Automatic Skeleton-Driven Performance Optimizations for Transactional Memory

Luís Fabrício Wanderley Góes

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2012

Abstract

The recent shift toward multi-core chips has pushed the burden of extracting performance to the programmer. In fact, programmers now have to be able to uncover more coarse-grain parallelism with every new generation of processors, or the performance of their applications will remain roughly the same or even degrade. Unfortunately, parallel programming is still hard and error prone. This has driven the development of many new parallel programming models that aim to make this process efficient.

This thesis first combines the skeleton-based and transactional memory programming models in a new framework, called OpenSkel, in order to improve performance and programmability of parallel applications. This framework provides a single skeleton that allows the implementation of transactional worklist applications. Skeleton or pattern-based programming allows parallel programs to be expressed as specialized instances of generic communication and computation patterns. This leaves the programmer with only the implementation of the particular operations required to solve the problem at hand. Thus, this programming approach simplifies parallel programming by eliminating some of the major challenges of parallel programming, namely thread communication, scheduling and orchestration. However, the application programmer has still to correctly synchronize threads on data races. This commonly requires the use of locks to guarantee atomic access to shared data. In particular, lock programming is vulnerable to deadlocks and also limits coarse grain parallelism by blocking threads that could be potentially executed in parallel.

Transactional Memory (TM) thus emerges as an attractive alternative model to simplify parallel programming by removing this burden of handling data races explicitly. This model allows programmers to write parallel code as transactions, which are then guaranteed by the runtime system to execute atomically and in isolation regardless of eventual data races. TM programming thus frees the application from deadlocks and enables the exploitation of coarse grain parallelism when transactions do not conflict very often. Nevertheless, thread management and orchestration are left for the application programmer. Fortunately, this can be naturally handled by a skeleton framework. This fact makes the combination of skeleton-based and transactional programming a natural step to improve programmability since these models complement each other. In fact, this combination releases the application programmer from dealing with thread management and data races, and also inherits the performance improvements of both models. In addition to it, a skeleton framework is also amenable to skeleton-driven

performance optimizations that exploits the application pattern and system information.

This thesis thus also presents a set of pattern-oriented optimizations that are automatically selected and applied in a significant subset of transactional memory applications that shares a common pattern called *worklist*. These optimizations exploit the knowledge about the worklist pattern and the TM nature of the applications to avoid transaction conflicts, to prefetch data, to reduce contention etc. Using a novel autotuning mechanism, OpenSkel dynamically selects the most suitable set of these pattern-oriented performance optimizations for each application and adjusts them accordingly. Experimental results on a subset of five applications from the STAMP benchmark suite show that the proposed autotuning mechanism can achieve performance improvements within 2%, on average, of a static oracle for a 16-core UMA (Uniform Memory Access) platform and surpasses it by 7% on average for a 32-core NUMA (Non-Uniform Memory Access) platform.

Finally, this thesis also investigates skeleton-driven system-oriented performance optimizations such as thread mapping and memory page allocation. In order to do it, the OpenSkel system and also the autotuning mechanism are extended to accommodate these optimizations. The conducted experimental results on a subset of five applications from the STAMP benchmark show that the OpenSkel framework with the extended autotuning mechanism driving both pattern and system-oriented optimizations can achieve performance improvements of up to 88%, with an average of 46%, over a baseline version for a 16-core UMA platform and up to 162%, with an average of 91%, for a 32-core NUMA platform.

Acknowledgements

First, I would like to thank my family, friends and everyone that helped me somehow throughout my life. In particular, I am very grateful to my parents Dona Bete and Seu Luis that taught me the moral values in which my life was built upon.

It is impossible to express all my gratitude to my wife Raquel, the love and reason of my life. She is my link to the real world, my sanity checker. Her love, care, support and particularly her patience were key to keep me alive through this daunting journey. In return, the least I can do is to say that I will never let a job come before our lives again.

I would like to thank my supervisors Marcelo Cintra and Murray Cole for their support and encouragement. Their guidance helped enourmously to keep me concentrated and focused during this research even in moments of adversity. I would like also to thank the commitee members Vijayanand Nagarajan and Jeremy Singer for their helpful reviews that contributed to enhance this thesis.

I was wondering if I would be able to go that far without my compatriotes from New Texas: Vasileios, Karthik, Nikolas, Georgious, Konstantina and Andrew. This nation of friends helped me to come up with *Hamazing* ideas, and to cheer me up whenever *I'm tired!! Ahhh!!* Maybe *this is not even a sentence*, but the fact is that I will remember forever *all the things* we have done together.

Last, I would like to acknowledge the financial, academic and technical support of the University of Edinburgh and the Scottish Informatics and Computer Science Alliance, particularly in the award of a Postgraduate Research Studentship which allowed me to undertake this research.

Declaration

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

- "A Machine Learning-Based Approach for Thread Mapping on Transactional Memory Applications", High Performance Computing Conference (HiPC), 2011.

(*Luís Fabrício Wanderley Góes*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Leading processor manufacturers have recently shifted toward the multi-core design paradigm [39, 43]. As devices continue to scale we can expect future systems to be comprised of an even larger number of cores. Unfortunately, this means that to sustain performance improvements the programmers/compilers now have to exploit the available cores as much as possible through coarse-grain parallelism. Although parallel programming is not a new concept, the vast majority of programmers still find it a hard and error-prone process, especially when based on low-level programming approaches, such as threads with locks [54]. Among the many difficulties associated with such low-level parallel programming, data partitioning, synchronization, communication, and thread scheduling are the most challenging to deal with. Unfortunately, tools to support parallel programming are still in their infancy, while automatic parallelizing compilers still fail to deliver significant results in all but very few instances.

One alternative to simplify the development of parallel applications is to employ parallel algorithmic skeletons or patterns [5, 19, 61]. Skeleton-based programming stems from the observation that many parallel algorithms fit into generic communication and computation patterns, such as pipeline, map and reduce. The communication and computation pattern can be encapsulated in a common infrastructure, leaving the programmer with only the implementation of the particular operations required to solve the problem at hand. Thus, this programming approach eliminates some of the major challenges of parallel programming, namely thread communication, scheduling and orchestration. At present, skeletal principles are exploited in the Berkeley's motifs [5], Intel's Threading Building Blocks (TBB) [69] and Google's MapReduce [25]. These

are an important step toward consolidating parallel skeletons as a practical programming model for regular data and task-parallel applications.

Transactional Memory (TM) [37, 40] is another alternative parallel programming model. From a different perspective, it simplifies parallel programming by removing the burden of correctly synchronizing threads on data races [70]. This model allows programmers to write parallel code as transactions, which are then guaranteed by the runtime system to execute atomically and in isolation regardless of eventual data races. Hardware implementations of TM have been proposed, but software transactional memory (STM) [26, 29, 31, 57] is especially appealing since it can be deployed in existing shared-memory systems, including multi-cores. Although removing the burden of correctly synchronizing parallel applications is an important simplification, the programmer is still left with the tasks of thread scheduling and orchestration. These tasks can be naturally handled by a skeleton framework.

Another opportunity provided by skeletons in addition to the simplification of programming is the enabling of performance optimizations. The skeleton framework can exploit pattern, application and/or system information to perform optimizations such as communication contention management and data prefetching. Moreover, such skeleton-driven optimizations can be performed automatically, that is, without requiring any additional programming effort from the application programmer. Nevertheless, the decision of which optimizations should be enabled and how to adjust them for a given application is still a daunting task.

1.2 Main Contributions

1.2.1 Combining Parallel Skeletons and Transactional Memory

In this thesis, it has been identified that many current transactional memory applications share a common parallel pattern. This pattern is called *Worklist*, in which worker threads grab work-units from a worklist. In order to process these work-units, worker threads access and update shared data structures as depicted in Figure 1.1. These operations are performed under the supervision of a transactional memory system to handle memory conflicts transparently. That pattern can be made explicit to the programmer by employing a skeleton approach while *hiding* transactional memory semantics.

The first contribution of this thesis is to combine the worklist parallel skeleton with software transactional memory into a single framework, inheriting the programming

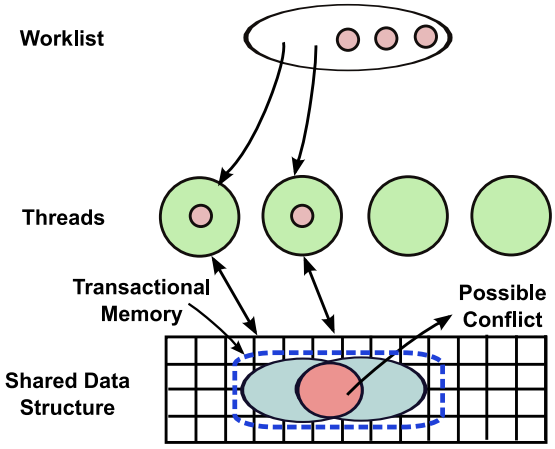

Figure 1.1: An example of a transactional worklist application.

and performance benefits of both models. This framework is called OpenSkel and it provides the new *Transactional Worklist* pattern. In previous work, to the best of our knowledge, no one has exploited the use of skeletons on top of existing STM applications and systems as a means of improving performance and programmability.

OpenSkel simplifies parallel programming based on the fact that it makes thread and worklist management, and the interaction with the STM system fully transparent to the application programmer. In fact, OpenSkel provides a minimal API that permits the programmer to focus mainly in the implementation of the application kernel. This releases the programmer from dealing with many parallel programming issues such as thread synchronization, data races, load balancing etc. Additionally, experimental results on a subset of five applications from the STAMP benchmark suite also show that the baseline version can achieve performance improvements of up to 28%, with an average of 3%, over the original TM version for a 16-core UMA (Uniform Memory Access) platform and up to 26%, with an average of 3%, for a 32-core NUMA (Non-Uniform Memory Access) platform. These results show that on average the OpenSkel framework improves programmability without degrading the application performance.

1.2.2 Pattern-Oriented Performance Optimizations

As mentioned earlier, parallel skeletons create an opportunity to enable transparent skeleton-driven performance optimizations. Some of these optimizations make use of the well structured pattern defined by a skeleton. In particular, both the worklist and TM nature of the applications can be exploited in the *transactional worklist* pattern.

Another contribution of this thesis is to adapt and enable pattern-oriented perfor-

mance optimizations and analyze their combined performance impact on real applications. In this thesis, several pattern-oriented optimizations such as *work stealing*, *work coalescing*, *swap retry* and *helper threads* are analyzed in detail. Among these optimizations, *helper threads* are employed for the first time (to the best of our knowledge) to transactional applications using a completely novel approach. It exploits the fact that transactional applications have scalability limitations and spare cores can be used for data prefetching. Additionally, it uses the skeleton structure to solve the issue of synchronization between the worker and helper threads.

Experimental results on a subset of five applications from the STAMP benchmark suite show that employing combined pattern-oriented performance optimizations the OpenSkel framework can achieve performance improvements of up to 90%, with an average of 48%, over a baseline version for a 16-core UMA platform and up to 111%, with an average of 49%, for a 32-core NUMA platform.

1.2.3 Dynamic Autotuning Mechanism

Although enabling many pattern-oriented optimizations transparently in a single framework is an important achievement, the application programmer is still left with the daunting task of choosing the most profitable set of optimizations. He has also to manually determine internal parameters for each optimization. Additionally, the static selection of optimization parameters inhibits the framework from exploiting dynamic information to adjust them and improve performance even further.

Another contribution of this thesis is a novel autotuning mechanism that dynamically selects a suitable performing set of pattern-oriented optimizations for each application. This mechanism incrementally enables and tunes optimizations following their performance impact and abstraction level order. It also automatically tunes the thread concurrency level using a hill-climbing strategy.

Experimental results on a subset of five applications from the STAMP benchmark suite show that the proposed autotuning mechanism can achieve performance improvements within 2%, on average, of a static oracle for a 16-core UMA platform and surpasses it by 7% on average for a 32-core NUMA platform.

1.2.4 System-Oriented Performance Optimizations

Another branch of skeleton-driven performance optimizations are the system-oriented ones used to enhance system functionalities such as thread scheduling/mapping and

memory page allocation. The operating system usually takes care of these services applying some generic heuristics. In some cases, these heuristics deliver good performance to applications for the average case, but they perform poorly for several specific instances depending on the application behavior and platform. A skeleton-based framework can provide a set of alternatives to the default operating system strategies in a transparent way. In addition to it, these alternative strategies can be automatically selected by the skeleton framework.

The last contribution of this thesis is to enable system-oriented performance optimizations for thread mapping and memory page allocation within OpenSkel and investigate their performance impact on transactional worklist applications. In particular, the proposed autotuning mechanism is extended to also dynamically select system-oriented optimizations combined with the pattern-oriented ones.

Experimental results on a subset of five applications from the STAMP benchmark suite show that combining system-oriented and pattern-oriented optimizations guided by an extended version of the proposed autotuning mechanism, the OpenSkel framework can achieve performance improvements of up to 88%, with an average of 46%, over a baseline version for a 16-core UMA platform and up to 162%, with an average of 91%, for a 32-core NUMA platform.

1.3 Thesis Structure

The rest of this thesis is organized as follows.

Chapter 2 provides background information on skeleton-based and transactional programming. It also briefly summarizes some of the state-of-the-art techniques designed for automatic parallel software optimizations focusing on transparent optimizations and autotuners.

Chapter 3 describes the proposed transactional skeleton-based framework. It starts by presenting the worklist pattern and the optimization opportunities unveiled when combined with transactional memory. Then it presents the OpenSkel framework, its application programming interface (API), its implementation details and the baseline version.

Chapter 4 outlines our experimental methodology. It provides details on the hardware platforms, transactional memory system and benchmark applications. In subsequent chapters, this information will be needed in order to analyze the experimental results. This chapter ends with a performance comparison between the original ver-

sion of the STAMP benchmark applications and their corresponding versions ported to OpenSkel.

Chapter 5 presents the concepts and implementation details of four pattern-oriented performance optimizations. In this chapter, each optimization is evaluated individually and compared to the baseline version. This chapter ends with a discussion of the overall performance improvement across the evaluated optimizations.

Chapter 6 introduces the proposed autotuning mechanism. It describes each step of the autotuning mechanism and the hill-climbing algorithms to tune *helper threads* and thread concurrency level. Then the performance improvement of the autotuning mechanism is compared to a static oracle. Additionally, a sensitivity analysis of the autotuning mechanism is conducted in order to better understand its internal parameters behavior. Finally, this chapter shows the dynamic behavior of the mechanism for each application.

Chapter 7 presents skeleton-driven system-oriented optimizations for thread mapping and memory page allocation. This chapter discusses the performance benefits of applying these optimizations compared to the baseline version. It also proposes an extension to the autotuning mechanism in order to automatically select system-oriented optimizations. Then it concludes with an analysis of the performance improvement of system-oriented optimizations combined with pattern-oriented ones.

Chapter 8 describes and compares the existing related work with this thesis contributions. It covers parallel programming frameworks, including parallel skeletons, pattern-oriented and system-oriented optimizations, and autotuning mechanisms.

Chapter 9 summarizes the contributions of this thesis and highlights some possible future work.

Chapter 2

Background

This chapter first presents the background on parallel programming models. Second, it outlines concepts of the skeleton-based parallel programming model and discusses some existing skeleton frameworks. Then it introduces the transactional memory programming model and presents some available software libraries. Finally, this chapter describes automatic parallel software optimizations including autotuners.

2.1 Parallel Programming Models

Parallel programming can be split in two main programming models: message passing and shared variable [24, 34, 42]. Both models provide low-level primitives to build parallel applications. The former model is more suitable for distributed memory machines such as clusters and grids. In this model, each node has its own private memory, as depicted in Figure 2.1a. For instance, if a thread placed on core 1 (C1) in node 0 needs to read data from node 1, it must send an explicit data request message to another thread that holds the required data on node 1. When the message is delivered, the thread on node 1 then reads the data from its local memory and sends it back to requesting thread through the network. Note that this programming model forces an application programmer to handle parallelism explicitly. As a result, the programmer has to be aware of thread communication and synchronization, data placement and network topology. This parallel programming model is supported by message passing libraries such as the Message Passing Interface (MPI) [34].

In contrast, the shared variable model is widely used in shared memory machines such as SMPs (Symmetric Multiprocessors), NUMA (Non-Uniform Memory Access) machines and multi-cores. Figure 2.1b shows that in this model cores share a common

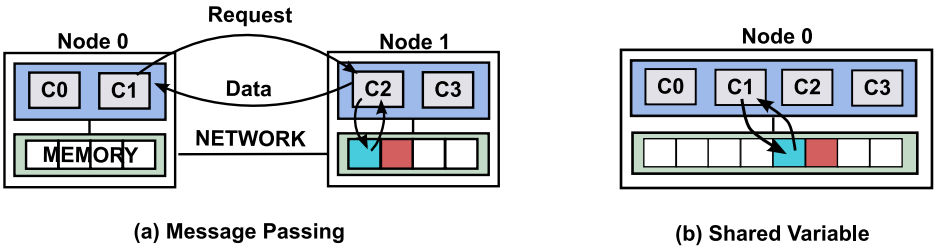

Figure 2.1: Parallel programming models: (a) In the message passing model, each node has its own private memory and threads placed on cores (C0, C1, C2 or C3) from different nodes need to exchange messages in order to read a remote data; (b) In the shared variable model, threads can access data directly in the shared memory.

main memory wherein threads communicate by synchronized read and write operations on shared variables. Nowadays, locks are the most usual technique to implement the shared variable model. In lock-based programming, for each shared data access or critical section access, a thread must acquire a lock before accessing the data [45]. If another thread has already acquired the same lock, the latter thread must wait until the lock is released by the former thread [42]. This approach has some drawbacks, for instance: i) coarse-grain locks or large critical sections limit parallelism; ii) misplaced locks results into deadlock; iii) error recovery is not trivial; and iv) lock programming is not modular.

A large number of programming libraries such as pthreads and Java threads support the shared variable model based on lock-based programming [42]. These libraries force the programmer to explicitly handle thread control and locks. Nevertheless, there are other approaches to provide shared variable programming known as implicit parallelism techniques, such as language extensions to support parallelism (e.g., HPF - High Performance Fortran) and compiler directives (e.g., OpenMP) [24, 42, 77]. Implicit parallelism avoids that programmers have to create and synchronize threads. Instead, they only have to annotate pieces of code that can be executed independently in parallel. A compiler thus automatically handles the parallelism issues. In short, implicit parallelism works only on regular loops in which iterations are free of data dependencies. Additionally, implicit parallelism techniques do not completely free programmers from locks. Finally, if a programmer wants to exploit task parallelism within an application, he must protect global shared counters and conditional variable accesses with locks [77].

2.2 Skeletal Parallel Programming

Skeletal parallel programming is a high-level programming model that is orthogonal to message passing and shared variable programming. It is a pattern-based approach which proposes that parallel programming complexity should be addressed by extending the programming model with a small number of architecture independent constructs, known as *algorithmic skeletons* [19, 32]. Each skeleton specification captures the behavior of a commonly occurring pattern of computation and interaction, while packaging and hiding the details of its concrete implementation [60]. This both simplifies programming, by encouraging application and combination of the appropriate skeletons, and enables optimizations, by virtue of the *macro* knowledge of application structure that is provided.

Essentially, the skeleton "knows what will happen next" and can use this knowledge to choose and adjust implementation details. For instance, skeleton implementations may be able to place threads that communicate frequently on cores that share some level of cache memory, to prefetch data for the next step of a thread computation, and so on. A key benefit of skeletons is that the optimizations can be applied *transparently* and *architecture-sensitively*, without user intervention, to any application for which the programmer has used the corresponding skeleton.

2.2.1 Parallel Algorithmic Skeletons

Algorithmic skeletons have been classified in many categories according to their behavior [14, 32, 55]. A recent and general classification [55] divides skeletons into three categories: i) control skeletons - they act as a link to compose other skeletons; ii) data-parallel skeletons - these skeletons exploit data-parallelism, that is, threads execute the same task across multiple input data; and iii) task-parallel skeletons - these skeletons exploit task-parallelism, that is, threads execute a different task across a single or multiple input data. For instance, the most common control skeletons are: i) *seq* (sequential) - it acts as a barrier to ensure the termination of recursive nesting of other skeletons; ii) *while* - a skeleton is executed several times until a condition is not satisfied; iii) *for* - a skeleton is executed a pre-defined number of times; and iv) *if* - it is a conditional branch that receives an output from a skeleton, and based on a condition, decides which skeleton will be executed next [55, 81].

