report the hardware parameters in Table 5.1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Configuration(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>ARMv7, 3-stage in-order pipeline, 200Mhz</td>
</tr>
<tr>
<td>L1 cache</td>
<td>8KB 1-way Icache, 16KB 4-way Dcache</td>
</tr>
<tr>
<td>Cores</td>
<td>{1, 2, 4, 8}</td>
</tr>
<tr>
<td>Policies</td>
<td>Bloom-module, Locking</td>
</tr>
<tr>
<td>Signature</td>
<td>{2KBits 4-way} Read and Write filters</td>
</tr>
</tbody>
</table>

Table 5.1: Hardware configurations.

We adopted two system policies:

- **Locking.** The core must acquire a lock (implemented as a binary hardware semaphore) before entering a critical section. The level of granularity (ie coarse vs fine) depends on the specific application. Note that operations on locks are done efficiently through low-level primitives (i.e., no OS function calls).

- **Bloom-module.** Our proposed Embedded-TM architecture with the centralized Bloom module. The size of the Read and Write signatures contained in each BFU is 2Kbits.

We chose a representative subset of applications from the STAMP benchmark suite [50]. The workloads of these applications can be categorized according to their synchronization pattern and the length of their transactions:

1. large non-conflicting transactions (Vacation)

2. barrier-based synchronization with small transactions (Kmeans)
Figure 5.12: Execution Cycles of the STAMP benchmarks, for the Locking and Bloom-module configurations. All the values are relative to a 1core Locking configuration.

In our experiments we collected both the execution cycles and the energy of each component. For the Bloom module we only modeled the energy of the signatures, since they constitute the main source of energy consumption. Future work will also include the energy spent by the control logic and the hash functions.

3. Large transactions (Genome)

4. mix of short and large transactions (Labyrinth)

In our experiments we collected both the execution cycles and the energy of each component. For the Bloom module we only modeled the energy of the signatures, since they constitute the main source of energy consumption. Future work will also include the energy spent by the control logic and the hash functions.

Figure 5.12 shows the performance of the Embedded-TM architecture, compared to a Locking scheme. The values are relative to a 1core Locking scheme. We see that for almost all benchmarks, Embedded-TM achieves much better results than Locking. The only exception is Kmeans, where Embedded-TM performs the same as Locking. This is expected, since in Kmeans the cores synchronize with barriers, and only 5% of the time are executing transactions. Also, Embedded-TM is penalized with a 1core configuration, because of the extra bus accesses required to updated the Bloom module (as explained in Section 5.2.1).

Figure 5.13 shows the energy distributions for both Locking and Embedded-TM.
Figure 5.13: Energy Distribution of the STAMP benchmarks for the Locking and Bloom-module configurations for different number of cores. The values are relative to the 1core Locking configuration.

System Energy
(normalized to 1core locking)
All the values are relative to a 1core Locking configuration. In general, Embedded-TM gives better energy results than Locking. Same as for performance, Embedded-TM results also penalized in terms of energy in a 1core configuration by the extra bus accesses. Also, note that the higher the number of cores, the greater is the weight of the Bloom module on the overall energy consumption. This is expected, since the number of hashing operations grows linearly with the number of cores.

In the worst case (i.e., with 8 cores) the Bloom module consumes about 5% of the system energy, as shown in Figure 5.14.

**Figure 5.14:** Energy Distribution of the Vacation benchmark for the Bloom-module configuration with 8 cores.

### 5.3 Summary and Discussion

In this Chapter we presented two design optimizations targeting data versioning and conflict detection. We introduced the “Transactional-log” scheme along with its Filter Cache variants. Our preliminary results show that the logging scheme can greatly reduce the amount of overflows for the TM-vanilla configuration. However, there is an increase in abort rate for all benchmarks, since the longer a transaction lasts, the greater chance it has of being interrupted by another transaction. The impact on
execution time using this scheme is a bit mixed; while memory requirements may be reduced, there may increased reads to memory due to log updates.

We introduced another important optimization which greatly simplifies the conflict detection scheme. Detecting data conflicts is one of the most critical and complex aspects of any transactional memory solution. For example, many hardware transactional memory systems require important changes to the existing cache coherency protocol, which in turns puts severe limitations to their applicability. Modifying the coherency protocol, in fact, affects both the timing and the testability of the design. For such reasons, we devised the “Bloom module”, an external signature-based module that 1) monitors all the transactional accesses and saves them in per-core signatures, and 2) notifies the CPUs in case of data conflicts.