In some skeleton programming libraries such as Intel TBB, instead of a control skeleton, the *for* skeleton is a data-parallel skeleton in which iterations of a *for* loop

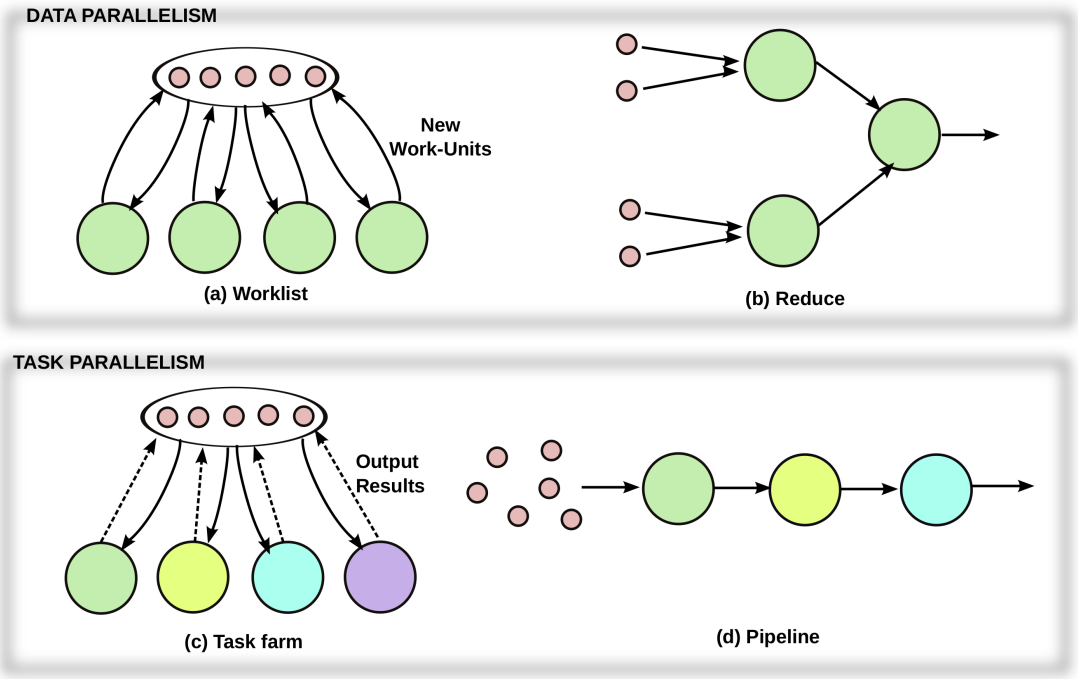

Figure 2.2: Examples of parallel algorithmic skeletons. Small red circles represent input data elements. Big circles correspond to threads and different colors represent different tasks. Arrows stand for the communication flow between threads.

are executed in parallel [69]. Other well-known data-parallel skeletons are: i) *map* - it represents the classical single instruction multiple data parallelism (SIMD). An input data is split into smaller data elements, and the skeleton maps each element to a different thread; ii) *reduce* (or scan) - it is the opposite operation of *map*. It takes different input data elements and applies an operator recursively on them until it has a single output (see Figure 2.2b); iii) *mapreduce* - it is a more complex skeleton that basically combines *map* and *reduce* skeletons; iv) *divide and conquer* - it is a general case of *mapreduce*. It recursively divides the input data until it reaches a pre-defined condition. Then it executes some computation over the data. After this stage, it starts hierarchically merging pairs of partial results until it has a single result; and v) *worklist* - it iterates over a worklist until the list is empty as depicted in Figure 2.2a. The work-units or data elements are executed in parallel by worker threads. Additionally, a worker thread can produce and add new work-units to the worklist during its execution [25, 52, 55].

In task-parallelism skeletons, each thread computes a task over a single or multiple data. For example, a *task farm* skeleton has one master and a number of worker threads. The master thread schedules tasks to workers and collects their results. As opposed to

the *worklist* pattern, worker threads process *different* tasks on a single or multiple data as depicted in Figure 2.2c. Another classical task-parallel skeleton is the *pipeline*. As shown in Figure 2.2d, it is composed of a number of sequential stages executing different tasks in parallel [32, 55, 81]. Interestingly, each stage of a *pipeline* skeleton can be composed by another data-parallel skeleton (e.g., *map* or *worklist*) to increase parallelism.

In the literature, there are many other skeletons, for instance: *parmod, multi-block, deal, butterfly, hallowswap, filter, intermem, loop, ring, torus, comp, fold, fork, expander, geometric, queue, heartbeat, wavefront, singleton, sort, replicate* and so far [1, 32, 55]. Unfortunately, there is not a standard for designing parallel skeletons, therefore some of these skeletons have different names to represent the same behavior. Furthermore, some skeletons, as mentioned before, exist just for control purposes to enable skeleton composition, such as *while* and *seq* [1].

2.2.2 Skeleton Frameworks

Parallel skeletons are commonly implemented as language extensions or libraries. As described in [32, 55], there is an extensive list of these skeleton-based frameworks. This section summarizes three typical frameworks in chronological order: eSkel, Lithium (Muskel) and Calcium. These frameworks provide a comprehensive set of skeletons and also highlight the evolution of skeleton-based programming.

The Edinburgh Skeleton library (eSkel) was the first implementation of parallel skeletons [19]. It is a C skeleton library that extends MPI [20]. Basically, the eSkel framework defines five skeletons: *pipeline*, *task farm*, *deal*, *halloswap* and *butterfly*. These skeletons thus can be composed and nested to describe complex parallel communication patterns. Recently, it has been extended to support automatic scheduling of algorithmic skeletons [32].

Lithium is a Java library that provides a set of composable classic parallel skeletons, such as *map*, *pipeline*, *divide-and-conquer*, *comp*, *farm* and *while* [1]. These template-based skeletons are stateless, therefore Lithium does not support any kind of global state. Its successor, called Muskel, adressed this limitation by implementing stateful skeletons such as farm and pipeline [2]. Both are designed for distributed memory machines such as clusters and grids (i.e., shared memory is not supported).

Calcium is a Java skeleton framework, inspired by Lithium and Muskel, that uses the ProActive middleware for distributed computing [55]. It supports nestable, task and data-parallel skeletons. Although it focuses on grid computing, it also provides multithreading for shared memory machines. In comparison to Muskel, Calcium has stateless skeletons, type safe operations, transparent file access and an additional skeleton called *fork*.

Skeletons can also be found in the Berkeley ParLab software strategy [5], where they are known as "program frameworks", are present in Intel's Threading Building Blocks (TBB) software [69] in the form of the pipeline and scan operations, and are exemplified by Google's MapReduce paradigm [25] and frameworks derived from it such as Phoenix [68], Hadoop [79] and MRJ [71].

2.3 Programming with Transactional Memory

For at least the short and medium term, multi-core processors will present shared address space programming interfaces. If a conventional approach is followed, these will present complex, weakened memory models, synchronization built around locks and condition variables. In contrast, the Transactional Memory (TM) model [31, 37] offers both conceptually simplified programming and potential for competitive, or even improved performance against traditional approaches. In essence, TM requires the programmer to express all synchronization through transactions: blocks of code that must *appear* to execute atomically. The core aim of a TM implementation is to allow as many transactions as possible to proceed concurrently, in potential conflict with the atomic semantics, backtracking (or "aborting") on one or more transactions only when the memory accesses *actually* conflict. It is important to note that TM is an interesting model to program applications in which the data access pattern is not known a priori. For instance, in a stencil application in which the communication and synchronization pattern is known at compile time, the use of transactional memory would lead to unnecessary runtime overhead to deal with data races that could be avoided by applying simple data partitioning mechanisms.

TM systems can be either implemented in software [16, 22, 45, 57, 58, 75] or hardware [9, 10, 18, 36, 64, 67]. In this thesis, the main focus is on Software Transactional Memory (STM). Overall, STMs provide a small set of basic operations implemented

```
01: void foo(void *args)
02: {
03:     ...
04:     atomic {
05:         a = b + c;
06:     }
07:     ...
08: }
```

(a) Source code

```
01: void foo(void *args)
02: {
03:     ...
04:     tm_start();
05:         tm_write(a,tm_read(b) + tm_read(c));
06:     tm_commit();
07:     ...
08: }
```

(b) Translated code

Figure 2.3: A high-level user code using the atomic statement before and after a TM compiler translation.

as an API. The first operation is *tm_start()* that is responsible to create a transaction. Once a transaction is started, each access to shared data must be protected by transactional barriers. These barriers are commonly implemented within *tm_read()* and *tm_write()* operations. The STM system thus can keep track of each data update and identify possible data conflicts. Another important operation is *tm_commit()* that terminates the transaction, updating shared data to memory with new values produced by *tm_write()* operations. These operations can be automatically placed by a TM compiler as shown in Figure 2.3. However, it requires the use of the *atomic* statement to delimitate the scope of a transaction. Finally, the *tm_abort()* operation may be used during any other TM operation if a conflict is detected and a transaction must be squashed and restarted [57, 59, 73].

In STM systems, transactions modify data in a shared memory without being concerned about other concurrent transactions. In order to provide this abstraction, each memory address accessed by a *tm_read()* or *tm_write()* operation is recorded in a log, instead of updating the memory address directly. On a transaction commit, the STM runtime system verifies if another transaction has modified any common memory addresses. If none of them has modified any shared data, the transaction validates and commits the data. Otherwise a transaction is aborted and re-executed from its beginning. Although STMs avoid deadlocks and livelocks (most of the time), some I/O operations can not be reversed. In this case, some systems provide an inevitable mode to guarantee that I/O transactions never abort [59, 73, 74].

2.3.1 STM Synchronization Mechanisms

STM implementations exploit nonblocking synchronization mechanisms in order to provide concurrent execution of transactions. These mechanisms can be split in three classes: i) wait-freedom - it guarantees that all transactions sharing the same set of concurrent data make progress in a finite number of time steps. It avoids the occurrence of deadlocks and starvation; ii) lock-freedom - it guarantees that at least one transaction sharing the same concurrent data makes progress in a finite number of time steps. It avoids the occurrence of deadlocks but not starvation; and iii) obstruction-freedom - given a group of transactions accessing the same set of concurrent data, it guarantees that at least one transaction will make progress in the absence of contention. It avoids the occurrence of deadlocks, however livelocks can occur if all transactions start aborting each other.

In practice, despite wait-freedom being a desirable property, the synchronization overhead is too high to be implemented in a real system. Lock-free based STMs present reasonable performance. However, when compared to obstruction-free STMs with good contention resolution policies, lock-free based STMs design is more complex, less flexible and efficient. As a result, most recent STMs are built based on the obstruction-freedom property [57]. Nevertheless, conflict resolution, formally called contention management, thus becomes an important part of the STM design.

In order to be resolved, a conflict has first to be detected. This is handled by a conflict detection mechanism. Typically, a conflict can be detected as soon as it happens, called eager detection, or left to commit time, that is, lazy detection. Eager detection reduces the overhead in the *tm_commit()* operation, while increasing the overhead during the transaction execution. In contrast, lazy conflict detection can potentially result in excessive wasted work, specially if there are long transactions under high contention to shared data [57, 74].

2.3.2 STM Libraries

Existing STMs can be split in two categories: object-based [58] and word-based [74]. Word-based STMs detect conflicts and guarantee consistency for each access to shared words in the memory. This may result in unacceptable overheads due to its fine granularity level [58]. On the other hand, in object-based STMs, each object behaves as a buffer that supports multiple accesses by different transactions. In this case, these coarse-grained objects may cause high contention, leading applications to perform

poorly in some instances [57]. This section describes the most widely known object and word-based STM systems: TL2, TinySTM and RSTM.

Transactional Locking II (TL2) is implemented as a C library. The main idea behind TL2 is the use of a global version-clock that is incremented on each *tm_write()* operation, and then read by all transactions [26]. This approach avoids the reading of inconsistent memory states and it allows that memory locations allocated within a transaction be reallocated in a non-transactional code. In its original version, it supported only lazy conflict detection. A newer version available in the STAMP benchmark also provides eager conflict detection [15].

TinySTM is a word-based STM library that uses locks, instead of CAS (compare-and-swap) operations, to access shared memory locations [29]. Similar to TL2, it is also a word and time-based STM that guarantees consistent memory states to transactions. Furthermore, it provides two memory access policies: write-through (i.e., directly updates memory) and write-back (i.e., updates only at commit time). Additionally, TinySTM has some contention management strategies such as suicide (i.e., aborts a transaction as soon as it detects a conflict) and delay (i.e., same as suicide, but it waits until the contended lock has been released).

Rochester Software Transactional Memory (RSTM) is a C++ library for object-oriented transactional programming [58]. It supports a wide range of contention management mechanisms (e.g., aggressive, eruption, greedy, highlander, justice etc.). RSTM also reduces cache misses by employing one single level of indirection to access shared objects. It means that each object has a unique metadata structure during its lifetime, avoiding the creation of a new locator whenever a object is acquired by a transaction. In addition to it, RSTM has an epoch-based collector for transactional data objects. Recently, RSTM also provides inevitability and privatization policies. Inevitability allows transactions to perform I/O operations within a transaction [59, 73] while privatization policies deal with the interaction between transactional and non-transactional code over the same shared data. Furthermore, RSTM has a collection of applications and supports a wide range of architectures and operating systems.

Despite the effort to design a high performance STM system, [16] points out some weaknesses of existing STM systems. For instance, I/O operations within a transaction

must be handled by the application programmer through inevitability calls. Otherwise, during a transaction abort operation, the rollback of I/O operations can be very complex and time consuming to be implemented in a STM. Furthermore, STMs are still vunerable to livelock and privatization. Although [16] claims that many applications can not scale due to high contention in the current STM systems, [28] shows that these STM systems still achieve high performance compared to sequential applications.

2.4 Automatic Parallel Software Optimizations

In the multi-core era, automatic software parallelization and optimization has become crucial to achieve high performance [5]. Various types of performance optimizations have been employed to parallelize and speed up applications [25, 62, 63, 69]. They are commonly *transparent*, that is, *hidden* from the application programmer by compiler and/or runtime system support. This thesis focuses on runtime optimizations.

2.4.1 Runtime Performance Optimizations

Runtime optimizations are mainly performed transparently by the operating system and middlewares such as the Java Virtual Machine (JVM). These optimizations use runtime information (e.g., memory behavior) to migrate threads and data, to optimize data allocation and garbage collection, to adjust the thread concurrency level etc. The program structure can also be used to apply runtime optimizations. For instance, a *for* loop has a fixed number of iterations. In OpenMP, these iterations can be scheduled dynamically in order to balance the amount of work between threads. In contrast to compiler optimizations, global optimizations can be applied at runtime. However, compiler support is still needed to auto-parallelization.

A more interesting approach is to combine both compiler and runtime support to provide automatic optimizations. MapReduce is a classical example in which the application programmer writes sequential code independent from a target platform. Then, a compiler translates that code into parallel code and a runtime system deals with thread and communication management. Additionally, the runtime system performs transparent and automatic optimizations such as task scheduling, load balancing and replication [25]. These optimizations and some others, such as work stealing, are also exploited in systems such Intel TBB and OpenMP.

2.4.2 Autotuners

An interesting problem arises when the same code has to be compiled and executed on several different platforms. The number of possible combinations between optimizations and platforms becomes exponential. In order to find the most efficient code plus runtime optimizations, it requires exhaustive search across the space of execution instances. In addition to it, many optimizations themselves need parameter tuning, increasing even further the search space. To tackle this problem, autotuners are employed to generate variants of the given code, test, adjust and select the most efficient instance [5].

Autotuning techniques can be applied at compile time and/or runtime. Usually, off-line approaches involve Machine Learning (ML) techniques to reduce the search space. Machine learning has become a common component of approaches to model the behavior of complex interactions between applications, optimizations and platforms [33, 71, 76, 78]. It provides a portable solution to predict the behavior of new combinations of application/optimizations/platforms, also called instances, based on apriori profiled runs. ML-based approaches share a common framework that is commonly composed of a static and a dynamic phase. The static phase is subdivided in the following three major steps: i) application profiling; ii) data pre-processing and feature selection; and iii) learning process. Its target is to build up a predictor to be integrated in the actual system. In the dynamic phase, each new instance is profiled at runtime and then the collected data is used as an input to the predictor. Finally, the predictor outputs the target variable and the system applies it.

In a pure runtime approach, exhaustive search is not feasible due to timing constraints. Thus it normally relies on simple heuristics based on application runtime profiling [29, 82]. It takes into account application features, platform characteristics and performance metrics to tune performance optimizations dynamically. For instance, hill-climbing is a technique that converges to an efficient solution based on feedback information to evaluate the actual performance improvement on each progression step [29]. The side-effect of dynamic approaches is that the on-line profiling can lead to high runtime overhead.

Chapter 3

Combining Parallel Skeletons and Transactional Memory

Lock-based programming is often very efficient in terms of performance for fine-grained parallelism. However, when programability is considered as an important goal, locks fail to provide an intuitive abstraction for the programmer. For coarse-grained parallelism, lock-based programming does not deliver both programmability and performance. This is, thus, the case where the transactional memory programming model shines. It provides a programming abstraction equivalent to a lock-based one, without side-effects such as deadlock, and at the same time attaining high performance on coarse-grained parallel applications [37]. Regardless of the performance benefits of transactional memory, the application programmer still has to deal with the placement of transactional barriers and thread management. Skeleton-based programming can naturally address these issues since it provides structured parallelism constructs and provide a more intuitive programming abstraction.

3.1 How Skeletons Can Improve Transactional Memory Applications

In order to apply the skeletal programming methodology within the context of TM, there are three important questions to be answered: *What are the relevant skeletons?*, *Which optimization opportunities can be exploited by these skeletons?* and *How to automatically tune these skeletons?* The focus of this thesis is a TM oriented *worklist* skeleton, for which a number of performance optimizations have been investigated and

an autotuning mechanism to select and adjust them has been proposed. The skeleton was derived from a study of TM applications from the STAMP benchmark suite. After detailed code analysis, it was detected that the most predominant pattern on the STAMP benchmark suite was the worklist. Other applications from STAMP present an irregular behavior with specific characteristics which make them hard to generalize to a single skeleton, such as an arbitrary number of synchronization barriers and very fine grained transactions. Thus only the STAMP benchmark applications that fit the worklist pattern are used in this thesis. This is in keeping with standard pattern-oriented programming philosophy: no single skeleton will fit all applications, and it is likely that access to ad-hoc parallelism will always be required for the most inherently unstructured applications. Finally, the study of other skeletons is left as future work.

3.1.1 A Skeleton for Transactional Applications

Many TM applications exhibit the *worklist* pattern. Such applications are characterized by the existence of a single operation: process an item of work known as a *work-unit* from a dynamically managed collection of work-unit instances, the *worklist*.

```
   input: Work-Units u ∈ U, Worklist W,
          Threads t ∈ T
 1 begin
 2 |  Add seed work-units u_i into W
 3 |  foreach t_i ∈ T do
 4 |  |  while W ≠ Ø do
 5 |  |  |  Remove a work-unit u_j from W
 6 |  |  |  Process u_j
 7 |  |  |  [Add new work-units u'_j to W]
 8 |  |  end
 9 |  end
10 end
```

Figure 3.1: Generic behavior of the worklist skeleton found in many transactional memory applications.

The algorithm in Figure 3.1 sketches the generic behavior of worklist algorithms. The worklist is seeded with an initial collection of work-units. The worker threads then iterate, grabbing and executing work-units until the worklist is empty. As a side

effect of work-unit execution, a worker may add new work-units to the worklist.

Typically, work-units access and update common data and require mutual exclusion mechanisms to avoid conflicts and ensure correct behavior. Such applications, in areas such as routing, computer graphics, and networking [52, 73] are fertile territory for TM programming models. The proposed *transactional worklist skeleton* ensures correctness by executing all concurrent computation of work-units protected by transactional memory barriers. The implementation details of this new skeleton is discussed later in this chapter.

3.1.2 Performance Optimization Opportunities

The proposed transactional worklist skeleton also provides many performance optimization opportunities. These opportunities derive from pattern and STM information. First, the worklist pattern carries the important semantics that there is no required ordering on execution of available work-units. This frees the implementation to radically alter *the mechanisms by which the worklist is stored and accessed*.

Second, the worklist pattern does not specify lock-step progression by workers through iterations. This means that an execution in which some worker commits the effects of several work-units in sequence, without interleaving with other workers is valid. This gives the skeleton freedom to experiment with the *granularity of transactions*.

Another opportunity stems from the fact that the proposed skeleton deals with transactional applications. Preliminary investigations revealed that some of the worklists as implemented in the original benchmarks were causing aborts by tending to focus activity within small regions of the application data space. These were often unnecessary, since many other *potential work-units were available* which were better distributed across the data space.

Additionally, many of the selected applications seem to experience inherent scalability constraints, leading to poor returns for the use of additional cores. This observation creates an opportunity to exploit the *assignment of additional cores to do some other useful computation* that would help to boost the execution of the application as a whole. These pattern-oriented optimizations mentioned above are discussed in detail in Chapter 5.

Finally, skeleton frameworks can also exploit system level information to optimize the execution of applications. For instance, depending on the machine's memory hier-

archy a specific thread mapping strategy can be more efficient than the one provided by the operating system. The skeleton framework can provide these optimizations as an alternative to the operating system default strategies. These system-oriented optimizations are discussed in Chapter 7.

3.1.3 Autotuning Skeletons

As mentioned earlier, skeletons have the full control of threads and know the structure of the application behavior. In particular, for the worklist transactional skeleton, this allows the skeleton to communicate directly with the underlying TM system and adjust the worklist structure accordingly. More specifically, the skeleton collects runtime performance measurements such as work-unit aborts and commits, the number of stalls to access the worklist and work-units throughput. This information, coupled with the current parameters with which the measurements are collected, and then used to drive decisions on how to adjust the behavior so that performance improvements are attained. Adjusting the behavior is done by enabling or disabling performance optimizations or even by fine-tuning internal parameters for specific optimizations. Chapter 6 presents a novel dynamic autotuning mechanism that allows skeletons to automatically adjust to the application behavior.

3.2 The OpenSkel System

OpenSkel is a C runtime system library that enables the use of the *transactional worklist skeleton*. It provides an API to handle transactional worklists and implements transparent skeleton-driven performance optimizations with autotuning. OpenSkel relies on existing word-based STM systems to deal with transactions.

Figure 3.2 shows the steps to implement and execute transactional worklist applications using OpenSkel. Additionally, it shows the interaction between the OpenSkel framework, the STM system and the application. First, a programmer identifies if a given sequential code or a kernel can be potentially executed as a transactional worklist. Then, he adds OpenSkel API calls around that sequential code in order to satisfy the OpenSkel skeleton structure. Second, this new structured code is instrumented with TM calls by a TM compiler and translated into binary code. The TM compiler also generates code for one of the pattern-oriented optimizations as discussed in Chapter 5. Finally, the application interacts with the OpenSkel and STM runtime systems dur-

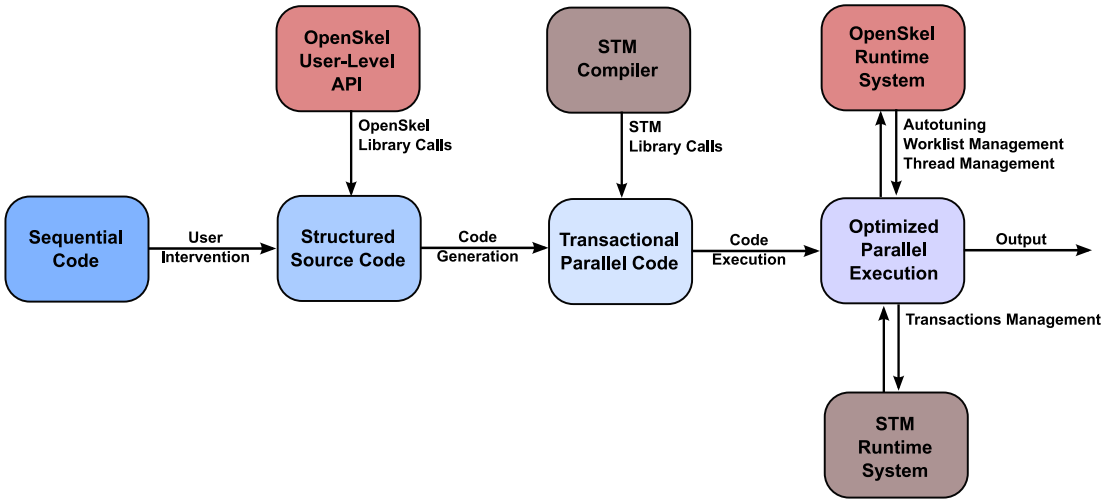

Figure 3.2: The OpenSkel framework interaction with the application code and transactional memory system.

ing its execution. The STM runtime system takes care of data conflicts. When they are detected, transactions are squashed and re-executed. In parallel, the OpenSkel runtime system manages threads and the worklist. Furthermore, it applies pattern-oriented optimizations automatically depending on the application behavior.

3.2.1 The OpenSkel API

In order to build a transactional worklist application, the OpenSkel framework provides an API presented in Table 3.1. The programmer is provided with three basic primitives so as to allocate, run and free a worklist, respectively *oskel_wl_alloc*(), *oskel_wl_run*() and *oskel_wl_free*() functions. Additionally, the API provides a function, namely *oskel_wl_addWorkUnit()*, with which the programmer can dynamically add work-units to the worklist. Following the abstract model, the programmer should feed the worklist with initial work-units using *oskel_wl_addWorkUnit()* before calling *oskel_wl_run()* or do it dynamically, when the skeleton is already running. It is important to note that these are the only function calls that a programmer has to learn to start programming with OpenSkel.

Figure 3.3 shows a typical application pseudocode based on the OpenSkel API. The programmer has to implement four functions required to describe a transactional worklist. The functions *oskel_wl_initWorker()* and *oskel_wl_destroyWorker()* respectively initialize and terminate local variables used by each worker thread. The main

OpenSkel Transactional Worklist API
oskel_t* **oskel_wl_alloc**(void* (***oskel_wl_initWorker**)(oskel_wl_shared_t* global), void (***oskel_wl_processWorkUnit**)(void* w, oskel_wl_private_t* local, oskel_wl_shared_t* global), void (***oskel_wl_update**)(oskel_wl_private_t* local, oskel_wl_shared_t* global), void (***oskel_wl_destroyWorker)**(oskel_wl_private_t* local, oskel_wl_shared_t* global))
Return an instance of a **transactional worklist skeleton** and fills each thread skeletal structure.
void **oskel_wl_addWorkUnit**(oskel_t* **oskelPtr**, void* **workUnit**)
Insert a **work-unit** into the **transactional worklist skeleton**.
void **oskel_wl_run**(oskel_t* **oskelPtr**, oskel_wl_shared_t* **global**)
Run the **transactional worklist skeleton** with **global** arguments until the worklist is empty.
void **oskel_wl_free**(oskel_t* **oskelPtr**)
Free the **transactional worklist skeleton**.

Table 3.1: The OpenSkel Application Programming Interface function calls.

function is *oskel_wl_processWorkUnit()*, in which the programmer implements the kernel to process an individual work-unit. Lastly, the *oskel_wl_update()* implements any kind of operation to update the global data when a worker thread is just about to finish.

Figure 3.4 presents a simple application code to update a bank account. In this application, clients keep depositing a certain amount to a bank account until there are no more deposits to be done. First, the worklist has to be filled up with the amounts to be deposited (i.e., work-units) as presented on line 14 in Figure 3.3. This is a generic step in which all worklist applications have to fill up the worklist with work-units. Then, each client has a simple task: to add a given amount to the current balance and to count the number of deposits it has done so far. This kernel is implemented on line 16 in Figure 3.4. Particularly, each client has to initialize its counter in the *oskel_wl_initWorker()* function before start processing deposits and reset it in the *oskel_wl_destroyWorker()* function when the client is about to finish.

In addition to the aforementioned functions, the programmer has also to declare two structures as part of the interface specification. The *oskel_wl_shared_t* structure contains all shared global variables, and the *oskel_wl_private_t* data structure specifies all private local variables of each thread. In this example, the number of deposits is local to each client, so it has to be declared in the *oskel_wl_private_t* data structure. In contrast, the total number of deposits is a global variable that is calculated as the

```
01: int main(void *args)
02: {
03:     file f = open(args[1]);
04:
05:     oskel_wl_t* oskelPtr =
06:                     oskel_wl_alloc(&oskel_wl_initWorker,
07:                     &oskel_wl_processWorkUnit,
08:                     &oskel_wl_update,
09:                     &oskel_wl_destroyWorker);
10:
11:     oskel_wl_shared_t global = malloc();
12:     global -> data = malloc();
13:
14:     while (!feof(f))
15:       oskel_wl_addWorkUnit(oskelPtr,read(f));
16:
17:     oskel_wl_run(oskelPtr,global);
18:     oskel_wl_free(oskelPtr);
19: }
```

Figure 3.3: The main function of a typical transactional worklist application on OpenSkel.

clients execute the *oskel_wl_update()* function. For this reason, it has to be declared in the *oskel_wl_shared_t* structure along with the bank account balance.

Once an *oskel_wl_shared_t* instance is initialized and the worklist is loaded with work-units, the programmer has just to call *oskel_wl_run()*. The *oskel_wl_run()* function starts all worker threads and waits in a barrier. Figure 3.5 shows OpenSkel's internal implementation of each worker. Each worker thread coordinates the execution of the aforementioned user functions. After initialization, each worker grabs work-units with *oskel_wl_getWorkUnit()* and calls the *oskel_wl_processWorkUnit()* function until the worklist is empty. Although the *oskel_wl_getWorkUnit()* is within a transaction, its variables are not protected by transactional barriers. Instead, this function internally uses locks to access OpenSkel worklist and internal state. This is essential to decouple the worklist management from the transactional memory system, avoiding extra transaction conflicts and contention.

The *oskel_wl_processWorkUnit()* procedure is executed within transactional barriers placed by the skeleton library. This function is then translated to transactional code at compile time by any existing TM compiler such as Dresden TM [29]. This process is transparent and completely relieves the application programmer of the burden of

```
01: struct oskel_wl_private_t {
02:     long nDeposits;
03: };
04:
05: struct oskel_wl_shared_t {
06:     long balance;
07:     long totalDeposits;
08: };
09:
10: void oskel_wl_initWorker(oskel_wl_private_t* p,
11:                          oskel_wl_shared_t* s)
12: {
13:     p -> nDeposits = 0;
14: }
15:
16: void oskel_wl_processWorkUnit(void* workUnit,
17:                              oskel_wl_private_t* p,
18:                              oskel_wl_shared_t* s)
19: {
20:     long amount = *(long*)workUnit;
21:     s -> balance += amount;
22:     p -> nDeposits++;
23: }
24:
25: void oskel_wl_update(oskel_wl_private_t* p,
26:                       oskel_wl_shared_t* s)
27: {
28:     s -> totalDeposits += p -> nDeposits;
29: }
30:
31: void oskel_wl_destroyWorker(oskel_wl_private_t* p,
32:                              oskel_wl_shared_t* s)
33: {
34:     p -> nDeposits = 0;
35: }
```

Figure 3.4: A simple application code using the OpenSkel API.

having to handle transactions explicitly.

3.2.2 The OpenSkel Baseline

The OpenSkel default implementation is known as the *OpenSkel baseline*. It means that any performance optimization is applied on top of this baseline. There are two important implementation issues to be addressed in the baseline version: i) the data

```
01: void oskel_wl_worker(oskel_wl_t* oskelPtr,
02:                      oskel_wl_shared_t* global)
03: {
04:     void* workUnit;
05:     int tid = getThreadId();
06:     oskel_wl_private_t* local = oskelPtr -> locals[tid];
07:     oskel_wl_initWorker(local,global);
08:
09:     do {
10:         atomic {
11:             if((workUnit = oskel_wl_getWorkUnit()))
12:                 oskel_wl_processWorkUnit(workUnit,local,global);
13:         }
14:     } while(workUnit);
15:
16:     atomic {
17:         oskel_wl_update(local,global);
18:     }
19:
20:     oskel_wl_destroyWorker(local,global);
21: }
```

Figure 3.5: The OpenSkel internal worker pseudocode. This code releases the programmer from the burden of dealing with the worklist and placing STM barriers.

structure with which to implement the worklist and ii) how to access the worklist.

The worklist data structure can be implemented in many ways. The main target is to find a simple, efficient, and flexible enough structure to allow different optimizations. Based on that, the OpenSkel worklist is implemented as a stack under a work sharing scheme. First, work sharing provides a centralized worklist and promotes load balancing. Second, a stack is a fast structure to insert to and remove elements from it since there are no search operations for elements. In particular, because OpenSkel does not execute any search operations in the worklist, there is no need for a more complex structure. Additionally, if consecutive elements in a stack are memory correlated (i.e., work-units that access data in common), the stack may improve data locality because workers would work within the same memory region. However, this very property is a double-edged sword. Poor exploitation of this behavior could lead to high contention as workers will start competing for the same data. This side-effect could be avoided with the use of a queue data structure, as in [50], at the cost of losing the data locality property. In addition to it, a queue allows that work-units are inserted and removed at the same time as opposed to a stack.

Accesses to the worklist, implemented as a stack, can be protected by a transaction or a lock. Since an access to the worklist is short and transactional concurrent accesses to the worklist usually lead to a conflict, locks seems to be the best alternative as implemented in [50]. It means that any operations on the stack (e.g. push and pop) are protected by locks, avoiding data races when different threads try to push and pop at the same time. This might sequentialize accesses to the worklist under high contention. In particular, this effect is minimized on the OpenSkel baseline version by implementing fast locks based on compare-and-swap primitives with busy waiting to access the worklist. It is expected that under high abort ratio, this synchronization strategy will lower contention for the worklist. When existing STM applications are ported to OpenSkel, the baseline performance matches the one of the original TM version as discussed in Chapter 4.

Chapter 4

Evaluation Methodology

This chapter presents the experimental methodology used to evaluate the solutions proposed in this thesis. It first describes in detail the metrics, platforms, tools, and applications employed to obtain the performance results. Then this chapter describes and discusses how the selected applications were ported to OpenSkel. Finally, it compares the performance of the original and ported versions.

4.1 Experimental Setup

This thesis uses three metrics to evaluate performance: speedup, transaction abort ratio and last level cache (LLC) miss ratio. The speedup metric can be easily calculated by dividing the execution time of the sequential (hence transactionless) code by the execution time of the target parallel code as presented in Equation 4.1. Particularly, the execution time is measured by the *gettimeofday()* call.

$$Speedup = \frac{Execution\ Time_{\ Sequential}}{Execution\ Time_{\ Parallel}} \tag{4.1}$$

Equation 4.2 shows how the transaction abort ratio is calculated using available information provided by the STM system.

$$Abort\ Ratio = \frac{Aborted\ Transactions}{Aborted\ Transactions\ +\ Committed\ Transactions} \tag{4.2}$$

Finally, Equation 4.3 shows how to compute the LLC miss ratio. Particularly, this information is obtained through PAPI [13], a library that provides a simple interface to access hardware performance counters. It is important to notice that this tool is

available only in one of the three platforms (i.e., NUMA32). The cache miss ratio then is showed only for this specific platform in Section 5.2 when data prefetching is evaluated.

$$Cache\ Miss\ Ratio = \frac{Cache\ Misses}{Cache\ Accesses} \tag{4.3}$$

All average performance improvements reported in this thesis are calculated as the arithmetic mean across all benchmarks of the best performance improvement for each individual benchmark on a specific number of cores. This is described in Equation 4.4 where N is the number of benchmarks and B is the best performance improvement of a benchmark.

$$Average\ Performance\ Improvement = \frac{\sum_{i=1}^{N} B_i}{N} \tag{4.4}$$

The experimental tests were conducted on the platforms depicted in Table 4.1. They differ in many aspects as number of cores, memory hierarchy, processor frequency etc., providing ground for investigating the proposed optimizations in different scenarios. Figure 4.1 shows details of the memory hierarchy and topologies of these platforms. In particular, the NUMA16 platform is used exclusively in Section 7.2 in order to evaluate page memory allocation. The rest of the results were conducted on the UMA16 and NUMA32 platforms. In particular, the NUMA32 platform supports Simultaneous Multi-Threading (SMT) but this feature was disabled in the conducted experiments.

TinySTM [29] was selected as the STM platform. It can be configured with several locking and contention management strategies. In all the experiments conducted, TinySTM was configured with encounter-time locking, write-back memory update and the commit suicide contention strategy. Encounter-time locking is an eager conflict detection strategy. Once a conflict is detected, the commit suicide contention strategy immediately aborts and restarts the conflicting transaction. The same experiments have been also conducted configuring TinySTM with a commit-time locking strategy and with TL2 [26]. Since the applications and optimizations behaved similarly, these results are not shown in this thesis. All code was compiled using GCC with the -O3 option enabled. Finally, all the results presented in this thesis are based on an arithmetic mean of at least 10 runs. The standard deviation is presented later in Section 4.3.

Table 4.1: Overview of the multi-core platforms.

Characteristic	UMA16	NUMA16	NUMA32
Number of cores	16	16	32
Number of sockets	4	8	4
NUMA Nodes	1	8	4
Processor	Intel Xeon E7320	AMD Opteron 875	Intel Xeon x7560
Clock (GHz)	2.13	2.2	2.27
Last level cache (MB)	2 (L2)	1 (L2)	24 (L3)
DRAM capacity (GB)	64	32	64
Linux kernel version	2.6.18	2.6.32-5	2.6.32-5
GCC version	4.1.2	4.4.4	4.4.4

4.2 Analyzing the STAMP Benchmark Suite

To evaluate the performance trade-offs of the transactional worklists under the OpenSkel system, five applications from the STAMP benchmark suite [15] that matched the worklist pattern were selected: *Intruder*, *Kmeans*, *Labyrinth*, *Vacation* and *Yada*. Other applications from STAMP present an irregular behavior with specific characteristics which make them hard to generalize to a single skeleton, such as an arbitrary number of synchronization barriers and very fine grained transactions (*SSCA2* and *Genome*). Although *Bayes* was ported to OpenSkel, it was discarded due to its high variability in performance observed in preliminary experiments. All selected applications were executed with the recommended input data sets. *Kmeans* and *Vacation* have two input data sets, high and low contention. As *Intruder* and *Yada* only have high contention input data sets, the low contention inputs for *Kmeans* and *Vacation* were selected in order to cover a wider range of behaviors. Particularly, preliminary experiments confirm that both high and low contention input data sets do not affect the behavior of *Vacation* significantly. The original STAMP versions of these five applications were profiled according to four criteria: scalability, transaction abort ratio, L3 cache miss ratio and transaction length on the NUMA32 platform. These results are summarized in Table 4.2, which demonstrates that the applications selected span a varied range of points in the behavior space, and so provide a sound basis for evaluation.

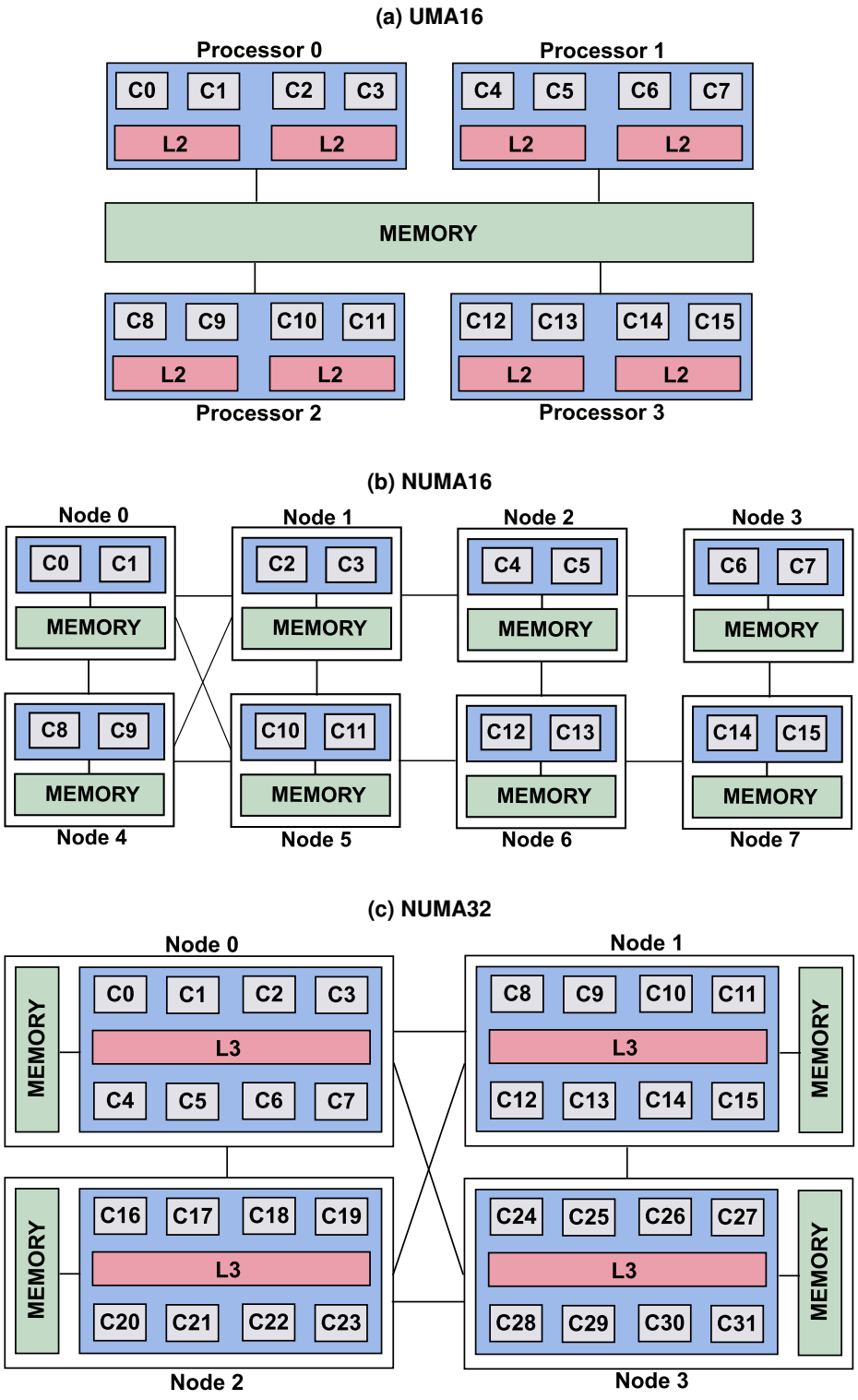

Figure 4.1: Memory hierarchy and machine topology diagrams for the platforms used in the experiments. Only shared caches are shown.

Table 4.2: Summary of STAMP application runtime characteristics on TinySTM for the NUMA32 platform.

Application	Scalable up to # Cores	Transaction Abort Ratio	L3 Cache Miss Ratio	Transaction Length
Intruder	4	high	medium	short
Kmeans	8	high	high	medium
Labyrinth	32	medium	low	long
Vacation	16	low	low	short
Yada	16	high	medium	medium

4.3 Porting to OpenSkel

To port the selected applications, single worker functions were decomposed into the ones required by OpenSkel, global and local variables were grouped into the new declared structures and a work-unit structure was declared when it was not already there. The code transformation was straightforward for all five applications. However, the original version of *Vacation* created work-units in a distributed fashion without a worklist. The inclusion of a centralized worklist in the new ported OpenSkel version led to a significant performance difference. In the next section, this will be discussed in more detail. This thesis's authors believe that implementing an application from scratch under the OpenSkel philosophy as described in Section 3.2.1 would be even more intuitive.

Regarding the transactional code transformation, it was decided to use the manually instrumented transactional code available in the STAMP benchmark suite, since [7] showed that the code generated by the OpenTM compiler achieves very similar performance. Additionally, this thesis focuses on the runtime system instead of the compiler support. Finally, the rest of this chapter focuses on the implementation details and performance trade-offs between the original TM version and the version ported to OpenSkel.

4.3.1 Performance Trade-offs of the Baseline vs. Original

Some compromises have to be made when existing transactional applications are ported to OpenSkel. These implementation decisions may impact the performance of the OpenSkel version compared to the original application.

First, the worklist data structure in OpenSkel is implemented as a stack. This will influence the ordering and amount of time to access work-units. For instance, the original *Yada* and *Intruder* implement the worklist as a heap and as a queue, respectively. Second, as mentioned before, the application has to have work-units declared as a structure. *Vacation* and *Kmeans* do not have an explicit worklist in the original benchmark, although they match the worklist pattern. Allocating and handling these work-units may introduce some overhead.

Transactional applications may split the processing of a work-unit into a few phases. Each phase is executed within a fine-grained transaction to reduce the number of aborts. For example, *Yada* and *Intruder* use more than two transactions to process each work-unit. On the other hand, to free the programmer from the burden of handling transactions explicitly, each work-unit in OpenSkel is processed within a single coarse-grained transaction. However, a single coarse-grained transaction may increase the number of aborts, since each transaction becomes longer. Fortunately, if the majority of aborts are concentrated into a single phase, combining multiple transactions into a single transaction does not become a bottleneck. Based on preliminary experiments, that is the case for *Intruder* and *Yada*. Since the other applications use only a single transaction to process each work-unit then their performance is not affected. Another issue is thread mapping and scheduling. STMs do not manage threads and thus they are left with the operating system default scheduling strategy. The Linux scheduling strategy in the selected platforms and applications tends to map threads initially following the *scatter* mapping strategy. *Scatter* distributes threads across different processors avoiding cache sharing between cores in order to reduce memory contention. However, at runtime the Linux scheduler migrates threads trying to reduce memory accesses and I/O costs. The baseline version employs a static *scatter* mapping strategy in which threads are not allowed to migrate at runtime, guaranteeing a more predictable performance.