We designed the Bloom module to be programmable with simple writes to its internal registers. Exposing the Bloom module to software was the key that allowed the integration of the Hardware TM system with the compiler. In the next Chapter we present our complete HW/SW framework for transactional programming for MPSoCs, and show compiler extensions for speculative loop-level parallelization of applications.
Chapter 6

Integrating HTM with Compiler:
The SoC-TM Approach

In the previous Chapters we studied design methods to reduce the complexity of the TM system. In fact, we believe practical TM design for embedded systems must emphasize simplicity; complex hardware designs that require extensive changes to established protocols (i.e., cache coherency), will likely be too costly to adapt.

Similarly, TM must be integrated into practical and familiar programming environments, with simple programming abstractions, in order to easily use it in an embedded domain. A first attempt in this direction was made with the OpenTM programming framework, which extends the popular OpenMP API for shared-memory systems with compiler directives to express TM programming patterns [5].

In this Chapter we present SoC-TM, an integrated HW/SW solution for transactional programming on embedded MPSoCs. At the heart of our proposal sits a modified Embedded-TM (see Chapter 4) version with attached the Bloom module
(described in Section 5.2.1). While other works have also proposed the use of Bloom filters as a means of managing conflicts among transactions (e.g., [67]), what we propose is a centralized unit that is fully implemented in hardware. This approach has the advantage of being very lightweight and fast, requiring minimal changes to the cache coherency protocol logic, and removing the need for each core to make individual decisions on transaction conflict management. As with [67], it also allows for more flexibility in determining how to manage conflicts.

6.1 Motivation

Ease of use and transparency of the internal functionality are first-class design constraints of the SoC-TM system. As such, application developers are not meant to cope directly with low-level transactional programming. Instead, in our system transactional features are triggered through the use of a custom set of compiler directives, implemented as an extension to the popular OpenMP programming model. Our enhanced compiler transforms the annotated program so as to interact with our runtime system, where low-level APIs are transparently used. To further improve ease of programming, our framework supports speculative task- and data-level parallelism. Specifically, loops which may carry cross-iteration dependencies (non-DOALL) can be annotated as parallel loops. Similarly, tasks which may possibly be inter-dependent can be annotated for parallel execution. The underlying TM system ensures that, in case a real dependence arises, the original sequential program semantics is preserved. In our SoC-TM system this can be achieved by forcing transactions to commit in program order, thanks to specific hardware support for prioritized commit that we provide. At the program level, this feature can be leveraged by the OpenMP worksharing constructs (parallel sections and parallel
for), to generate parallel transactions holding speculative workloads (tasks or loop iterations). The compiler properly generates code that programs the Bloom module to commit speculative transactions in order.

6.2 System Overview

6.2.1 Architectural Template

To scale to many-core systems embedded MPSoCs are increasingly relying on a design paradigm where a general purpose host processor controls a on-chip GPU-like accelerator, made of tens to hundreds of very simple homogeneous cores [61] [56]. Our focus in this paper is on the host processor subsystem. Current designs feature a dual-core system. Quad-cores are underway, and next-generation products will be based on eight-core designs.

Fig. 6.1 depicts the block diagram of the target platform considered in this work.
It features a configurable number (up to 8) of RISC-like cores, interconnected through a shared bus (AMBA). Each core has private L1 instruction and data caches, the latter being kept globally coherent through per-core snoop devices which implement a MESI coherence protocol. The shared memory is organized as a two-level hierarchy, adhering to the Partitioned Global Address Space (PGAS) paradigm. More specifically, our MPSoC features several distinct physical memory banks, globally visible throughout the system. Each core has a small L1 local scratchpad (SPM), which can be accessed without traversing the system interconnect. Remote SPMs can also be accessed directly, but corresponding transactions travel through the bus, and thus are subject to higher latencies. The overall L1 shared memory can be seen as the sum of all SPMs, and is globally non-coherent. This means that its addresses are not cacheable, and it is explicitly managed by software. We use L1 shared memory to optimize the implementation of runtime services (e.g., thread scheduling, barrier synchronization), but expert programmers can also explicitly use it to optimize critical parts of their applications.

L2 shared memory physically consists of a single device, but we logically partition it in a big, shared segment, plus small "private" segments to each core. Addresses belonging to the logically shared chunk are cacheable and globally coherent. "Private" segments are also cacheable, but their addresses are not involved in coherency traffic.

Explicitly separating shared data from private data is done automatically from our compiler (Sec. 6.4), and has the advantage of making conflict management in the Bloom module much lighter (Sec. 6.3).
6.3 Hardware TM Design

In this Section we will describe the details of our Transactional Memory Architecture. Our solution brings two key features. First, to meet the target requirement of low complexity, and to maximize design reusability, we devised a transactional memory architecture that didn’t involve any modifications to the hardware of the CPU. Neither custom instructions, nor changes to the pipeline are in fact needed to handle transactional events. As we will explain later, our approach is based exclusively on software-triggered operations on memory mapped registers.