4.3.2 Performance Analysis of the Baseline vs. Original

Based on the results in Figure 4.2, this section presents each application and compares the performance of the baseline against the original version. For all applications, the input work-units are first shuffled and then inserted in the worklist. This is done to avoid benefits from a particular input order. Table 4.3 shows the average standard deviation (i.e., the average for all number of cores) per application version and plat-

Table 4.3: Average standard deviation (in percentage %) of the execution time for the baseline and original versions of the STAMP benchmark applications for the UMA16 and NUMA32 platforms.

Application	UMA16		NUMA32	
	Baseline	Original	Baseline	Original
Intruder	0.2%	1.2%	0.8%	1.6%
Kmeans	1.8%	3.9%	2.6%	4.2%
Labyrinth	4.3%	6.5%	4.9%	7.5%
Vacation	0.8%	1.4%	1.5%	1.6%
Yada	0.7%	2.0%	1.8%	1.7%

form. It can be observed that most baseline versions present less variation than the original versions. The reason for this is twofold. Firstly, the number of transactions created on both versions are very different as explained in the previous section. This is also showed in Figure 4.3. These results also point out that the more transactions are created, the more unpredictable is the performance behavior of the observed applications. Lastly, the Linux scheduler migrates threads at runtime in the original version. However, the first factor still accounts for the most part of the variability.

Table 4.3 also shows that *Intruder*, *Vacation* and *Yada* present very low variability in performance. In contrast, the performance of *Labyrinth* and *Kmeans* varies significantly due to their intrinsic behaviors. In short, they are applications in which their behaviors are very dependent on the input data order, leading to more unpredictable performance. In the rest of this section, these details are discussed for each application.

Intruder. This application is a signature-based network intrusion detection system (NIDS). It matches network packets against a pre-configured set of known intrusion signatures. The application has three phases: packet capture, stateful NIDS avoidance countermeasures and signature matching. First, packets are captured from a queue. In the second phase, packets belonging to the same flow are processed by the same thread. A shared tree keeps information used in this phase. The last phase is the signature matching which basically involves string searching [15]. In the OpenSkel version, phases two and three are executed in a single transaction unlike the original version. One could expect that the increase in the transaction size could lead to an increase in the number of transaction aborts. However, the third phase involves mostly

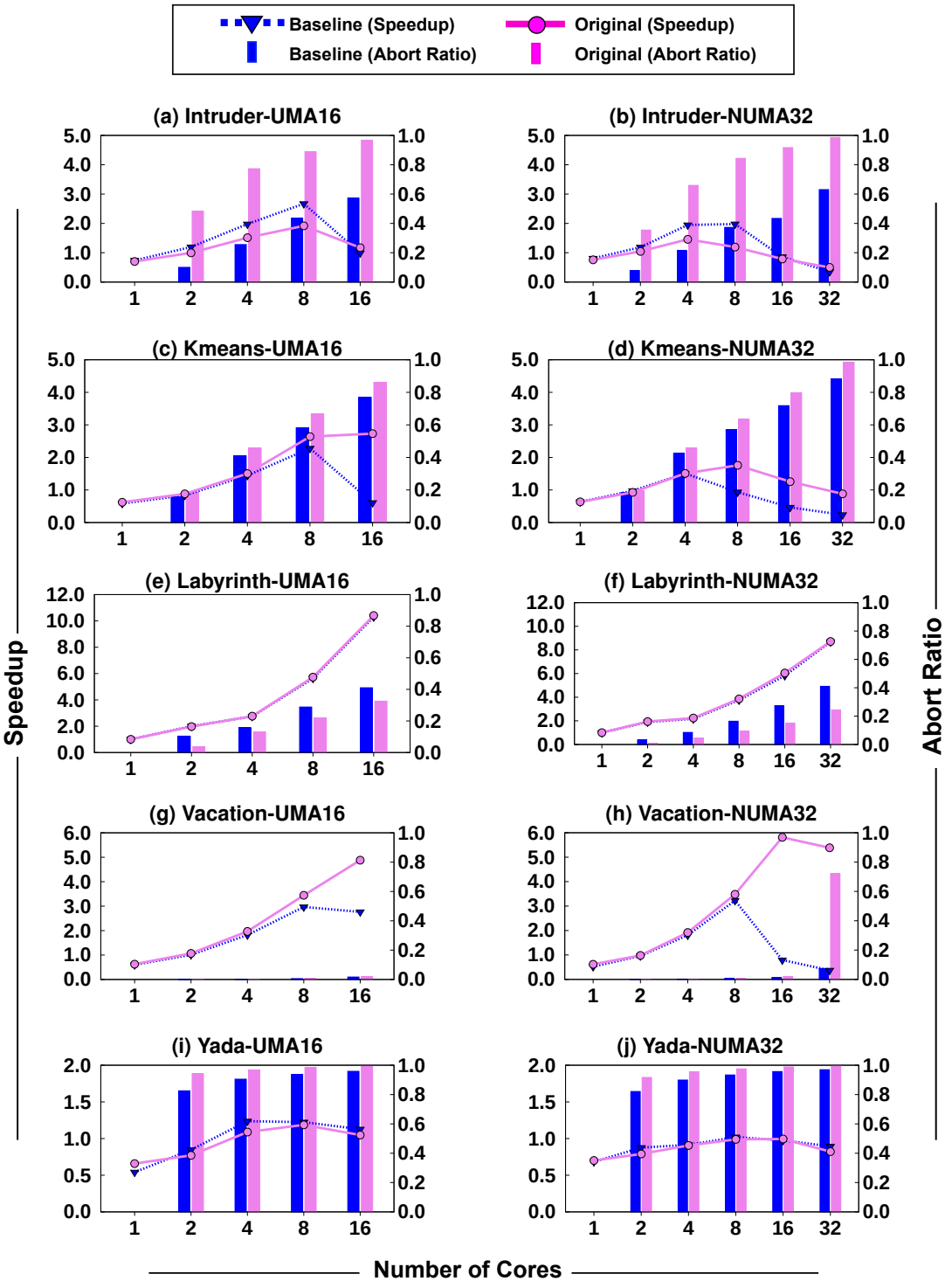

Figure 4.2: Comparison between the baseline and original versions of the STAMP benchmark applications for the UMA16 and NUMA32 machines.

searching and not update operations such that combining phases two and three does not increase the number of aborts. In fact, as depicted in Figures 4.2a and 4.2b, the baseline exhibits better performance than the original version.

Intruder computes a large number of small transactions that conflict frequently imposing high contention to access the worklist. Figures 4.2a and 4.2b show that the original version exhibits significantly higher abort ratio than the baseline. This is also observed in Figures 4.3a and 4.3b in which the total number of transactions created by the original version is much higher than the baseline version. In an eager transactional memory system, the contention on the worklist is even higher due to the eager abort of transactions. Combined with the fact that the original version protects the worklist accesses within transactional barriers, it confirms that the baseline has a more efficient implementation compared to the original version. Even so, the baseline *Intruder* could not scale beyond 8 cores on both machines due to contention in accessing shared data.

Kmeans. This is a clustering algorithm that tries to group similar elements into K clusters. It iterates over a set of elements and calculates the distance between these elements and their centroids. At the end of each step, threads synchronize and a master thread recalculates all the centroids based on the new distances calculated. All elements are contained in an array and on each step, each thread processes chunks of elements. The OpenSkel baseline version implements each step as a worklist and each element as a work-unit. At each step, the worklist has to be re-populated with all work-units. Unlike *Vacation*, the introduction of a worklist itself did not present a significant negative performance impact since chunks are also grabbed in a centralized fashion in the original version. However, the introduction of the re-population step makes the baseline slower than the original version on average, as shown in Figures 4.2c and 4.2d. Regarding the abort ratio, the baseline version presents a slightly lower abort ratio as it avoids the use of transactions to grab work-units. In contrast, the *Kmeans* original version grabs new chunks from a global variable within a transaction, increasing the number of conflicts. Figures 4.3c and 4.3d show that this increase in the number of transactions did not contribute to degrade the performance of the original version. Finally, both implementations do not scale beyond 8 cores as depicted in Figures 4.2c and 4.2d.

Labyrinth. This application implements a routing algorithm that finds paths through a grid between source and destination points that do not overlap [15]. On each iteration

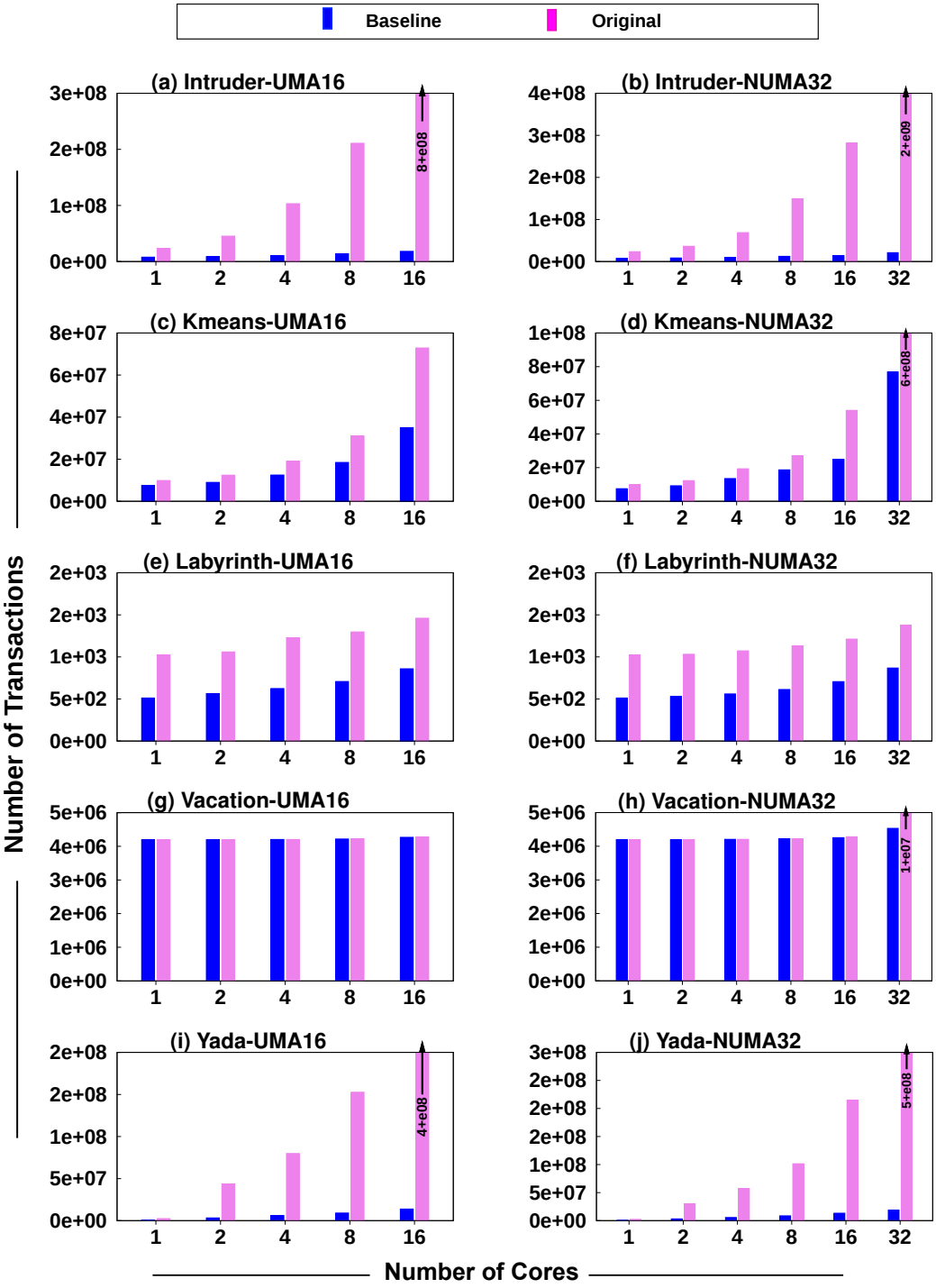

Figure 4.3: Total number of transactions in the baseline and original versions of the STAMP benchmark applications for the UMA16 and NUMA32 machines.

of the algorithm, a thread grabs a pair of source and destination points from a shared queue. It then creates a local copy of the grid instead of updating it directly. After it routes a path between the pair of points, it reads all grid points that belong to the path and tries to commit by adding the new path to a global list. The routing process and the addition to the path list is encapsulated into a single transaction.

Figures 4.2e and 4.2f show that the baseline and original versions present similar performance. This stems from the fact that *Labyrinth* has low contention to the worklist due to its long and few transactions. Furthermore, the baseline code is very similar to the original one. However, the abort ratio of the baseline is higher since each access to the worklist is not within a transaction. The baseline version thus has half the number of transactions but almost the same amount of aborts, as depicted in Figures 4.3e and 4.3f. As a consequence, the baseline version presents a higher abort ratio.

Vacation. This application emulates an on-line travel reservation system [15]. Each client has a fixed number of requests generated randomly in a distributed fashion. Each request is enclosed in a single transaction that performs the accesses to the database server. The system keeps track of customer reservations through a set of shared trees. The baseline version introduced a worklist and transformed the requests into work-units. As *Vacation* executes short transactions, the new centralized worklist becomes a bottleneck. These code transformations led to a semantically different version since all work-units are generated sequentially by a single worker thread. This makes the baseline much slower than the original version on more than 4 cores as observed in Figures 4.2g and 4.2h.

The abort ratio and the total number of transactions are very similar for both versions as presented in Figures 4.3g and 4.3h. It stems from the fact that transactions in *Vacation* rarely conflict. An exception is the NUMA32 platform with 32 cores in which the abort ratio increases exponentially for the original version. In contrast, the abort ratio in the baseline version continues low due to the high contention to access the worklist. This contention forces many threads to waste time waiting to grab a work-unit from the worklist instead of executing transactions concurrently.

Yada. This is an implementation of Ruppert's algorithm for Delaunay mesh refinement [15]. It consists of a shared graph structure where each node is a triangle, a set of segments that delimits the mesh boundary and a shared worklist of bad triangles (i.e., triangles that do not satisfy a quality criteria). The refinement is an iterative process

over a worklist of bad triangles that removes these bad triangles from the mesh and determines the affected neighborhood. It repeatedly replaces the affected neighbor triangles with new ones (re-triangulation phase), adds new bad triangles, and continues until the worklist is empty. As seen in Figures 4.2i and 4.2j, the baseline and original version behave similarly. Both versions do not scale due to the high abort ratio and contention. However, the baseline version presents a slightly lower abort ratio. It stems from the fact that accesses to the worklist in the baseline version are done without transactions, reducing the number of conflicts.

Although the number of transactions on the original version is much higher compared to the baseline version, as shown in Figures 4.3i and 4.3j, the abort ratio is high on both cases. This leads to roughly the same performance for both versions.

In short, these experimental results show that the baseline version can achieve performance improvements of up to 28%, with an average of 3%, over the original TM version for a 16-core UMA system and up to 26%, with an average of 3%, for a 32-core NUMA system. The only exception is *Vacation* as discussed before. This thus indicates that, on average, existing applications ported to OpenSkel do not lose performance and can enjoy the performance improvement provided by the automatic optimizations presented in the next chapter.

Chapter 5

Pattern-Oriented Performance Optimizations

A skeleton-driven approach makes available useful information about the pattern of the applications at compile time and runtime. This information allows OpenSkel to provide a set of performance optimizations. In this chapter, the implementation details of several pattern-oriented performance optimizations to the transactional worklist skeleton are presented. These optimizations have been proposed and used separately in other contexts [4, 63, 69, 72]. However, in order to apply them on a transactional worklist skeleton, they had to be adapted or re-designed within a completely new approach. Additionally, these optimizations can be enabled simultaneously and transparently in a single transactional skeleton framework.

5.1 Employing Pattern-Oriented Optimizations

5.1.1 Stealing Work-Units

The first implemented optimization is *work stealing* (WS), employed in many systems such as Intel TBB [69], Cilk [8] and the Java Fork/Join framework [53]. It tackles the contention to access the worklist which occurs with increasing number of work-units and worker threads. It exploits the knowledge that work-units can be executed in any order and by any core in the system.

The WS optimization is composed of a set of private worklists, with the initial number of work-units being split among workers in a round-robin fashion. As shown in Figure 5.1b, each worker has its own privatized worklist in which it inserts and

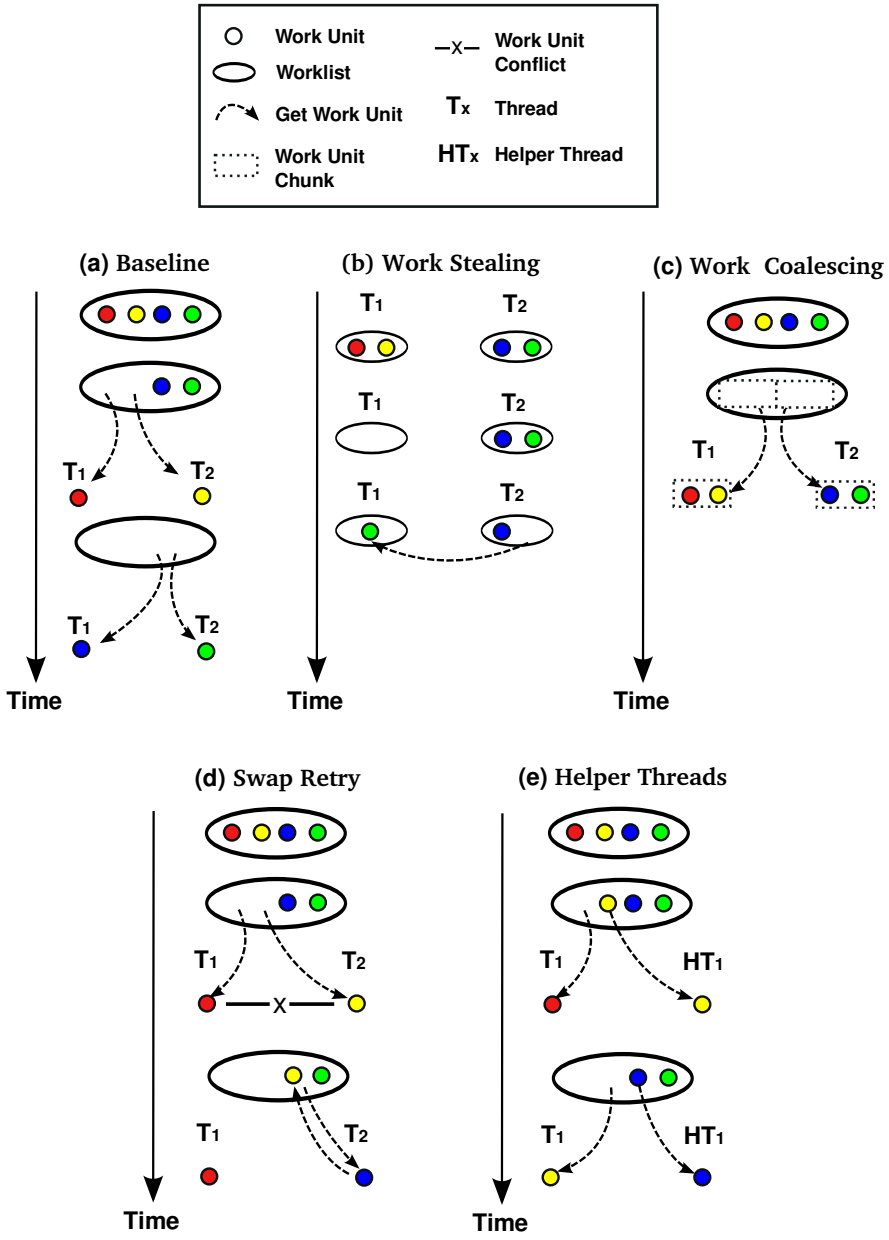

Figure 5.1: Pattern-oriented performance optimizations.

removes work-units. When a local worklist runs out of work-units it steals work-units from a victim worker.

In the OpenSkel framework, this WS optimization is implemented within the *oskel_wl_getWorkUnit()* function. This function call checks if there is any work-unit in its private worklist. If the worklist is empty it tries to steal work-units from another thread. This stealing policy is a standard implementation that uses busy-waiting locks to synchronize the victim and the thief workers, copies half of the work-units to the thief worker, removes them from the victim worker and frees both worker threads. In

order to identify when there are no work-units left, each worker increments a global counter when it is trying to steal. If the stealing process is successful, it decrements this counter. Otherwise it checks if the counter reached the total number of threads, that is, if there are no work-units to be stolen. In that case, the worker thread finishes its execution. However, if the counter is different from the total number of threads, the thief worker tries to steal from another randomly selected victim. This optimization tries to reduce the contention to the worklist. As a side-effect, when there are just a few work-units left and several idle worker threads, it may take a longer time before an idle worker steals new work-units. This is due to the randomness in selecting a victim worker.

5.1.2 Coalescing Work-Units

The second optimization exploits the fact that two work-units executed in a single transaction are as semantically correct as if executed in separate transactions. This optimization, called *"work coalescing"* (WC), is based on the technique proposed in [63]. It executes two work-units that would be executed in different transactions, within the same transaction. If a conflict is detected, the whole transaction is rolled-back. Figure 5.1c exemplifies how this optimization works.

The main benefits expected from coalescing work-units are a reduction of contention to the worklist and exploitation of memory locality. As worker threads grab two work-units at once, they take more time to access the worklist again. This reduces the contention over the worklist. As a side-effect, these longer transactions may increase the number of conflicts. Additionally, if two work-units are memory correlated, WC increases cache efficiency and reduces TM overhead. The former is done naturally by keeping the same data in the cache, instead of starting a new transaction and possibly fetching the same data again. The latter avoids that the same data is buffered and validated again by the TM system.

The OpenSkel framework implements the WC optimization within the main worker function (see *oskel_wl_worker()* in Figure 3.5). Instead of calling the *oskel_wl_ getWorkUnit()* just once in a transaction, the worker thread calls it twice and stores the returned work-units in a temporary buffer. Then it calls the *oskel_wl_processWorkUnit()* function to process the first work-unit. If this succeeds, then the worker thread tries to process the second work-unit. Finally, after the successful completion of the second

work-unit, it commits the transaction and clears the temporary buffer. If the worker thread produces any new work-units, they are first stored in a temporary list and then transfered to the worklist only after the transaction is committed. On a transaction abort, the worker thread resets the temporary list and calls the *oskel_wl_processWorkUnit()* function to start processing from the first work-unit again.