The second key feature is the clear separation of conflict detection from caches. Detecting data conflicts is one of the most critical and complex aspects of any transactional memory solution. For example, many hardware transactional memory systems require important changes to the existing cache coherency protocol, which in turns puts severe limitations to their applicability. Modifying the coherency protocol, in fact, affects both the timing and the testability of the design. For such reasons, we devised the “Bloom module” (described in Chapter 5.2.1), an external signature-based module that

1. monitors all the transactional accesses and saves them in per-core signatures

2. notifies the CPUs in case of data conflicts.

Figure 6.2 gives an overview of the proposed transactional architecture. The dark blocks represent the only hardware changes required by our solution. In particular, we can define three main components:

1. A new state bit (i.e., the $Tx$ bit) for each line of the Data Cache, which defines
Figure 6.2: Transaction management is triggered by regular read/write operations on memory-mapped registers of the bloom-module. The dark blocks represent the extra logic/registers reserved to the transactional execution

whether the data contained in the line is transactional or not.

2. Some new logic in the Cache Controller that handles the new transactional accesses.

3. The external Bloom module.

As we mentioned earlier, all the transactional events are triggered by regular read/write operations on memory mapped registers. For example, to start a transaction a core has to write to a particular register in the Bloom module. The cache controller detects that write operation, and sets an internal bit that enables the transactional logic.

When executing transactional, the transactional logic of the cache controller will eventually perform a few extra steps, as shown in Figure 6.3. Beside setting the Tx bit in the cache line, the cache controller must also manage two special cases: 1) carrying out an extra bus access (to notify the Bloom module) in case the line was already present in the cache before starting the transaction, and 2) performing a
transaction-overflow when the line to replace is already transactional. The adopted protocol requires the overflowing core to notify the Bloom module by writing in one of its registers. The Bloom module will intercept that write, and set the requesting Core_ID in a overflow request queue (to deal with concurrent requests). The Bloom module will process the overflow request by resetting the entry from the request queue, and by sending an Overflow Interrupt Request (i.e., Overflow-IRQ) to the other (non overflowing) cores. Interrupted cores enter an idle mode until, at the end of the overflowing transaction, the Bloom module sends a Wakeup-IRQ to restart their execution, with the overall effect of serializing transaction execution.

### 6.3.1 Hardware Support for Ordered Commits

Auto-parallelizing compilers and directive-based parallel programming models (such as OpenMP) often focus on regular data-intensive loops with no cross-iteration
dependencies (DOALL). Speculative loop parallelization and thread-level speculation [85] [60] have been proposed in the past to effectively extract parallelism from programs whose execution cycles are mostly spent inside non DOALL loops or inside possibly inter-dependent tasks. In particular, some approaches based on TM and ordered commits have been proposed in the general purpose domain [47] [76]. Several embedded applications exhibit a similar pattern [37], thus making support for speculation very valuable in this domain. However, previous TM-based approaches are unlikely to be beneficial for embedded systems, due to non-negligible software (and hardware) complexity, which induces high overhead, and thus also high energy consumption.

To enable speculative parallelization of target applications we enhance our Bloom module design with the capability of ensuring a predetermined commit order for concurrent transactions. Having this feature allows to optimistically partition the program workload among parallel threads even when dependencies may be present. To support this mechanism our TM system should be capable of imposing a commit order that respects the original sequential program order. Then, in case a conflict is detected, we need a mechanism to undo the effects on memory for the transaction that is trying to commit out of order. Note that for our design, we use a lazy-versioning, eager-conflict detection policy, which does not require to undo the effects of aborted transactions (all changes to shared data take place in the cache, and are only visible to shared memory upon commit).

Most parallel programming models provide facilities to schedule available workload among threads statically or dynamically. Parallel workload may be represented as a set of concurrent tasks. Statically assigning tasks to cores may result in better locality and improved predictability. Dynamically distributing tasks is preferred to solve load imbalancing issues. It is therefore important to support both schemes in
SoC-TM, where each task is wrapped within a transaction.

With dynamic scheduling is not possible to determine in advance which core will take ownership of a transaction. Moreover, since the number of co-existing transactions in the system at any time\(^1\) is bounded by the number of cores \(N\), their priority can always be expressed as a number in the range \([0..N−1]\) through a modulo operation

\[
Pri = TxID\%N
\]

Due to these considerations, to support ordered commits in case of dynamically scheduled workloads we consider a circular buffer of \(N\) slots, each associated to a specific priority level. We use two pointers to mark the “next available” slot in the buffer and the “next to commit” transaction. Every time that a core queries the system for a new transaction the physical ID of that core is inserted in the next available slot of the buffer, and the corresponding pointer is atomically shifted one slot ahead. This allows to keep at any instant in time a record of the order in

\(^1\)We consider a \textit{run-to-completion} thread model, where no preemption is allowed.
which cores can commit their transactions. Upon reaching the commit point in a
transaction every core invokes the Bloom module. The core whose ID is stored in the
slot pointed by the “next to commit” pointer is allowed to complete its transaction,
while the others are stalled. The pointer is eventually shifted to the next element
in the buffer. In the example in Fig. 6.4 we consider \( N = 8 \). Color codes (shades
of blue) are used to indicate priorities (the darker, the less priority). Slots of the
circular buffer are thus labeled with transaction IDs whose priority level decreases
clockwise. Due to dynamic workload scheduling transaction dispatching is serialized
and transactions are not tied to a specific core. At instant \( t_1 \) transactions 0 to 6 have
been dispatched, and the “next available” pointer points to the seventh slot, where
the ID of core 1 is to be annotated shortly afterwards. Transactions 0 to 2 have
already committed, so the “next to commit” pointer points to slot 3. At instant \( t_2 \)
core 7 will try to commit transaction 4, but the Bloom module will stall its execution
until the point where core 0 will commit transaction 3.

With static scheduling, on the contrary, the set of transactions that each core
will be in charge of processing is fixed. Each core (i.e. each of the threads composing
the parallel team) computes locally the ID of the next transaction to be processed,
based on the number of available cores and on its own physical ID. For this reason,
to support ordered commit in this scenario we build a different buffering mechanism,
with core-specific registers where the ID of the transaction being executed is stored.
After a core computes the ID of the next iteration to process, it communicates
this value to the Bloom module, which annotates it in the register associated to
that core. A separate register is used to store the ID of the transaction that “last
committed”. Upon arrival onto a commit point each transaction queries the Bloom
module, which will only allow the transaction whose ID equals the content of the
“last committed” register +1 to commit, and stall the others. In Fig. 6.5 we show
Figure 6.5: Design of per-core priority buffers to support ordered commits of statically scheduled transactional workload

an example with $N = 8$. The same color codes described previously apply to this example. At instant $t_0$ (upper part of the figure) all threads have independently (i.e. in parallel) computed the ID of their next transaction, and the Bloom module has been programmed so that each core’s register holds this ID. The “last committed” register contains the value $-1$, so that when core 0 tries to commit transaction 0 it succeeds without further delays. At instant $t_1$ (lower part of the figure) transactions 0 to 2 have committed, and cores 0 to 2 have started executing transactions 8 to 10, whose IDs are now stored in the respective registers. Core 4 tries to commit transaction 4, but the “last committed” register reads 2, so the core will be delayed until core $2+1 = 3$ commits first. Similarly, when at instant $t_2$ core 0 tries to commit transaction 8 it is stalled until core 7 has committed transaction 7.
### Table 6.1: Low-level Transactional API (LLTA)

<table>
<thead>
<tr>
<th>Function name</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void begin_transaction ()</td>
<td>Marks the beginning of a transaction by enabling conflict management on shared memory accesses</td>
</tr>
<tr>
<td>void end_transaction ()</td>
<td>Marks the end of a transaction by disabling conflict management on shared memory accesses</td>
</tr>
<tr>
<td>void force_abort ()</td>
<td>Forces the current transaction to abort</td>
</tr>
<tr>
<td>int check_abort ()</td>
<td>Returns 1 if the current transaction was aborted, 0 otherwise</td>
</tr>
<tr>
<td>void bloom_max_retries (int n)</td>
<td>Sets the maximum number of times a single transaction can be aborted before switching to serial mode</td>
</tr>
<tr>
<td>void update_thread_priority (int pri)</td>
<td>Sets current thread’s commit priority</td>
</tr>
<tr>
<td>void start_ordered_transaction ()</td>
<td>Marks the beginning of an ordered transaction by enabling conflict management on shared memory accesses. Prevents current transaction from committing earlier than transactions with higher priority</td>
</tr>
<tr>
<td>void end_ordered_transaction ()</td>
<td>Marks the end of an ordered transaction by disabling conflict management on shared memory accesses and ordered commits</td>
</tr>
</tbody>
</table>
6.4 Software TM Support

As explained in the previous section our Bloom module can be programmed via software by reading and writing into memory mapped registers. We developed a small set of low-level primitives which expose to the software a practical way of accomplishing this task. The functions of this transactional API are briefly described in Table 6.1.

However, dealing with transactional behavior of the application at this very low level of abstraction would require extensive manual modification of the source code, which is a tedious and error-prone task, hindering the ease of use of such a programming abstraction. Higher level programming abstractions are desirable for the sake of simplifying embedded application development.