5.1.3 Swapping Work-Units

Swap retry (SR) is the third implemented optimization. This is based on the *steal-on-abort* technique proposed in [4]. Usually, when a transaction aborts, a STM re-executes the transaction hoping that the conflict will not re-occur. The other common available alternatives to the STM are to assign a higher priority to the transaction or wait for a time interval (e.g., back-off strategy) before re-executing it. Nevertheless, a STM does not have an alternative to try to execute a different transaction, unless explicitly implemented by the programmer through a retry function [73].

In a transactional worklist skeleton, it is possible to try a different work-unit since the skeleton has full control over the worklist and work-units can be executed in any order. This optimization takes advantage of this high level information. As shown in Figure 5.1d, when a transaction aborts, this optimization swaps the current work-unit with another one before it re-executes. In particular, in the OpenSkel implementation, this optimization is implemented within the *oskel_wl_getWorkUnit()* function. Each time a worker thread tries to grab a new work-unit, it checks if the current work-unit has been executed. Otherwise it means that the worker thread has aborted and restarted. In this case, it swaps the current work-unit with a randomly chosen work-unit in the worklist. In this way, transactions that keep aborting can be postponed and executed later.

Swap retry is employed to reduce the number of aborts. However, one side-effect is to reduce cache prefetching. When a transaction aborts, it works as a cache prefetcher for its own re-execution. In contrast, if the work-unit is swapped, then this prefetching effect is lost since the transaction will restart and access different data. To alleviate this problem, a parameter to limit the number of retries was introduced. The swapping is only actually done after a transaction reaches a particular number of retries. It also avoids an excessive number of swaps that may increase contention to access the worklist. However, this parameter is application and system dependent, making it hard to

Table 5.1: OpenSkel helper thread internal programming interface.

OpenSkel Helper Thread Internal Functions
helperThread_t* **oskel_wl_ht_alloc**(void* **(*htFunction)(void*)**, void* **htArgs**);
Create and spawn a helper thread for the calling thread.
void **oskel_wl_ht_signal**(helperThread_t* **ht**, void* **sharedBuffer**);
Signal a helper thread sending the next *work-unit* to be prefetched.
void **oskel_wl_ht_wait**(helperThread_t* **ht**);
A helper thread waits in a barrier until the next work-unit is signaled.

determine an optimal value.

5.1.4 Employing Helper Threads

Another pattern-oriented optimization is to perform data prefetching using automatically created *helper threads* (HT). They are auxiliary threads that run concurrently with a main thread. Their purpose is not to directly contribute to the actual program computation, which is still performed in full by the main thread, but to facilitate the execution of the main thread indirectly. Typically modern multi-cores have at least one shared level of cache among the cores, so that HTs may try to bring data that will be required by the main thread into this shared cache ahead of time. Helper threads have previously been developed in software [72] and hardware [21, 80].

TM applications have a number of characteristics that render the use of HTs appealing. First of all, some transactional applications do not scale up to a large number of cores because the number of aborts and restarts increases. If more cores are available, they can be used to run HTs instead of more TM threads and thus improve the performance of the applications. Another characteristic of STM applications is the high overhead and cache miss ratio of transactional loads and stores. This suggests that HT can more easily stay ahead of the main thread while effectively prefetching for it.

Unfortunately, a STM does not have the required information to implement HT on its own. The worklist skeleton, on the other hand, provides two key information to make HTs feasible: when to start a HT and which data to prefetch. As observed in Figure 5.1e, every time a worker thread starts computing a work-unit, a HT should start computing the next work-unit assigned to the worker in the worklist.

Helper thread code is generated and instrumented in the same way compilers like the Tanger [30] and OpenTM [7] would do for STM systems. However, instead of function calls to the STM system, they use modified functions for reading and writing shared variables. Instead, as with normal TM threads, every time a HT has to access a shared global variable it has to use special functions to redirect accesses to the internal metadata structures managed by OpenSkel runtime system. This is a somewhat similar approach to hardware HT, but we do not rely on hardware to perform the buffering. As HTs do not change the state of the application, each write to a global variable is done in its local entry in a hash table rather than in the actual memory location. If the same variable is read after being written, the value will be extracted from the hash table instead of the actual memory location. This enables HTs to follow the correct path of control and hopefully prefetch the correct data. However, if a transaction modifies shared data, the HT may go down the wrong path, possibly prefetching wrong data or even worse, raising exceptions that could crash the whole application. OpenSkel HT thus implement a transparent mechanism to deal with exceptions. If an exception is raised, the OpenSkel library aborts the helper thread and restarts it in a wait barrier (*oskel_wl_ht_wait()*).

Each main thread calls *oskel_wl_ht_alloc()* to start a new helper thread. The OpenSkel system first checks whether there are cores that share any level of cache memory and if this is true, it schedules the main thread and the helper thread to cores that share the same level of cache. To reduce cache pollution due to inefficient prefetching, a lifespan parameter (i.e., number of words prefetched per work-unit) and a limit to the hash table size are employed.

Figure 5.2 shows how helper threads interact with the worklist skeleton using the internal HT API provided by OpenSkel (see Table 5.1). The *oskel_wl_ht_signal()* and *oskel_wl_ht_ wait()* are placed inside the *oskel_wl_getWorkUnit()*. When the calling thread is a helper thread it waits, otherwise it sends a signal. Each helper thread is a copy of the *oskel_wl_processWorkUnit()* function, but instrumented with the HT calls. However, instead of grabbing a work-unit directly from the worklist or starting processing of a range of elements, it waits for a signal from the transactional *oskel_wl_processWorkUnit()* function with the work-unit to be processed. This communication between the main thread and the helper thread is done through an efficient lock-free shared buffer. The main thread just writes the next work-unit to a shared buffer and the cache coherence protocol does the rest, updating the value in the shared cache level. Since the *oskel_wl_ht_wait()* implements a busy wait, as soon as the value

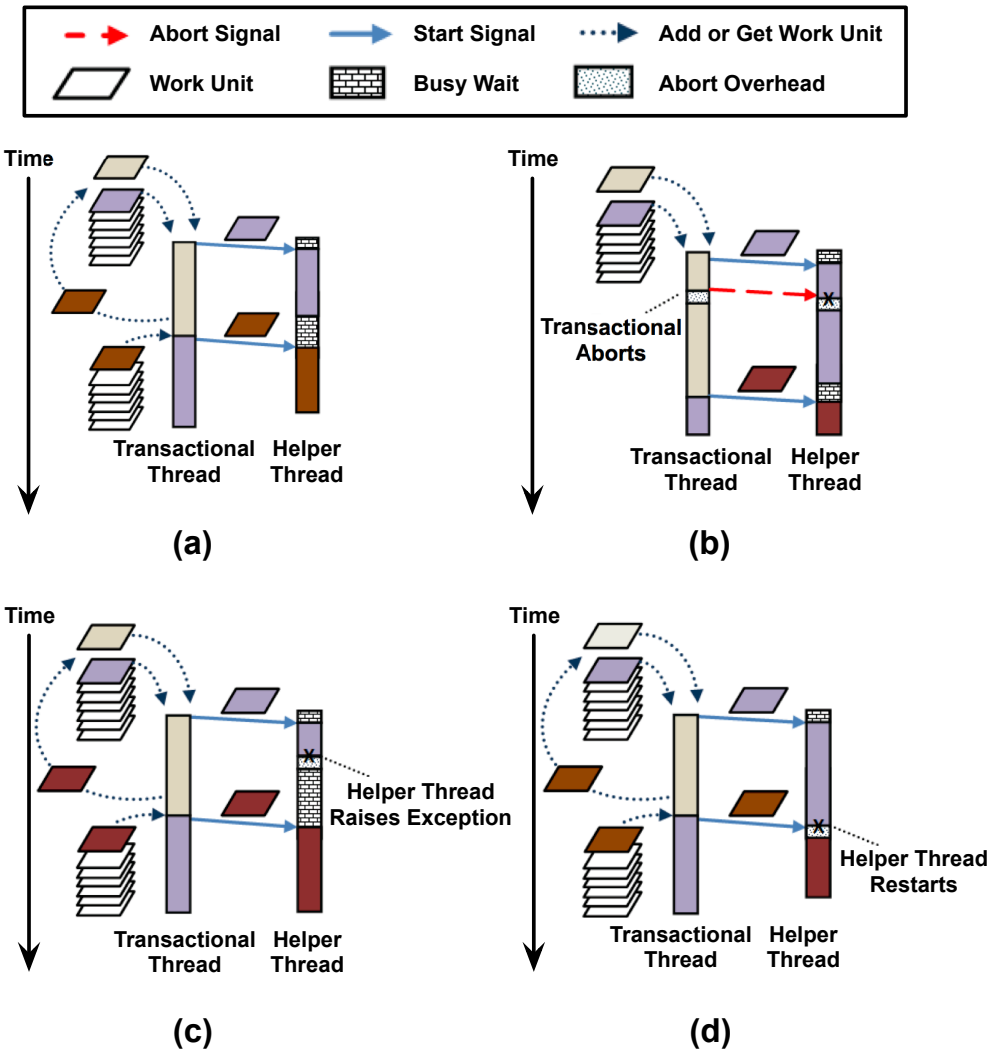

Figure 5.2: Interaction between Helper Threads (HT) and the worklist skeleton: (a) a correct execution from both transaction and HT; (b) a transaction abort results in a HT abort and restart; (c) when a HT raises an exception, it transparently aborts and restarts; and (d) when a HT is too slow or seems stuck in a infinite loop, the transactional thread forces it to abort and restart with the next work-unit.

is updated by the coherence protocol, the helper thread is able to detect that a new work-unit should be executed. The use of locks is avoided for the following two reasons: locks introduce a large overhead to the main thread, eliminating any chance of performance improvement; to guarantee that the helper thread will not be scheduled by the operating system or sleep for an undetermined period of time, missing signals and starting prefetching too late.

In order to guarantee that the HT is always executing the correct next work-unit,

Table 5.2: Summary of the pattern-oriented optimizations characteristics.

Optimization	Pros	Cons	Parameters
Work Stealing	• improve locality • reduce contention	• increase imbalance for few work-units	-
Work Coalescing	• improve locality • reduce contention	• increase conflicts • load imbalance	-
Swap Retry	• reduce aborts	• increase contention • reduce prefetching	• number retries
Helper Threads	• reduce LLC misses	• pollute cache	• lifespan • buffer size

the transactional or main thread always keeps two work-units at the same time. While it processes the former work-unit, it signals the HT with the latter one. Thus, when the transactional thread calls *oskel_wl_getWorkUnit()* for the first time, it grabs two work-units. For all the following calls to *oskel_wl_getWorkUnit()*, it grabs just one work-unit, sends it to the HT and starts executing the previous one as shown in Figure 5.2a. Meanwhile, the transactional thread can insert new work-units into the work-list with an *oskel_wl_addWorkUnit()* call. Without keeping two work-units at the same time per transactional thread, this would lead to inefficient data prefetching as the HT would be prefetching data for the wrong thread. Figure 5.2b, 5.2c and 5.2d show how OpenSkel deals with transaction aborts and helper thread incorrect executions.

Table 5.2 summarizes the pros, cons and input parameters of each pattern-oriented optimization. In particular, WS and WC optimizations tackle the same problem but present different side-effects. Additionally, they do not have input parameters to be determined. SR is the only optimization that is designed to reduce the number of aborts. However, WS may also reduce the abort ratio in a specific scenario discussed later in the following section. Finally, HTs have two input parameters and focus on reducing the cache miss ratio.

5.2 Analysis of the Pattern-Oriented Optimizations

In this section, the performance of the proposed pattern-oriented optimizations are analyzed. For all transactional worklist applications, the input work-units are shuffled

before start computing them. This is done to avoid benefits from a particular input order. Since the average standard deviations for each application/optimization are similar to the baseline version ones (see Table 4.3), they are not shown in this section. For the *helper threads* optimization half the cores run transactional threads and the other half run helper threads. Due to this, the results for *helper threads* range from 2 to the maximum number of cores. Nevertheless, HT are expected to be profitable only for 8 or more cores, depending on when the baseline version stops scaling.

In the rest of this section, the performance benefits of individually applying each optimization are first analyzed. Then, these results are summarized and compared altogether in Section 5.2.5.

5.2.1 Evaluating Work Stealing

The *work stealing* optimization is effective in reducing the contention to access the worklist. According to Figure 5.3, it improves the performance of most applications, up to 102% for *Intruder*. With the increasing number of cores, applications that execute small to medium transactions are bound to stop scaling due to contention to a centralized worklist. *Work stealing* significantly reduces the contention in the worklist, splitting it between threads. The exception is *Labyrinth*, which executes a small number of long transactions, as shown in Figures 5.3e and 5.3f. When the number of threads increases the time to search for a victim with a non-empty worklist also increases. This stems from the fact that work stealing selects a victim in a random fashion. Since Labyrinth is left with just a few long transactions in the end of its execution, worker threads may have zero or just one work-unit to be stolen. Thus this searching process takes longer to converge, that is, to find a suitable target victim.

For *Kmeans* and *Vacation*, *work stealing* improved significantly their performance, by respectively 35% and 47%, over the baseline version. In particular, as aforementioned, when Vacation was ported to OpenSkel, the introduction of the worklist itself became a bottleneck, alleviated by the WS optimization. Nevertheless, Figures 5.3d and 5.3h show that these applications were not able to scale up to 32 cores in the NUMA32 platform. It then turns out to be an application/STM scalability limitation rather than a bottleneck in the worklist itself.

The internal implementation of the OpenSkel worklist as a stack enabled the exploitation of a specific property of *Yada*. Each new bad triangle added to the worklist is correlated to the previous one. Since the *work stealing* optimization has independent

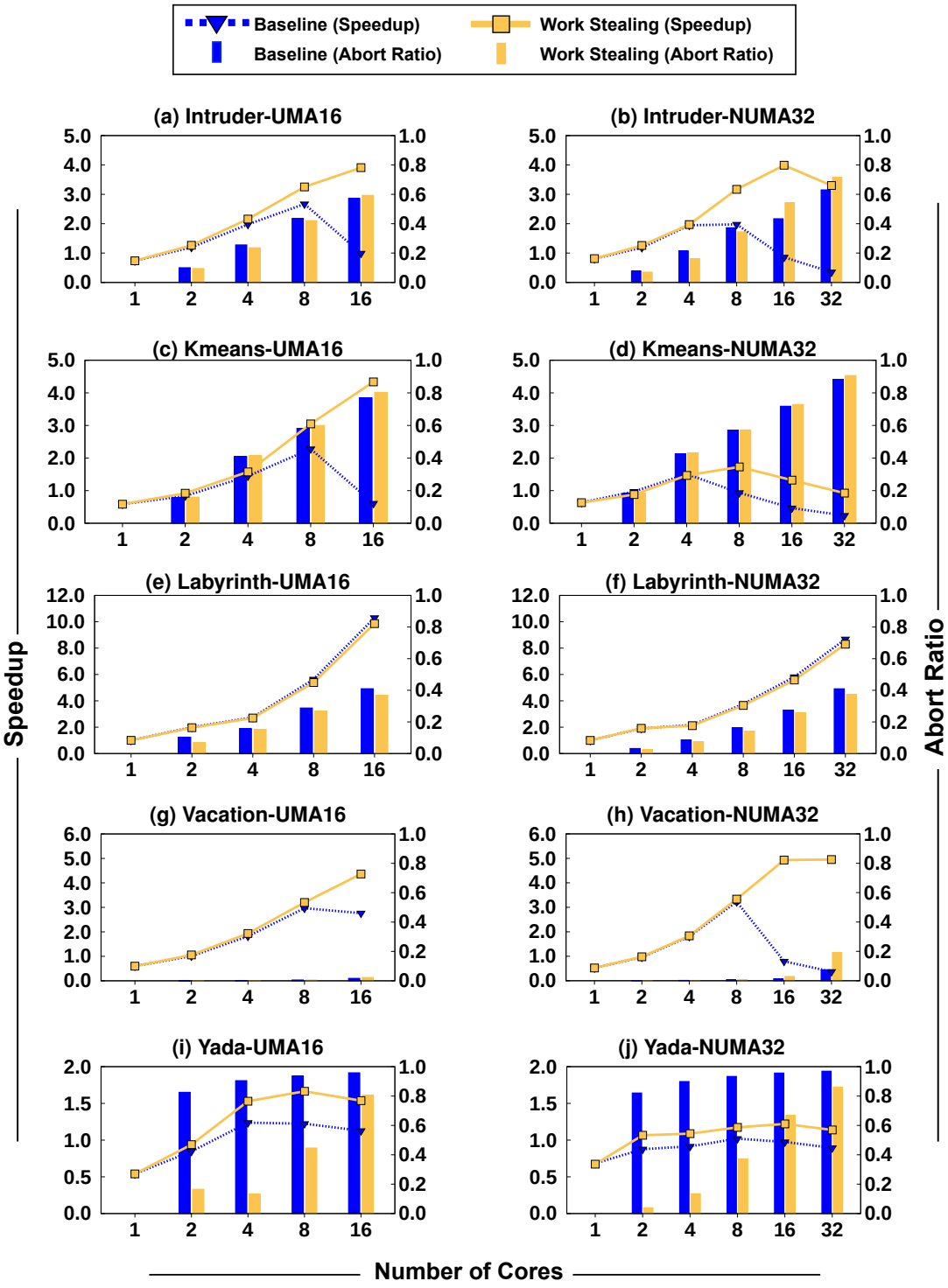

Figure 5.3: Comparison between the baseline version and the work stealing optimization on the STAMP benchmark applications for the UMA16 and NUMA32 platforms.

stacks for each thread, this leads to two beneficial effects. First, threads stop processing triangles in the same neighborhood, avoiding conflicts. Second, there is a natural prefetching mechanism as a thread will keep working in the same neighborhood. As depicted in Figures 5.3i and 5.3j, WS was able to reduce the number of aborts of *Yada*, and achieve performance improvements of up to 35% over the baseline version in the UMA16 platform.

5.2.2 Evaluating Work Coalescing

This optimization can reduce contention to the worklist at the cost of increasing aborts. As expected, Figures 5.4e and 5.4f show that *Labyrinth* had no improvement since combining long transactions within a single transaction leads to high abort ratio. On the other hand, transaction aborts are not an issue for *Vacation*, the abort ratio being less than 20% on both platforms. This enabled the *work coalescing* optimization to perform 32% better than the baseline due to a reduction in the contention to the worklist as depicted in Figure 5.4g.

In the NUMA32 platform, the contention problem becomes even worse due to the increase in the number of remote accesses to the worklist. In this case, Figures 5.4b and 5.4d show that *work coalescing* was able to add up to 16% for *Intruder* and 4% for *Kmeans* as they present high contention. This shows that in some cases it is profitable to reduce the contention to access the worklist even when the transaction abort ratio is increased. However, WC did not alleviate the contention for *Vacation* in the NUMA32 platform. *Vacation* has very short transactions, such that even combined work-units are not long enough to hide the latencies to access the worklist in a NUMA platform. As a consequence, the contention to access the worklist is still high, preventing any performance improvement.

This optimization was not able to exploit any memory locality due to the fact that consecutive work-units are not correlated in most of the applications, except for *Yada* as shown in Figure 5.4i. *Work coalescing* performed 7% faster than the baseline for *Yada* in the UMA16 platform. This stems from the fact that in *Yada* work-units generate new work-units that are memory correlated. In this case, combining two work-units also improves cache efficiency.

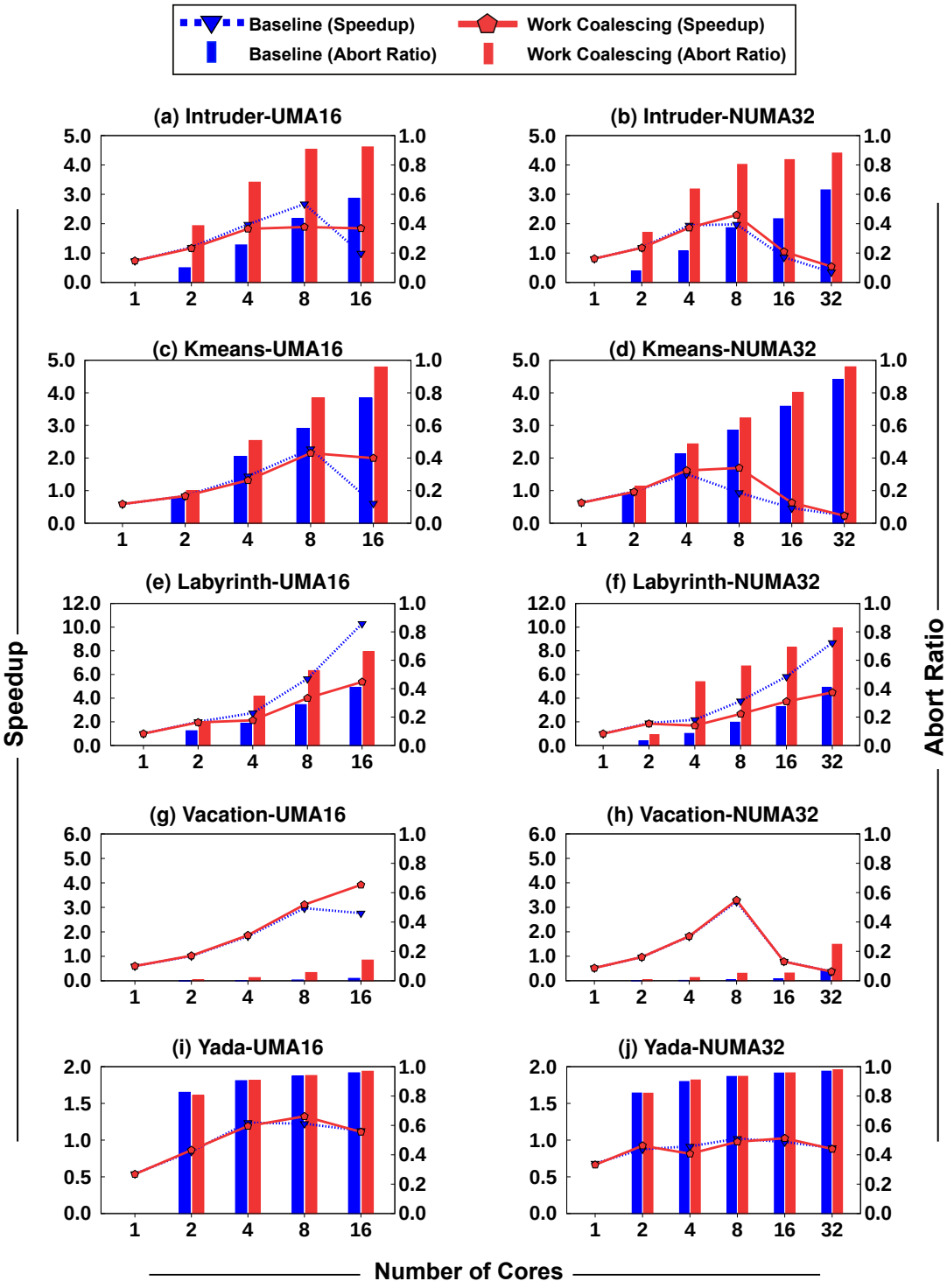

Figure 5.4: Comparison between the baseline version and work coalescing optimization on the STAMP benchmark applications for the UMA16 and NUMA32 platforms.

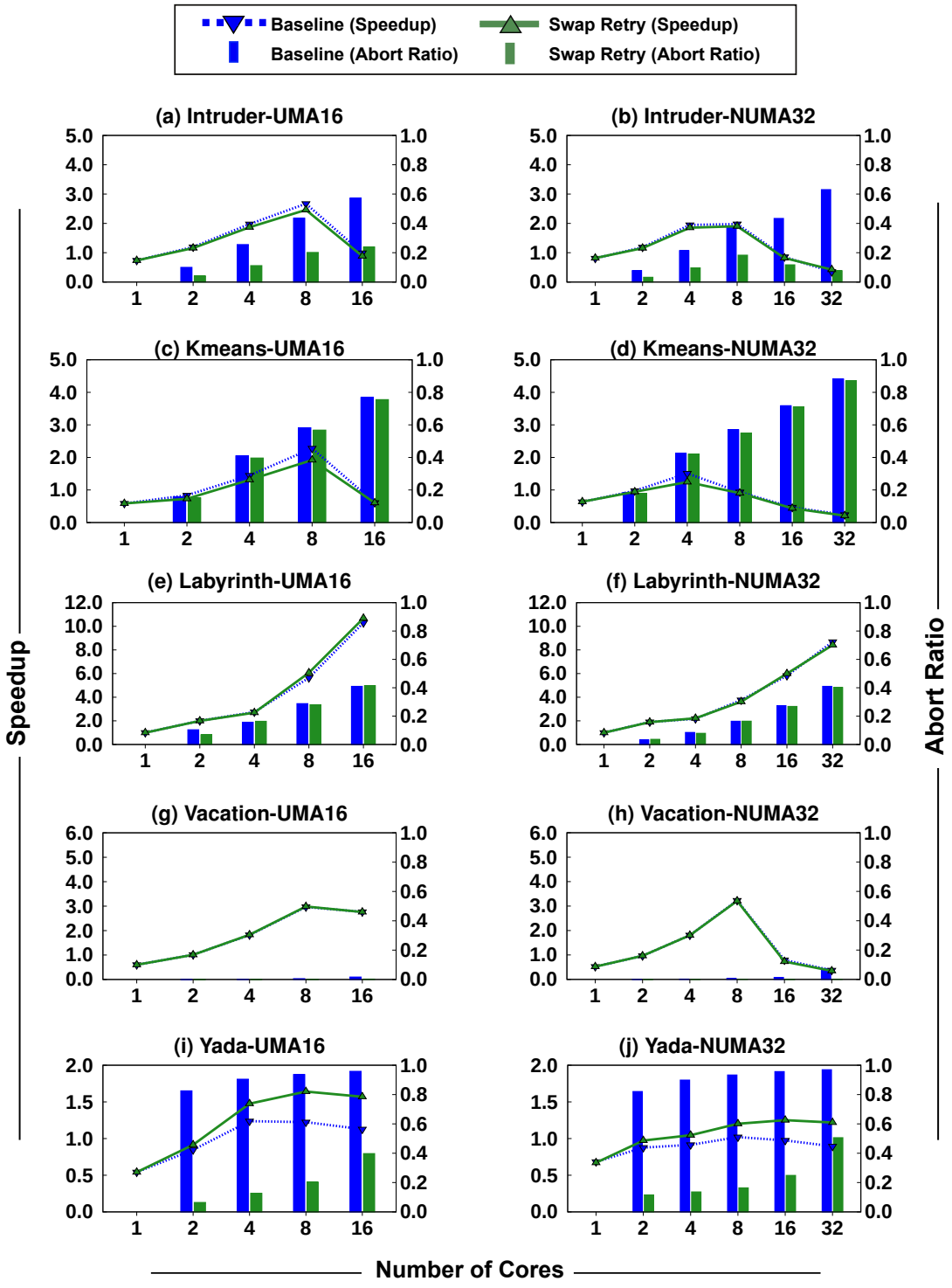

Figure 5.5: Comparison between the baseline version and swap retry optimization on the STAMP benchmark applications for the UMA16 and NUMA32 platforms.

5.2.3 Evaluating Swap Retry

This is an optimization that tackles transaction aborts by swapping conflicting work-units to other available ones. In the results presented in Figure 5.5, the parameter to limit the number of retries was set to 2. It avoids that work-units are swapped eagerly, reducing cache efficiency if a transaction is successfully committed in the following re-execution. According to Figures 5.5i and 5.5j, SR improves the performance of *Yada* by up to 33%. *Yada* presents a high abort ratio enabling SR to fit in nicely reducing the abort ratio. Since new inserted work-units are spatially correlated in *Yada*, SR also avoids having threads working on the same region.

The *swap retry* optimization adds accesses to the global worklist for each swap operation. Under high abort ratio, these extra accesses will contribute to increase the contention to the worklist. Thus any performance benefit attained from reducing transaction aborts will be outweighed by the contention overhead. Due to this behavior, SR does not provide any performance improvement to *Kmeans* and *Intruder* as depicted in Figures 5.5a-d. Unless the contention to the worklist is tackled by another optimization, these applications can not exploit the benefits from *swap retry*.

Vacation has a very low transaction abort ratio, thus *swap retry* is rarely invoked and does not impact on performance. Finally, *Labyrinth* shows a slight 4% performance improvement in the UMA16 platform as observed in Figure 5.5e.

5.2.4 Evaluating Helper Threads

The performance of helper threads is evaluated by using also the *last level cache miss ratio* as shown in Figure 5.7. It allows to measure the effectiveness of prefetching data on the last level cache. All miss ratios are computed based on the cache accesses and misses measured on a single core since it is representative of all cores in the system. As aforementioned, hardware performance counters are accessed through the PAPI [13] interface to compute the miss ratio. Unfortunately, this tool is available only in the NUMA32 platform. In the results presented in Figures 5.6 and 5.7, the lifespan was set to 1000 words and the buffer size to 32. Backed by preliminary results, these values showed that on average they present a reasonable trade-off between aggressive prefetching and cache pollution.

Figure 5.6 first compares the benefits of using helper threads with the baseline version based on the speedup and abort ratio. When helper threads are used, there is

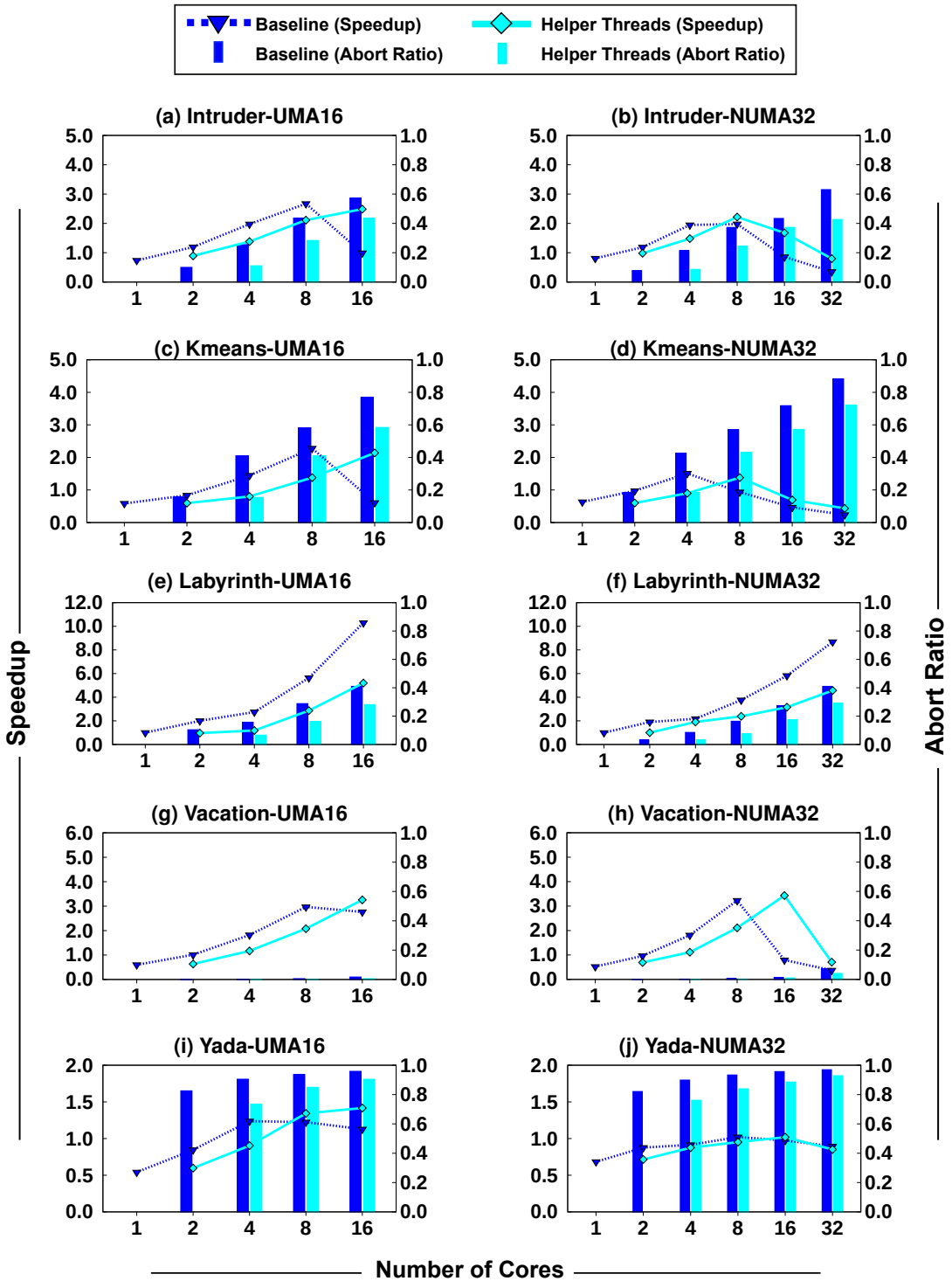

Figure 5.6: Comparison between the baseline and helper threads optimization on the STAMP benchmark applications for the UMA16 and NUMA32 platforms.

one auxiliary thread for each worker thread and both are placed on different cores as close as possible to profit from shared caches. Then, Figure 5.7 presents the results considering the cache miss ratio for the NUMA32 platform.

This optimization can be profitable only when an application stops scaling, leaving idle cores to run helper threads. Even so, the presented results include *helper threads* for all numbers of cores. Figures 5.6e and 5.6f show that *Labyrinth* scales up to the maximum number of cores. *Labyrinth* thus cannot be improved by HTs in the selected platforms. For the other applications, *helper threads* performed up to 20% faster than the baseline version. *Vacation* exhibits substantial cache miss ratio, alleviated with HTs which improved performance up to 10% according to Figure 5.6g.

Figure 5.6i shows that *Yada* benefits from the use of *helper threads* and in fact this benefit increases as the number of cores in the UMA16 platform is increased. As the abort ratio does not increase proportionally to the number of concurrent transactions, the abort ratio per thread is actually reduced. On the NUMA32 platform, *helper threads* increase the memory pressure over the shared L3 memory thus degrading performance. An exception is Intruder that sees an improvement of 12% over the baseline version as depicted in Figure 5.6b. Since it stops scaling with only four cores, adding more four *helper threads* leave only one main thread and one HT per node. That is, each L3 cache is shared by non-concurrent threads, and so avoids having competing transactions evicting each others cache lines.

As it can be noticed in Figure 5.7, by applying helper threads the last level cache miss ratio was reduced in all applications, except *Labyrinth*. This confirms the effectiveness of helper threads: triggering future cache miss events far enough in advance by the main thread reduces the memory miss latency. Although the miss ratio was decreased, such improvement did not significantly reflect well on the overall performance of all applications. In fact, there are performance improvements compared to the baseline version from 2 to 8 worker threads on most of the applications. However, they did not deliver any performance gains with 16 worker threads (*i.e.*, 16 worker threads + 16 HTs).

In Figure 5.7a, *Intruder* showed important cache miss ratio reduction when applying HTs. This led to a performance improvement when *Intruder* stops scaling with 8 worker threads. Helper threads did not improve the performance of the baseline version of *Kmeans* (Figure 5.7b) although the cache miss ratio was reduced. It is an iterative application that alternates between a sequential and a parallel phase, the latter implemented as a worklist. At the end of each phase, the worklist is empty and it is re-

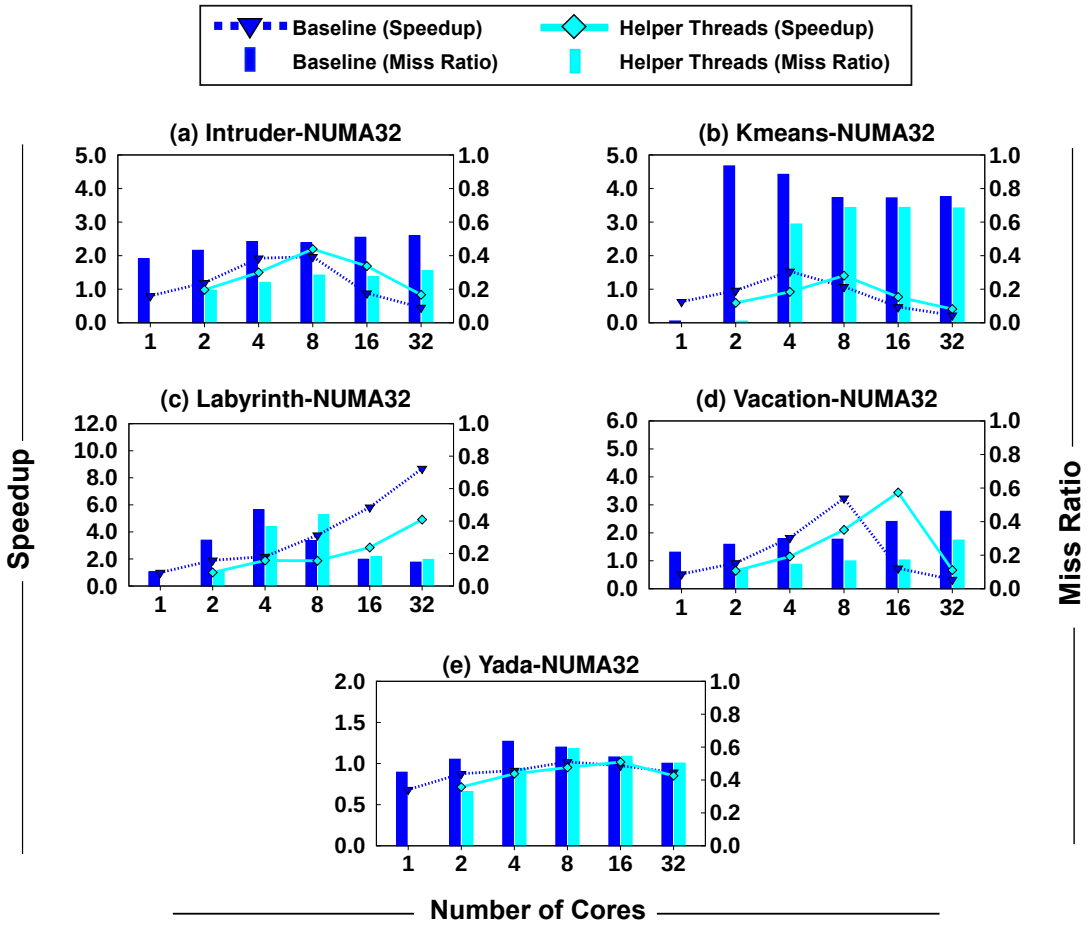

Figure 5.7: Comparison of cache miss ratio between the baseline version and helper threads optimization on the STAMP benchmark applications for the NUMA32 platform.

populated by a single thread (sequential code) before starting the next phase. On each iteration, HTs have to be re-initialized in the parallel phase. This causes extra overhead which is significant since *Kmeans* has a short execution time per parallel phase. Thus, this overhead of creating auxiliary threads surpasses the benefits obtained from them.

Helper threads led to significant cache miss ratio improvements on *Vacation* as depicted in Figure 5.7d. This stems from the fact that *Vacation* has short transactions. This allows actual and prefetched data to coexist in the shared cache without causing extra cache misses. Additionally, *Vacation* presents very low abort ratio avoiding the side-effect of wasting the prefetched data by transaction re-execution. As opposed to *Vacation*, *Yada* has long transactions. This makes it harder to attain fair timing between the worker and helper threads. In fact, it increases the probability that a helper thread

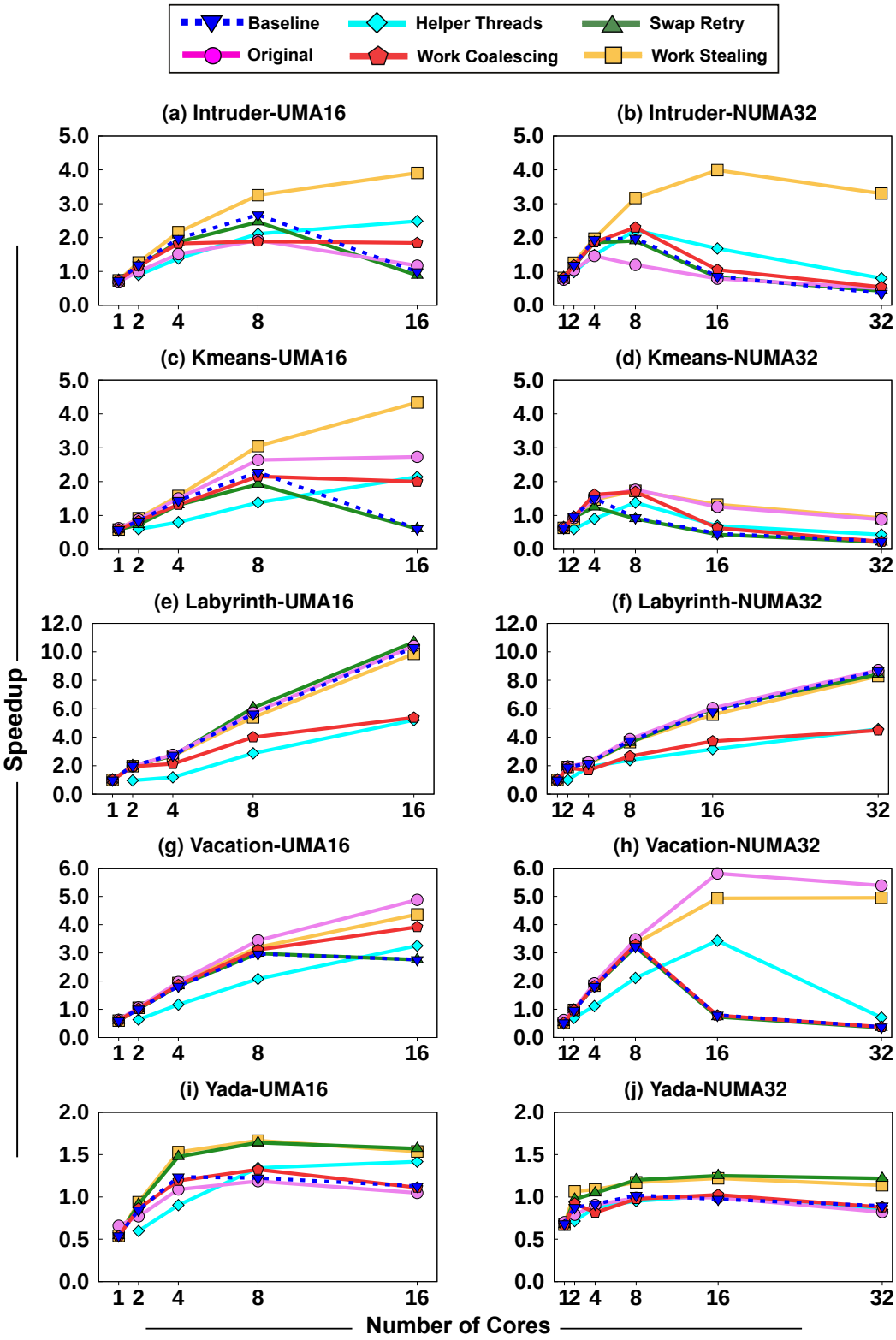

Figure 5.8: Comparison between all versions on the STAMP benchmark applications for the UMA16 and NUMA32 platforms.

takes a wrong path of execution due to premature execution. Even though, the cache miss ratio was reduced but it did not improved the performance of *Yada*.

5.2.5 Summary of Pattern-Oriented Optimizations

Figure 5.8 summarizes all the presented results. As it can be noticed, the *work stealing* optimization is responsible for many of the best performance benefits. An exception is *Labyrinth*, in which contention to the worklist is not a bottleneck. *Swap retry* has also delivered best performance improvements for *Yada* (see Figures 5.8i and 5.8j) on both platforms and for *Labyrinth* in the UMA16 platform. In particular, it reduced the abort ratio for other applications at the cost of increasing contention to the worklist. It means that once the contention is alleviated, it could also improve performance even further in other applications. Another optimization that achieved best performance improvements was *work coalescing* for *Kmeans* and *Vacation* as presented in Figures 5.8d and 5.8g respectively.