OpenMP is a well-recognized standard for shared memory parallelism, which provides an easy-to-use incremental approach to code parallelization based on compiler directives. Due to its appealing features and to the lack of a standard programming methodology for MPSoCs OpenMP has been proposed several times as a convenient programming paradigm for these machines as well [16] [57] [35] [42] [45]. Specific extensions to the standard programming interface have been proposed by researchers to efficiently leveraging heterogeneous computing resources [57], accelerators [42] or memory hierarchies [45]. Similar to those approaches, specific extensions to OpenMP can be designed for transactional programming as well. We are not the first to consider this option. In the general-purpose computing domain similar solutions have seen the light in the past few years, such as OpenTM from Stanford [5] or a similar proposal from the Barcelona Supercomputing Center [49].

OpenTM [5] specifies a small set of extensions to the standard OpenMP 2.5 direc-
tives that enable programmers to wrap critical sections in their codes within transactions. There are three basic means to specify transactional workload in OpenTM:

The boundaries of a transaction can be explicitly marked by enclosing a code region within a `transaction` directive.

```c
#pragma omp transaction [clause[,clause]...]
    structured-block
```

Transactions can also be implicitly outlined by transactifying the workload generated by worksharing directives. (Groups of) loop iterations can be enclosed within a transaction through the `transfor` directive:

```c
#pragma omp transfor schedule(type, chunk)
for (i=LB; i<UB; i = i OP stride)
    structured-block
```

This construct allows to group `chunk` iterations of the target loop within a single transaction, scheduled according to the policy specified by `type` (static, dynamic, guided).

Finally, task parallelism can also be transactified with the `transsections` directive:

```c
#pragma omp transsections
[#pragma omp transsection]
    structured-block-1
[pragma omp transsection]
    structured-block-2
...
A prototype implementation of the OpenTM directives was realized in the GCC 4.3.0 compiler, and the authors made the source code publicly available. The OpenTM compiler is capable of generating code which targets generic STM and HTM systems by emitting calls into custom runtime library functions which can be seen as generic primitives providing basic services for transactional programming. Their implementation can be specialized for a different target by leveraging platform-specific low-level facilities. Due to these considerations we decided to use the OpenTM programming interface and compiler as a starting point.

However, adapting the OpenTM framework to our MPSoC requires several modifications. First, the runtime library has been re-designed to cope with hardware and software limitations (memory space constraints, absence of MMU and OS support for virtual memory). For performance reasons thread management does not rely on standard threading libraries, but directly controls hardware resources, which makes it extremely fast and lightweight. Barrier synchronization is also made very cheap by leveraging L1 SPMs, thus avoiding bus congestion due to busy waiting.

We also introduce a few modifications to the compiler to take advantage of the peculiarities of the memory subsystem of our MPSoC. One of the performance limiters of HTM systems is the difficulty of distinguishing between private and shared data [62, 82, 65], which leads to increased coherence traffic and false sharing issues. The OpenMP programming model requires that all data items referenced within a parallel region are explicitly marked as shared or private.

```
#pragma omp parallel shared(a) private(i)
```

The compiler can thus leverage this information to automatically convey the allocation of private and shared variables onto distinct regions of the address space. Shared
data is mapped onto the address segment which is under control of the Bloom module, whereas private data reside on those “private” regions on the L2 shared memory which are not managed transactionally. This disambiguates addresses and allows a lighter and faster conflict management.

Due to the cache-line granularity of our TM system, another issue which hinders performance is false sharing. If two different shared data items, accessed from within two transactions, reside on the same cache line, an abort will be issued even if no real conflict takes place. This phenomenon can easily arise if data resides at addresses which are not aligned to cache line boundaries.

Let us consider the example in Fig. 6.6. Here a loop of 8 iterations initializes as

```c
int a[8];

#pragma omp parallel shared(a) num_threads(2)
#pragma omp transfor schedule (static,4)
for (i=0; i<8; i++)
{
   a[i] = i;
}
```

![False sharing due to misaligned data](image)

**Figure 6.6**: False sharing due to misaligned data

many elements of the array `a`, and the workload is parallelized among two threads, each accessing 4 elements. However, since the array `a` is allocated at an address
which is not aligned to a cache line size, the fourth element of each chunk resides in
a different cache line. All accesses to array a from both transactions will result in a
false conflict. In this example, aligning the variable a to the first address of a cache
line removes the problem.

Since extensive data alignment may lead to memory wastage, rather than au-
tomatically and silently aligning all OpenMP shared objects from our compiler we
provide a directive to allow programmers to control alignment of key data items.
The custom aligned qualifier can be used instead of the shared clause to achieve
this goal.