It is important to note that at least one optimization on each application achieved better performance than the original version, except for *Vacation*. Even though, the *work stealing* optimization led the OpenSkel implementation to a very close performance compared to the original version as depicted in Figures 5.8g and 5.8h. Additionally, *Labyrinth* on the NUMA32 platform is the only application where the baseline version achieved the best performance. Finally, *helper threads* improved the performance of some applications, although it was less efficient than other optimizations.

Overall, these results show that not a single optimization always delivers the best performance. In particular, it varies depending on the platforms and applications. It thus raises the question on how to automatically select them. Additionally, these optimizations can be combined in order to achieve even more performance. Furthermore, the optimization parameters such as lifespan and number of retries were fixed throughout the application executions. The tuning of these parameters could also provide additional performance. These opportunities are further exploited in the next chapter.

Chapter 6

Autotuning Performance Optimizations

Although enabling many pattern-oriented performance optimizations transparently in a single framework is an important step, the application programmer is still left with the daunting task of choosing the most profitable set of optimizations. In order to tackle this issue, this chapter first presents a novel autotuning mechanism and its implementation details. In addition to enabling the most efficient set of optimizations, it also adjusts each optimization's internal parameters. Additionally, this mechanism also tunes the number of concurrent threads automatically. Then, the autotuning mechanism is compared to individual pattern-oriented optimizations and a static oracle. Finally, the dynamic behavior of the proposed mechanism is investigated.

6.1 Autotuning Mechanism Overview

The previous chapter showed that pattern-oriented optimizations commonly tackle different performance bottlenecks. This creates an opportunity to combine more than one optimization to improve performance even further. Additionally, the pattern-oriented optimizations proposed in this thesis are orthogonal. As a result, the OpenSkel framework can enable multiple optimizations simultaneously. However, the dynamic selection of the best performing set of optimizations is still dependent on the order in which these optimizations are activated. This stems from the fact that enabling a specific optimization can influence the performance of a subsequent optimization. For instance, if the number of concurrent threads are adjusted before the worklist sharing optimization, it may lead to poor performance since an application may scale to a higher number of

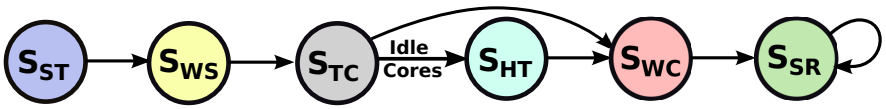

Figure 6.1: State diagram of the proposed autotuning mechanism.

cores after enabling the WS optimization. Additionally, there are some particular constraints that have to be taken into account in the selection of an autotuning strategy. First, the optimization space is large since some optimizations have also internal parameters to be tuned (i.e. swap retry). Second, based on the results presented in the previous chapter, it was observed that the transactional worklist applications are usually short (e.g. low execution time). The autotuning mechanism thus has limited time to converge to a set of tuned optimizations. Finally, as mentioned before, some optimizations have more performance impact than others, meaning that the order in which optimizations are enabled is significant.

This thesis thus proposes an autotuning mechanism that *boots* optimizations in an specific order determined by their performance impact on scalability. This one-factor-at-a-time (OFAT) method simplifies the design and improves the interpretability of the autotuning mechanism. In addition to it, this method allows the autotuning mechanism to be easily extended with new optimizations. Backed by the results presented in Section 5.2.5, it was possible to evaluate the performance impact of each optimization individually and determine a predominance order between these optimizations. Figure 6.1 shows the autotuning mechanism in the form of a state diagram that takes into account this predominance order.

The initial state is the start state (S_{ST}), which initializes the autotuning mechanism and triggers a collection of statistics such as number of aborts, stalls and commits. Then it moves onto the worklist sharing state (S_{WS}). In this state, it decides to enable or not the WS optimization.

The proposed autotuning mechanism also tunes the number of concurrent threads. The aforementioned results showed that the most profitable number of threads changes if the WS optimization is activated. It means that the enabling of the WS optimization affects the application scalability which in turn limits the maximum number of threads that can run concurrently without degrading performance. For this reason, S_{WS} precedes the thread counting state (S_{TC}). Once S_{WS} is finished, the autotuning mechanism then adjusts the number of concurrent threads in S_{TC}.

The other optimizations do not affect scalability, allowing them to be enabled in

a more flexible order. It was decided then that the next natural step was to check if there are any cores still available in order to enable and adjust HTs in the helper thread state (S_{HT}). In particular, the autotuning mechanism fine-tunes HTs by adjusting their lifespan parameter. Preliminary results showed that HTs are not sensitive to application phases. As a result, it would be too costly and inefficient to continuously tune HTs. In addition to it, the buffer size internal parameter do not also make a significant performance impact that would justify the inclusion of it in the tuning process. For these reasons, the autotuning mechanism adjusts only the lifespan of the HTs during the S_{HT} state and uses a fixed buffer size.

If there are no idle cores left, then the mechanism moves directly to the work coalescing state S_{WC}. The WC optimization is only enabled in a very specific scenario since the WS optimization have already tackled the same performance bottleneck (i.e., contention to access the worklist) for most scenarios in the S_{WS} state. Finally, the mechanism enables and keeps tuning the SR optimization in the swap retry state S_{SR}, until the program ends. It is assumed that swapping work-units is always beneficial since contention to access the worklist has been already alleviated by the preceding optimizations.

All the pattern-oriented optimizations could be continuously tuned throughout the application execution. However, based on preliminary results, it was identified that the first iterations of most STM applications are representative of their whole average behavior. This hints that optimizations could be enabled and tuned just once whereas achieving high performance during the rest of the application execution. Additionally, enabling and disabling optimizations repeatedly such as work sharing strategies and helper threads has a significant runtime overhead as mentioned before. Although the first iterations represents the average behavior of an application, it was also observed that some applications have multiple execution phases. In this case, an application could benefit from a continuous tuning mechanism. Among the available optimizations in OpenSkel, SR presents less runtime overhead and is more sensitive to the application execution phases (i.e., to the variation in the STM contention measured by the transaction abort ratio). For these reasons, in this thesis, it was decided that only the SR optimization is continuously tuned.

```
input  : State S, Number of Work-Units I,
         Number of Aborts A, Number of Stalls B,
         Number of Commits C, Lifespan L,
         Number of Threads N, Number of Cores P
output: State S, Number of Threads N,
         Lifespan L, Number of Retries R

1  begin
2      if C mod √I/P = 0 then
3          if S = S_ST then
4              S ← S_WS
5              T ← P
6              L ← 1
7          else if S = S_WS then
8              if B/C > α then
9                  optWorkStealing ← true
10                 S ← S_TC
11             end
12         else if S = S_TC then
13             autoTuningTC()
14         else if S = S_HT then
15             optHelperThreads ← true
16             autoTuningHT()
17         else if S = S_WC then
18             if A/C < β/10 and B/C > α then
19                 optWorkCoalescing ← true
20             S ← S_SR
21         else if S = S_SR then
22             optSwapRetry ← true
23             R ← 2^⌊(A/(A+C))×10
24         end
25     end
26 end
```

Figure 6.2: The main algorithm of the proposed autotuning mechanism.

6.2 Autotuning Mechanism Implementation

The main algorithm of the autotuning mechanism is shown in Figure 6.2. It implements the state diagram depicted in Figure 6.1. First, the application starts with the default baseline version with an optimistic number of threads, that is, the maximum number of available cores. This avoids the case where the application loses any available parallelism in its first iterations. However, as a side-effect, this approach increases the contention and may slowdown applications with low parallelism. Additionally, the

state variable S is initially set to the initial state S_{ST}. Then, on each iteration, the main worker thread (i.e., thread id = 0) calls the main algorithm before grabbing a new work-unit. The frequency at which the tuning process actually happens is proportional to the number of initial work-units I and cores P, as depicted on line 2 in Figure 6.2. On every $\sqrt{I}/P$ committed work-units, defined as an *epoch*, the autotuning mechanism re-evaluates its current state. This usually results in a change of state and/or the enabling of an optimization. The latter is represented by the assignment of *true* to the corresponding optimization variable. For instance, line 9 shows when the WS optimization is activated. In particular, some states take several epochs to switch to the next state. This is the case for the S_{TC} and S_{HT} states in which the *autoTuningTC()* and *autoTuningHT()* functions, on lines 13 and 16 in Figure 6.2, implement respectively the hill-climbing strategies to tune the number of threads and the lifespan of helper threads. These functions are presented in detail in Figures 6.3 and 6.4. Finally, the α and β thresholds, on lines 8 and 18 in Figure 6.2, are determined at design time by a sensitivity analysis that is discussed in Section 6.3.3. The rest of this section describes the implementation details to tune each of the pattern-oriented optimizations.

6.2.1 Enabling Work Stealing

After the initial S_{ST} state, the autotuning mechanism moves to the S_{WS} state. In this state, it evaluates if there is high contention to access the worklist as a condition to enable the WS optimization. In order to do this, it checks if the ratio between the number of *stalls* to access the worklist and the number of *committed* work-units is above a threshold α as shown on line 7 in Figure 6.2. Each time a thread access the worklist and has to wait in a lock, this is counted as a stall. Low stall ratio means that the worklist is not under contention. In this scenario, the work sharing should be maintained since it provides optimal load balancing.

6.2.2 Autotuning the Concurrency Level

16. p65, g 6.3. Mention that R values are limited to +1,0,-1. Somewhere (in caption or in code) you need to mention that this routine is named autoTuningTC().

The next step is to adjust the number of threads in the S_{TC} state. The proposed mechanism uses a hill climbing heuristic implemented in the *autoTuningTC()* function as depicted in Figure 6.3. It is based on the ratio between work-units *aborts* and *commits*. If this ratio is below a threshold β it means that the actual number of threads

```
input  : State S, Counter R, Number of Aborts A,
         Number of Commits C, Number of Threads N,
         Number of Cores P
output: State S, Counter R, Number of Threads N

1  begin
2      if A/C < β then
3          if R = 0 then
4              R ← R + 1
5          else if R = 1 then
6              if N < P then
7                  S ← S_HT
8              else
9                  S ← S_WC
10             end
11         else
12             R ← 0
13         end
14     else
15         if R = 0 then
16             R ← R − 1
17         else if R = −1 then
18             N ← max(N/2, 1)
19             R ← 0
20         else
21             R ← 0
22         end
23     end
24 end
```

Figure 6.3: The algorithm to autotune the thread concurrency level in the autoTuningTC() function.

exploits parallelism efficiently. However, this has to be confirmed in one more epoch through a *counter R* before fixing the number of threads. In particular, this counter R can assume only three values +1,0 and -1. This double checking avoids making a wrong decision based on a biased interval. The same is valid when the ratio is above β. The algorithm waits for a consecutive confirmation before halving the number of threads. This process ends when consecutive epochs present a ratio below β. In Section 6.3.3, a sensitivity analysis is performed in order to choose and understand the performance impact of these thresholds. In particular, it shows that there is small performance variation on most of the investigated space. It means that even when the mechanism uses non-optimal threshold values, it still makes the right choices.

```
input  : State S, Counter R, Lifespan L,
         Number of Commits C, Epoch E,
         Max Lifespan M
output: State S, Counter R, Lifespan L

1  begin
2      if C_i/E_i < C_{i-1}/E_{i-1} then
3          if R = 0 then
4              R ← R − 1
5          else if R = −1 then
6              S ← S_WC
7              if L = 1 then
8                  L ← 0
9              end
10         else
11             R ← 0
12         end
13     else
14         if R = 0 then
15             R ← R + 1
16         else if R = 1 then
17             L ← min(L × 10, M)
18             R ← 0
19         else
20             R ← 0
21         end
22     end
23 end
```

Figure 6.4: The algorithm to autotune the lifespan of helper threads in the autoTuningHT() function.

6.2.3 Autotuning Helper Threads

Next, the algorithm switches to the S_{HT} state if there are idle cores or goes straight to the S_{WC} state. The autotuning strategy to determine the lifespan of *helper threads* follows similar approach to the S_{TC} state. As shown in Figure 6.4, it also uses a hill climbing strategy but in the opposite direction. In contrast, it starts with a pessimistic lifespan equal to one and move towards a maximum lifespan. Basically, if the current throughput with HTs enabled is higher than without them, it multiplies the lifespan by a factor of 10. Once the lifespan is determined, the algorithm switches to the S_{WC} state.

6.2.4 Autotuning Work Coalescing

The S_{WC} state is implemented on line 17 in Figure 6.2. Since the WC optimization also tackles the contention problem in the worklist, it is only enabled if the number of stalls is high and the ratio between abort and commits is very low, a tenth of β. Then, it switches to the S_{SR} state.

6.2.5 Continuous Tuning of Swap Retry

Finally, in the last state, the SR optimization is enabled and adjusted continuously. It uses an exponential function based on the abort ratio to adjust the number of retries before a swap as described on line 23 in Figure 6.2. The intuition behind it is that as the abort ratio increases, work-unit swaps become expensive and inefficient. This stems from the fact that under high abort ratio, when a work-unit is swapped it will end up aborting anyway and the thread may also lose the natural prefetching of the previous execution. On the other hand, it is also assumed that aggressive SR (i.e., swap on every abort) is not beneficial since in the next execution the work-unit can execute and commit faster using the prefetched data. Thus, the number of retries is limited to a minimum of one retry before swapping.

6.3 Analysis of the Autotuning Mechanism

After analyzing the impact of each optimization individually in Chapter 5, this section compares the proposed autotuning mechanism with the best single optimization and the *static oracle*. In particular, the static oracle is obtained through exhaustive investigation of the search space of combined optimizations. Since it is static, that is, optimizations and their parameters are fixed throughout the application execution, so it is possible for the dynamic autotuning mechanism to outperform it.

Figure 6.5 shows the performance improvement, in percentage, over the best *baseline* execution. By best, it means the fastest execution for a specific application, from all possible number of cores (i.e., in some cases more cores result to slowdowns). Additionally, the set of optimizations (e.g., WS+SR+HT) used to achieve the performance improvement is also highlighted in Figure 6.5. In order to analyze the following results, the performance benefits of combining optimizations (i.e., static oracle) against individual optimizations are first discussed. Then, the static oracle is compared with the autotuning mechanism.

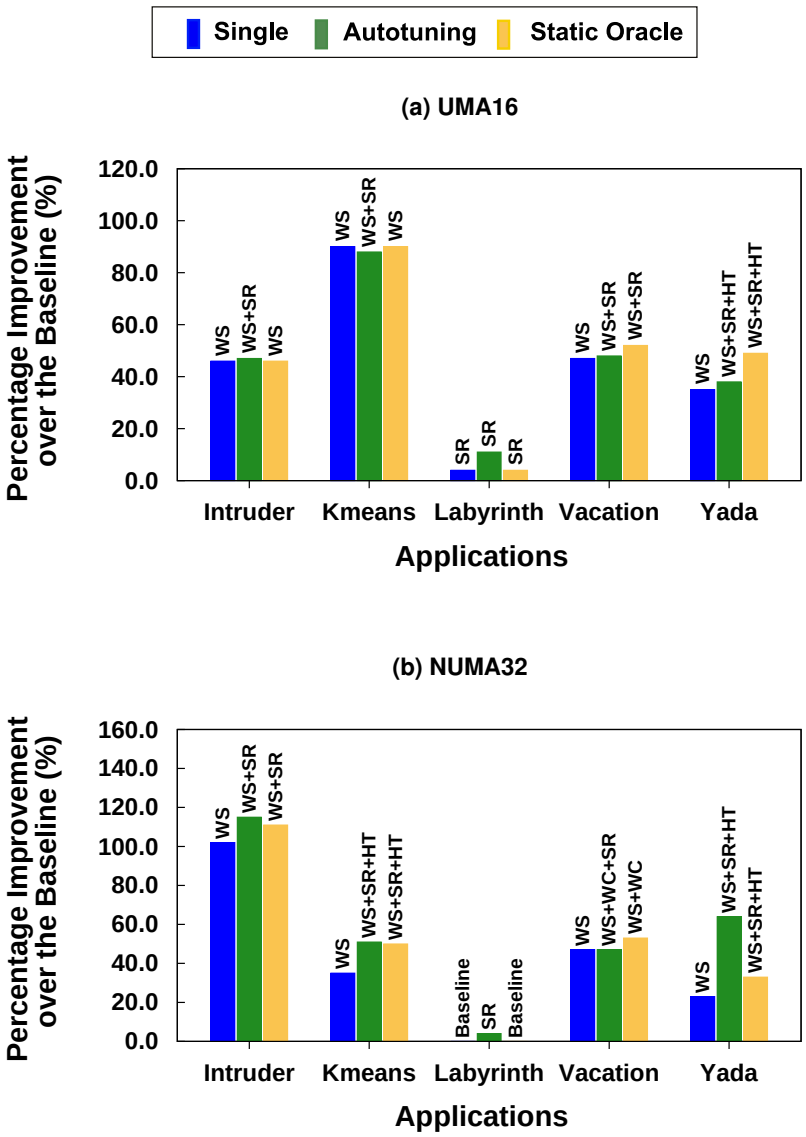

Figure 6.5: Performance improvement of the autotuning mechanism and the best combination of optimizations over the best baseline execution.

6.3.1 Single vs. Combined Optimizations

Figure 6.5 shows that the static oracle, that combines optimizations, delivers performance improvements up to 15%, when compared to individual optimizations for most of the applications. As expected, the WS optimization is beneficial for all applications except *Labyrinth*. *Labyrinth* reaches its best performance with the baseline version for the NUMA32 platform and with SR for the UMA16 platform.

As mentioned earlier, under high abort ratio, SR increases the contention to access

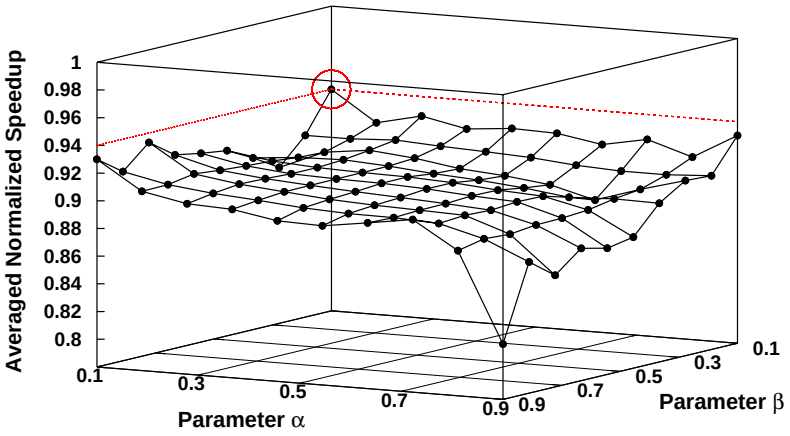

Figure 6.6: Sensitivity analysis of parameters α and β.

the centralized worklist. However, since the WS optimization distributes the worklist, this undesired behavior is largely negated when the two optimizations are combined. Based on the fact that SR only swaps work-units within its local worklist, it was able to deliver performance improvements for many applications.

The WS and WC optimizations tackle the same performance bottleneck, namely contention to access the worklist. Since the WS benefits outweigh the ones of WC, combining them unnecessarily increases the abort ratio in most cases. An exception to this is *Vacation* that has very small transactions and very low abort ratio. Enabling WC on top of WS, reduces contention even further in the NUMA32 platform as shown in Figure 6.5b. Finally, combining HTs with WS and SR improved performance even a slightly further in *Yada* and *Kmeans*, showing that idle cores, when available, can be utilized for prefetching (i.e., most applications scale to the maximum number of cores after optimizations). Since SR reduces the number of re-executions, HTs become more profitable by prefetching data that will be used soon instead of being discarded, polluting the cache.

6.3.2 Autotuning vs. Static Oracle

In order to better understand the behavior of the proposed autotuning mechanism, a sensitivity analysis for its internal parameters was performed. As aforementioned, the

α threshold is related to the ratio between stalls and commits and it is used in the WS step. The β threshold defines the desired ratio between aborts and commits and is used to choose the number of concurrent threads and also in the S_{WC} state.

Figure 6.6 shows the averaged normalized speedup for the applications running in the UMA16 platform. The best tuple is (0.1,0.1) as highlighted in Figure 6.6. This means that only under very low contention, the worklist should be not be distributed using the WS optimization. Additionally, the number of threads has to be reduced until a low ratio between aborts and commits is reached, so as to achieve the best performance. In contrast, the worst results are achieved with the (0.9,0.9) tuple. In this case, WS is never used and the number of threads is always the maximum number of cores, which is not profitable for some applications. However, it can also be observed that there is small performance variation across most of the investigated space. It shows that even when non-optimal values are used, the autotuning mechanism still delivers good average performance.

Based on this study, we chose the tuple (0.1,0.1) to drive the proposed mechanism in all results shown in Figure 6.5. The same tuple was also used for the NUMA32 machine, showing that it can be portable across machines.

Figure 6.5 shows that the autotuning mechanism can deliver similar performance, or in some cases be even more efficient, when compared to the best static combination of optimizations. The proposed mechanism was only 2% slower than the best combination on average for the UMA16 machine and 7% faster for the NUMA32 platform. It is also important to point out that the proposed mechanism was always equal to or faster than the single best optimization case. The continuous tuning of *swap retry* proved to have a significant impact for the NUMA32 platform, improving the performance of *Yada* by 31% over the static oracle.

6.3.3 Analysis of the Autotuning Dynamic Behavior

In Figure 6.7, the behavior of the autotuning component is analyzed further for all applications, running on the UMA16 platform. Each graph shows, when each of the states are activated, how many epochs it takes to finish and the values for each of the optimization internal parameters. For instance, Figure 6.7c in the S_{SR} state shows that for *Labyrinth* the number of retries is reduced from 32 to 4 across epochs. Note that it took less than 2% of Yada's total execution time to autotune the application as shown in Figure 6.7e. This trend is consistent across all applications, except *Labyrinth* which

has few long transactions. Despite this fact, it takes fewer epochs to converge since it is profitable to make use of all cores.