```plaintext
#pragma omp parallel aligned(a)
```

### 6.4.1 Extensions for Ordered Commits

OpenMP provides a ordered construct which specifies a structured block in a loop
region which will be executed in the order of the loop iterations. This sequentializes
and orders the code within an ordered region while allowing code outside the region
to run in parallel.
In the code snippet above statements S1 and S2 from different loop iterations can be executed in any order, and thus are allowed to run in parallel. On the contrary, the statement protected by the #pragma omp ordered construct must be executed following the sequential loop order. Similar loops where each iteration carries a data dependence with another iteration at a precise distance can be parallelized by synchronizing threads so that the dependent iteration (consumer) waits for its predecessor (producer) to update the shared data element before accessing it (DOACROSS).

However, more in general loops could carry dependences which only manifest themselves rarely, while most iterations can be executed fully in parallel. In this case enclosing the loop body – or just the statement(s) possibly carrying the dependence – within a transaction and ensuring that transactions commit following the sequential program order ensures that the original loop semantics are preserved.

Authors of OpenTM suggest that the ordered clause in OpenMP can be coupled to transactional variants of worksharing constructs (transfor and transsections). From the programming language point of view this would provide a convenient abstraction to enable speculative code parallelization. However, implementing a similar
OpenMP Program

```c
#pragma omp parallel
#pragma omp target schedule (static, CHUNK) ordered
for (i=GLB; i<GUB; i+=INCR)
{  LOOP_BODY (i);  }
```

Transformed Program (after OpenMP expansion)

```c
int LLB, LUB, i;
int nthreads = omp_get_num_threads ();
int tid = omp_get_thread_num ();

NEXT_CHUNK:
LLB = f(GLB, GUB, INCR, CHUNK, nthreads, tid);
LUB = min (LB + CHUNK, GUB);

if (LLB<GUB)
{
  int TX_ID = LLB/CHUNK;
  __builtin_omp_set_commit_order (TX_ID);
  __builtin_omp_begin_ordered_transaction ();
  for (i=LLB; i<LUB; i++)
  {
    LOOP_BODY (i);
  }
  __builtin_omp_end_ordered_transaction();
  goto NEXT_CHUNK;
}
```

Figure 6.7: Speculative loop parallelization with static scheduling

semantics require the underlying TM system to be capable of ensuring transaction ordering.

As discussed in Sec. 6.3.1, our Bloom module has been designed with this requirement in mind, and the order in which transactions can commit can be dynamically programmed through the functions of the LLTA (see Tab. 6.1). In Figures 6.7 and 6.8 we show how to annotate a loop for speculative parallelization
with static and dynamic scheduling, respectively, and how our compiler transforms it. We show in black the standard loop transformations made by the OpenMP compiler, in red and green the modifications that we introduced to the `transform` and `ordered` expansion code respectively. When a `CHUNK` size is provided to the `schedule` clause, OpenMP transforms loops in such a way that `CHUNK` consecutive iterations are executed within inner loop (hereafter called the `chunk loop`) which lower and upper bounds (LLB, LUB) are computed locally. If the schedule type is `static`, these bounds are computed based on the thread ID, the total number of threads and other parameters (e.g. `CHUNK` size, loop stride, global lower and upper bounds). The chunk loop is wrapped within a transaction by marking its beginning and end with calls to the runtime functions `omp_begin_ordered_transaction` and `omp_end_ordered_transaction`. Within these functions the corresponding LLTA primitives are finally invoked. Before initiating a transaction, its global ID is computed within the thread (TX_ID), and is passed to the `omp_set_commit_order` runtime function, which notifies the Bloom module about which core is currently executing the transaction by invoking the `update_thread_priority` LLTA primitive.

If the schedule type is dynamic, lower and upper bounds for the chunk loop are retrieved by invoking the iteration scheduler through calls to the runtime environment. We have implemented custom extensions to these scheduling functions which augment the standard scheduler with the capability of programming the Bloom module to ensure that transactions commit in the program order. As shown in Fig. 6.8 our functions are called `omp_dynamic_loop_ordered_start` and `omp_dynamic_loop_ordered_next`, and from the latter the `update_thread_priority` LLTA primitive is invoked.
6.5 Experimental Results for Speculative parallelization

In this section we describe our experiments with speculative parallelization of applications. We first consider a synthetic benchmark designed to explore the behavior of
our Soc-TM when ordered commits are imposed on a loop carrying cross-iteration dependences. This benchmark consists of a loop containing a certain amount of fully parallel “dummy” work (statement S1) – which can be executed in any order by any thread – plus an instruction (statement S2) carrying a forward dependence. S1 is also independent of S2.

```c
for (i=0; i<WORK; i++)
{
    S1:  DUMMY_WORK (w);
    S2:  a[i] = a[i + dep_dist];
}
```

To model varying amounts of independent work we consider different values of a parameter $w = \{0, 10, 100\}$, passed as an argument to the function `DUMMY_WORK`, called in statement S1. $w = 0$ indicates that the loop only contains the instruction creating the dependence (S2), $w = N(>0)$ indicates that the independent work in the loop accounts for (roughly) $N$ times the execution time spent in dependent computation (i.e. $\text{time}(S1)/\text{time}(S2) = N$).

The cross-iteration dependence has a parameterized distance $\text{dep}_\text{dist} = \{0, 8\}$. We consider $\{\text{dep}_\text{dist} = 0, w = 0\}$ to be a worst case for our system, since there is almost no parallelism among instructions: each iteration depends on the previous, and there is no independent work in the loop which can be executed in parallel. For larger values of $w$ we expect to be able to extract more and more parallelism (DOACROSS). Intuitively, when $\text{dep}_\text{dist}$ is bigger than the number of cores it can not happen that two threads holding inter-dependent iterations are executing concurrently, and thus the loop should execute fully parallel (DOALL). We thus also consider $\text{dep}_\text{dist} = 8$ (where 8 is the maximum number of cores in our system).
The results for this experiment are shown in Fig. 6.9, where we compare the scaling of dynamic speculative loop parallelization using both software and hardware iteration scheduling, and considering cache-aligned and non cache-aligned data.

![Graphs showing results for synthetic benchmark](image)

**Figure 6.9:** Results for synthetic benchmark

First of all, it is possible to notice that using hardware iteration scheduling is extremely beneficial when the loop body contains small amounts of work. As expected, when \( \{\text{dep dist} = 0, w = 0\} \) the system only scales up to 2 cores, then there is not enough parallelism and frequent aborts take place, due to dependence violations. Clearly this effect is mitigated when \( w \) increases. For \( w = 10 \) the system scales up to 4 cores, and with \( w = 100 \) it perfectly scales up to 8 cores. When \( \text{dep dist} = 8 \), as expected, the aborts due to cross-iteration dependence violation almost completely...
disappear. However we still see aborts due to false conflicts implied by bad data alignment. Annotating shared data with our aligned clause allows much better scaling even for small-medium values of $w$.

Besides this synthetic benchmark we evaluate our support to speculative parallelization with six real benchmarks, extracted from two representative benchmark suites for embedded systems, namely MiBench [29] and EEMBC [22]. The target applications were selected because most of their execution time is spent inside non DOALL loops [37], namely inside loops with cross-iteration dependencies (data-intensive or functional loops).

Our approach to parallelizing these application was simply that of annotating these loops with our custom directives for ordered commits. Fig. 6.10 shows the obtained results in terms of execution time speedup when the number of cores increases (from 1 to 8). Each plot compares (on the Y axis) the performance of the HW and SW scheduler, and the effect of aligning data to cache line boundaries.

For programs adpcm and CRC32 each transaction executes very few or no instructions besides the one(s) carrying the dependence. Thus, similar to what we noticed for the synthetic benchmark with WR=1 there is no parallelism to be extracted among threads. In this case it is important however to point out that our system has a very lightweight management of conflicts, as the execution time does not increase with an increasing number of processors even if the degree of contention does.

For these two benchmarks and for dijkstra it is also possible to notice a significant benefit introduced by the use of the hardware scheduler. Indeed, due to the
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Cache Alignment</th>
<th>Hardware Scheduler</th>
<th>Software Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC32 (MiBench)</td>
<td>Non cache aligned</td>
<td>0.2</td>
<td>0.4</td>
</tr>
<tr>
<td>adpcm (MiBench)</td>
<td>Non cache aligned</td>
<td>0.2</td>
<td>0.4</td>
</tr>
<tr>
<td>dijkstra (MiBench)</td>
<td>Non cache aligned</td>
<td>0.2</td>
<td>0.4</td>
</tr>
<tr>
<td>patricia (MiBench)</td>
<td>Non cache aligned</td>
<td>0.2</td>
<td>0.4</td>
</tr>
<tr>
<td>rotate (EEMBC)</td>
<td>Non cache aligned</td>
<td>0.2</td>
<td>0.4</td>
</tr>
<tr>
<td>rgbhpg01 (EEMBC)</td>
<td>Non cache aligned</td>
<td>0.2</td>
<td>0.4</td>
</tr>
</tbody>
</table>

Figure 6.10: Results for speculative parallelization
small amount of work within each transaction, the cost for invoking the software scheduler has a non-negligible impact on performance.

\textit{dijkstra} carries a different dependence pattern, in that the dependence may manifest or not depending on the input set. This ultimately leads to rare dependence violations at runtime, which allow an almost perfect scaling. It is also possible to notice that if data is not cache-aligned a higher number of aborts happens, due to false conflicts.

Finally, programs \textit{rotate} and \textit{rgbhpg01} behave as DOALL loops once false conflicts due to poor data alignment are removed with our \texttt{aligned} clause, which leads to a perfect scaling. The difference between SW and HW instruction scheduling is completely negligible, since transactions contain amounts of work large enough to amortize the overhead.

\section{Summary and Discussion}

In this Chapter we have presented \textit{SoC-TM}, a vertically integrated HW-SW framework for transactional programming on embedded MPSoCs.

For the hardware system, we further simplified the design of Embedded-TM (presented in Chapter 4), by decoupling conflict detection from cache coherency. We showed how we integrated the Bloom module (presented in Chapter 5) in our system.

To ease application development – which is of the utmost importance in this domain – we extended the lightweight yet efficient HTM design, with the simplicity of directive-based programming of OpenMP.
Speculative parallelization is also supported, which makes the task of extracting parallelism from sequential codes less cumbersome. Our results confirm the effectiveness of SoC-TM, concerning ease of use, performance, simplicity of design and energy consumption.

This work can be extended in several directions. Compiler analysis can be leveraged in the auto-parallelization pass in GCC to determine candidate loops for speculative parallelization by estimating the independent work within each transaction. Moreover, our programming model can be easily extended to incorporate the latest (3.0) OpenMP specification. Here the task construct enables very fine grained task parallelism, useful to build very flexible support for TLS. Also, a distributed version of the Bloom module could improve the scalability of the design, and potentially reduce the energy and performance penalty associated to the broadcasting bus operations.
Chapter 7

Conclusions

In this thesis we devised energy-aware optimizations for two different synchronization techniques for multi-core embedded systems.

In Chapter 3 we introduced *DVS-sleep* lock, a hybrid (wait/sleep) semaphore in which the low-power state is entered only after certain time-out period. Our experiments provided two main conclusions. First, we showed that the proposed timeout-based semaphore provides overall higher energy savings (30% on average), compared to previously proposed semaphores. Second, although each application will in general have its own optimal timeout value, it is possible to identify an average optimal length of the timeout value, whose value is empirically shown to be dependent on the time required to switch from the sleep to the active state.

In Chapters 4 we presented *Embedded-TM*, a hardware transactional memory architecture specifically tailored for MPSoCs. We showed that hardware transactional memory is a viable way to structure synchronization on power-constrained embedded systems, as long as it is designed with energy efficiency in mind. A straightforward
implementation of Embedded-TM, using a dedicated, fully-associative transactional cache, was power-hungry, even when selectively powered down. Instead, we found that we could devise a better design by mixing and matching known techniques, such as placing transactional entries in the L1 cache backed up by a small victim cache (powered down when not in use).

In Chapter 5 we described two further design optimizations for Embedded-TM. First we proposed Transactional-log, a scheme that requires saving accesses to private memory in a core-local log. The goal of such optimization was to reduce the number transactional entries to keep in the cache. We showed throughput improvements because of the reduced number of overflows. The second optimization simplified conflict detection by decoupling the TM system from cache coherency. We implemented the Bloom module, a centralized signature-based device that saves the cores’ transactional accesses into local signatures. Conflicts were detected by simple comparisons of signature bits. We designed the Bloom module to be programmable by the software with simple memory operations on its registers.

Finally, in Chapter 6, we presented SoC-TM, a complete HW/SW framework for Transactional Memory programming on MPSoCs. We showed how SoC-TM eases application development on shared memory MPSoCs leveraging a lightweight yet efficient HTM design, coupled with the simplicity of directive-based programming of OpenMP.

Speculative parallelization is also supported, which makes the task of extracting parallelism from sequential codes less cumbersome. Our results confirmed the effectiveness of SoC-TM, concerning ease of use, performance, simplicity of design and energy consumption.
The work presented in this thesis can be extended in several directions. For the hardware, a distributed version of the Bloom module could improve the scalability of the design, and potentially reduce the energy and performance penalty associated to the broadcasting bus operations. For the software, compiler analysis can be leveraged in the auto-parallelization pass in GCC to determine candidate loops for speculative parallelization by estimating the independent work within each transaction. Moreover, our programming model can be implemented using the latest (3.0) OpenMP specification. Here the task construct enables very fine grained task parallelism, useful to build very flexible support for TLS.

It would be interesting to explore how alternative thread-scheduling mechanisms (like work-staling approaches [10]) can be combined in our framework, and to study the energy and the performance impacts on MPSoCs.
Bibliography


[22] EEMBC. Eembc, the embedded microprocessor benchmark consortium.


[26] Freescale-QE. Freescale low-power QE family processor.