Another interesting result derived from these graphs is that they show how the SR optimization exploits the variation of the abort ratio throughout the application execution. For instance, the number of aborts increases radically by the end of *Intruder*'s execution. Since the number of retries is proportional to the abort ratio, this increase can be observed in the S_{SR} state in Figure 6.7a. SR thus almost stops swapping work-units since it will be probably worthless as aborts become inevitable.

In 6.7e, it is also important to observe that HTs are enabled and the lifespan is increased until 1000 which is not the maximum lifespan. This shows that the autotuning mechanism was able to detect the most profitable lifespan for *Yada* in the UMA16, which in this case is 1000 words per work-unit.

Overall, the proposed autotuning mechanism converges to a set of optimizations that matches the ones of the static oracle. Additionally, it surpasses the static oracle performance benefits when the dynamic tuning of SR is profitable. Finally, it can achieve performance improvements of up to 88%, with an average of 46%, over a baseline version for the UMA16 platform and up to 115%, with an average of 56% for the NUMA32 platform.

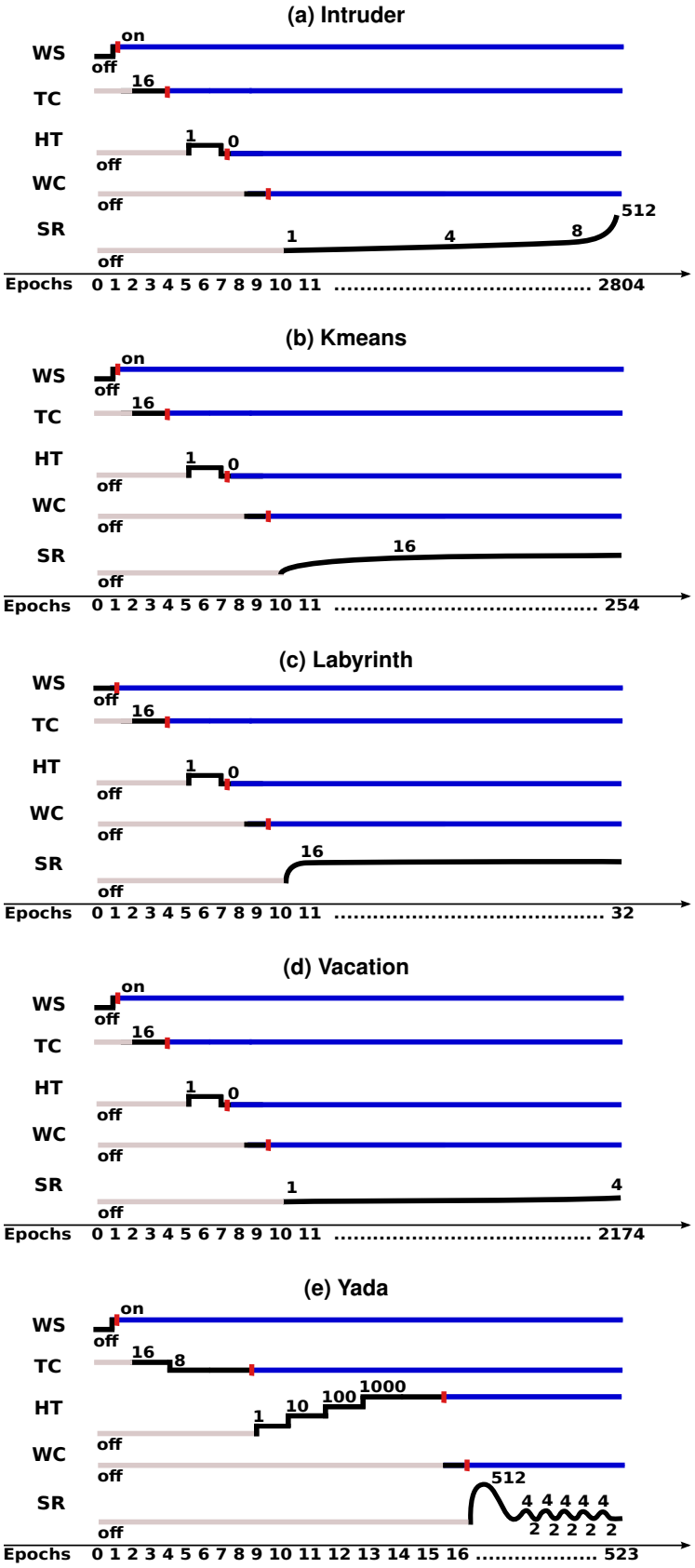

Figure 6.7: Autotuning dynamic behavior for all applications in the UMA16 platform.

Chapter 7

System-Oriented Performance Optimizations

In addition to enabling pattern-oriented optimizations, a skeleton framework can also employ *system-oriented* performance optimizations. These optimizations are usually used to enhance memory affinity, which keeps data close to the cores which access it [6, 11]. In particular, these system functionalities such as thread mapping and memory page allocation are provided by the operating system. However, the operating system commonly applies a fixed set of generic heuristics that delivers good performance to applications for the average case. This means that these heuristics do not take into account specific behavior characteristics shared by some classes of applications such as TM worklist applications.

This chapter presents how to enable and automatically tune system-oriented performance optimizations such as thread mapping and memory page allocation within OpenSkel. Additionally, the autotuning mechanism is extended to support both pattern-oriented and system-oriented optimizations. Then, it is compared to a static oracle that uses both categories of optimizations.

7.1 Employing System-Oriented Optimizations

7.1.1 Memory Accesses in TM Worklist Applications

On transactional worklist applications, data dependencies cannot be determined at compile time. In principle, no specific memory access pattern can be assumed since threads may potentially access memory addresses in an uniformly random fashion.

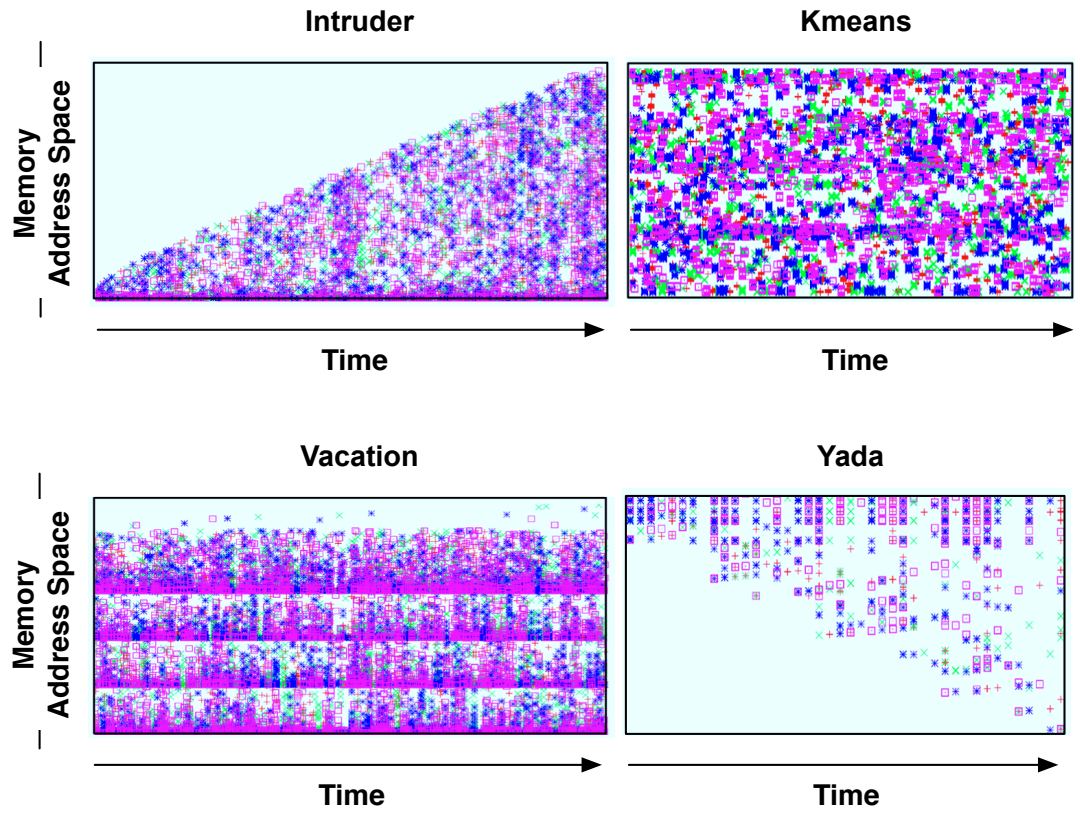

Figure 7.1: Sampled memory footprint of pages accesses for transactional worklist applications executing with four threads. Each data point represents a thread accessing a certain memory page at a specific time. Each symbol represents a different thread.

Figure 7.1 supports this observation by showing the memory page accesses footprint of four STAMP applications during their execution. In particular, this shows that each thread ends up traversing most of the memory page space, frequently conflicting with other threads. This fact makes harder to enhance memory affinity by applying pure static approaches. At the same time, dynamic approaches usually require frequent data or thread migrations that can be prohibitively expensive.

To tackle this problem, this chapter proposes an extension to the autotuning mechanism to enable and select system-oriented optimizations in order to enhance memory affinity for transactional worklist applications. Although the skeleton framework leaves to the STM system the handling of data accesses and dependencies, it can improve memory affinity applying these system-oriented optimizations based on the TM nature of the applications and the platform features. The solution proposed thus exploits static and dynamic information to provide affinity by thread mapping and memory page allocation. In the thread level, thread mapping strategies are implemented in

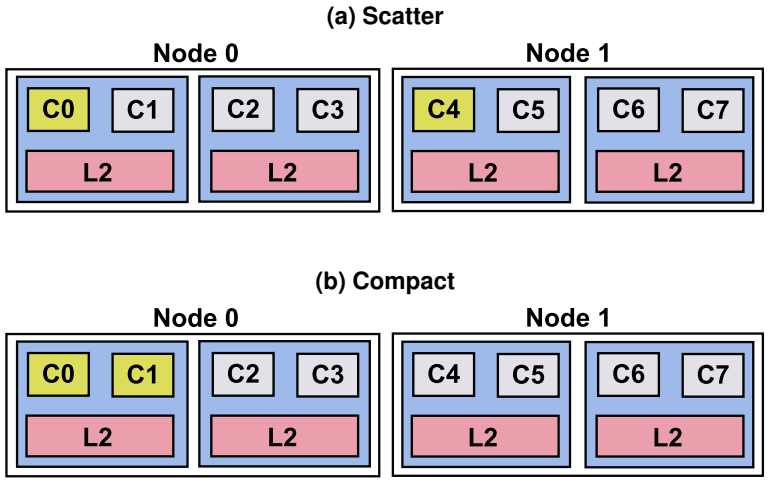

Figure 7.2: Thread mapping strategies enabled by the OpenSkel framework. The highlighted cores represent where threads are placed.

OpenSkel. These strategies place threads on specific cores in order to enhance memory affinity. This allows the autotuning mechanism to select the most efficient thread mapping strategy depending on the memory hierarchy of the platform. In the memory level, the autotuning mechanism takes into account this irregular memory access pattern to apply a more suitable page allocation policy. Such policy is used to specify how memory pages are distributed over the physical memory banks of a platform. Additionally, based in the skeleton information, the autotuning mechanism switches to a different policy if profitable. This approach do not employ page migration since it would be to costly as threads access different memory pages in a short period of time.

7.1.2 Mapping Threads to Cores

A well-known system-oriented functionality is thread mapping [41, 44]. It places threads on specific cores in order to reduce memory latency or alleviate memory contention. For instance, threads that communicate often could be placed on cores that share some level of cache memory to avoid high latency accesses to the main memory. Thread mapping strategies can use prior knowledge of the application behavior [27] and/or of the platform memory topology [23] to map threads to cores. Although several mapping strategies have been proposed [23, 27], there is no single solution that delivers high performance across different applications and platforms. In particular, transactional worklist applications present even more complex behavior due to the speculative nature of the transactional system. Thus, the programmer needs alternative

thread mapping strategies for each application and platform.

As aforementioned, the OpenSkel baseline version employs the *scatter* thread mapping strategy. It distributes threads across different processors avoiding cache sharing between cores in order to reduce memory contention as shown in Figure 7.2a. Another alternative thread mapping strategy is called *compact* (CP). This is one of the strategies employed on Intel OpenMP system [23]. In this optimization, threads are placed in cores in order to maximize memory sharing. In contrast to the *scatter* strategy, Figure 7.2b shows that *compact* forces threads to run on cores located on the same socket or node with shared caches. It reduces memory access latency for communicating threads. However, as a side-effect it also increases the contention of cache accesses, degrading the performance of memory intensive applications. Finally, the Linux default scheduling strategy is a dynamic priority-based one that allows threads to migrate to other cores considering their behavior.

7.1.3 Allocating Physical Memory

Memory allocation policies reduce latency costs or increase bandwidth for memory accesses by enhancing memory affinity [66]. This is specially appealing for NUMA platforms where latencies and bandwidth are variable between nodes. These memory allocation policies have been predominantly studied in the context of regular parallel applications on NUMA multi-core platforms, in which the memory access pattern is stable and predictable [6, 11, 66]. Irregular applications such as transactional worklists differ from regular ones as their memory access pattern is more complex, stemming from the fact that data dependencies between threads are only know at runtime as discussed in Section 7.1.1. An approach to improve memory affinity for irregular applications is to provide a set of customized data structures to the application programmer, so that the runtime system can be aware of where the data is allocated in the memory [51]. The runtime system is then able to partition the data between threads in such a way that each thread works on a different subset of the data whenever possible. This approach improves memory affinity at the cost of forcing the application programmer to use a set of pre-defined data structures and a very specialized runtime system. Another more general approach is to allocate memory based on the architecture features rather than the application data structures. This approach does not require any source code modifications, freeing the application programmer to use its own data structures. Furthermore, both regular and irregular applications are similarly affected

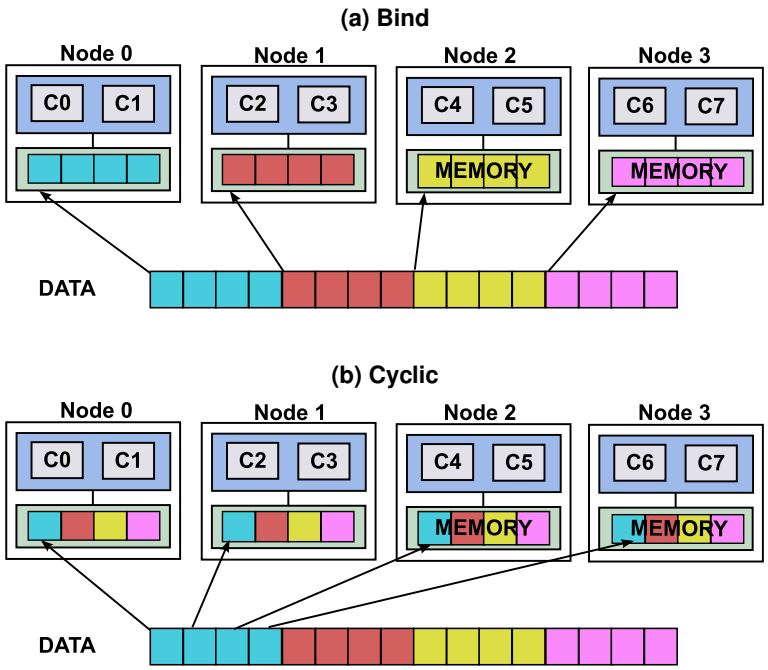

Figure 7.3: Bind and cyclic memory allocation policies enabled by the OpenSkel framework.

by the affinity dilemma: to reduce latency or to alleviate contention. In fact, this trade-off is intrinsic to NUMA platforms. For that reason, the same mechanisms employed to enhance affinity in regular applications should also be profitable to irregular ones.

In order to improve DRAM memory affinity in the OpenSkel framework, it was extended to provide NUMA-aware memory policies. The OpenSkel baseline does not provide any memory allocation support, leaving memory management to the native operating system. Furthermore, it expects that an application programmer provides accurate information regarding the platform topology. To overcome this limitation, that process was automated by the use of a tool to capture machine topology information similar to *hwloc* [12]. This information is essential to implement NUMA-aware memory policies.

A memory allocation policy is used to specify how memory pages are distributed over the physical memory banks of a machine. For instance, the Linux operating system uses the *first-touch* policy which places data on the node of the thread that first touches it [6]. To support memory affinity in transactional worklist applications, two existing memory policies named *bind* and *cyclic* were employed. Both memory policies are implemented within the OpenSkel runtime, avoiding application source code modifications.

The *bind* memory policy aims at reducing access latency by gluing a thread to a single memory bank. Whenever a thread allocates or requests a memory page, its corresponding virtual page is placed on a physical memory bank based on information about the NUMA platform topology.

Figure 7.3a depicts the *bind* policy in a NUMA platform with n nodes. In this example, the application data is allocated by a team of n threads. Then it is split in n parts, each one being placed on a different memory bank. A side-effect of binding thread to memory banks is that it may cause more memory contention when different threads share the same memory range. In order to avoid such behavior, the *cyclic* memory policy spreads memory pages over the memory banks following a round-robin distribution. The granularity unit used in the *cyclic* memory policy is a memory page. Therefore, a page i is placed in the memory bank $i \ mod \ m$, where m is the number of memory banks of the platform. The *cyclic* policy aims to balance memory bank usage, because it allows memory banks to be accessed in parallel, providing more bandwidth to cores.

Figure 7.3b shows a schema that represents the *cyclic* memory policy in a NUMA platform. Thus, the first memory page of the application data is physically stored on memory banks sequentially from 0 to 3 in a round-robin fashion.

In order to implement these policies in OpenSkel, the *libnuma* [48] library was integrated to the framework. It provides an API to set specific page allocation policies such as *bind* and *cyclic* which are encapsulated by the OpenSkel framework. As a consequence, page allocation policies are not exposed to the application programmer. In particular, it can fully automate the process of selecting page allocation policies.

7.1.4 Autotuning System-Oriented Optimizations

After extending OpenSkel to support both thread mapping strategies and memory page allocation policies, the autotuning mechanism can be also extended to handle the these optimizations. It is extended with a simple but efficient heuristic to automatically enable and select the system-oriented optimizations. This heuristic exploits the application algorithmic pattern and system information to choose which strategies should be applied.

As aforementioned in Section 7.1.1, transactional worklist applications exhibit very irregular memory access footprint tending to an uniform distribution. Based on this observation, the autotuning mechanism always set *cyclic* as the default page allocation

policy in the OpenSkel initialization (i.e., within the *oskel_wl_alloc()* function) as it distributes memory pages equally across nodes. This can increase the average performance of the applications since each thread will potentially access the same amount of data on each node. However, if the skeleton informs that threads are generating new work-units (i.e., use of the *oskel_wl_addWorkUnit()* function), the mechanism optimistically assumes that these work-units are memory related. Then it switches the page allocation policy to *bind*.

Backed by preliminary results, it was observed that selecting the most appropriate thread mapping strategy for a transactional application is more complex than expected. This stems from the fact that each combination of STM system/application/platform modifies the behavior of the application, so the best thread mapping strategy is also affected. In order to efficiently solve this problem, more sophisticated solutions are required. This led to the proposal of a machine learning-based (ML) approach to automatically infer a suitable thread mapping strategy for transactional memory applications, not constrained by TM worklist applications. First, it builds a set of input instances. Then, such data feeds a machine learning algorithm called ID3, which produces a decision tree able to predict the most suitable thread mapping strategy for new unobserved instances. The results obtained by applying this ML approach are presented in [17].

This thesis focuses only on transactional worklist applications and simple dynamic heuristics to autotune a broader range of performance optimizations. Then, instead of employing a more complex approach, the proposed autotuning mechanism in this thesis relies on a simple but efficient heuristic. First, it employs the *compact* strategy when the platform is NUMA. This stems from the fact that on NUMA platforms communication between nodes is expensive, so placing threads as close as possible can reduce the amount of remote data requests. And since transactional worklist applications communicate quite often, this is potentially an efficient solution. Then, the *scatter* strategy is employed when the application is running on an UMA platform. This can be explained by the fact that in the absence of different memory access latencies, cache memories become the main source of contention. The *scatter* strategy maps threads in such a way that each thread has usually exclusive access to a shared cache that would be otherwise used also by another thread at the same time. This can improve each thread individual performance potentially boosting the whole application. Finally, this automatic decision in made within the the *oskel_wl_alloc()* function.

System-oriented optimizations are orthogonal to pattern-oriented ones and can be

naturally applied simultaneously. However, the enabling of system-oriented optimizations affects performance and consequently the decision on which pattern-based optimizations should be activated. Nevertheless, system-oriented optimizations commonly cause higher performance impacts since they happen in a lower level. For this reason, they should be enabled before the pattern-based optimizations.

7.2 Analysis of the System-Oriented Optimizations

In this section, the performance of the enabled skeleton-driven system-oriented optimizations are analyzed. The memory allocation policies are executed on the NUMA16 platform instead of the UMA16, since they do not affect the performance of UMA platforms. In the rest of this section, the performance benefits of the thread mapping strategies are first analyzed. Second, the memory allocation policies are also investigated. Then, the results of combining both system-oriented and pattern-oriented ones are discussed. Finally, the new extended autotuning mechanism is compared to a static oracle.

7.2.1 Evaluating Thread Mapping

Overall, the Linux mapping strategy presents similar performance to the *scatter* strategy according to Figure 7.4. This is due to the fact that Linux also tries to split threads in different nodes/sockets rather than cluster them. In contrast, the *compact* mapping strategy tries to reduce communication latency between threads by placing them in cores that share higher levels of memory. As shown in Figure 7.4, *compact* improved application performance by up to 134%, and 48% on average. Basically, in the NUMA32 platform, it improved the best performance of all applications but *Labyrinth*. Since it scales up to 32 cores, then thread mapping does not affect its best performance. All other applications communicate frequently, so reducing this communication latency is an important issue. In particular, the NUMA32 platform has a L3 cache shared by each group of 8 cores. *Compact* thus places threads so they use this L3 cache as a fast communication channel, reducing memory access latency.

As opposed to the NUMA32 platform, in the UMA platform, only *Yada* and *Kmeans* benefit from *compact*. Both execute medium-sized transactions that demand an amount of data that fits within the L2 shared caches. Once threads competing for the same L2 cache do not degrade memory latency, they also benefit from fast communication.

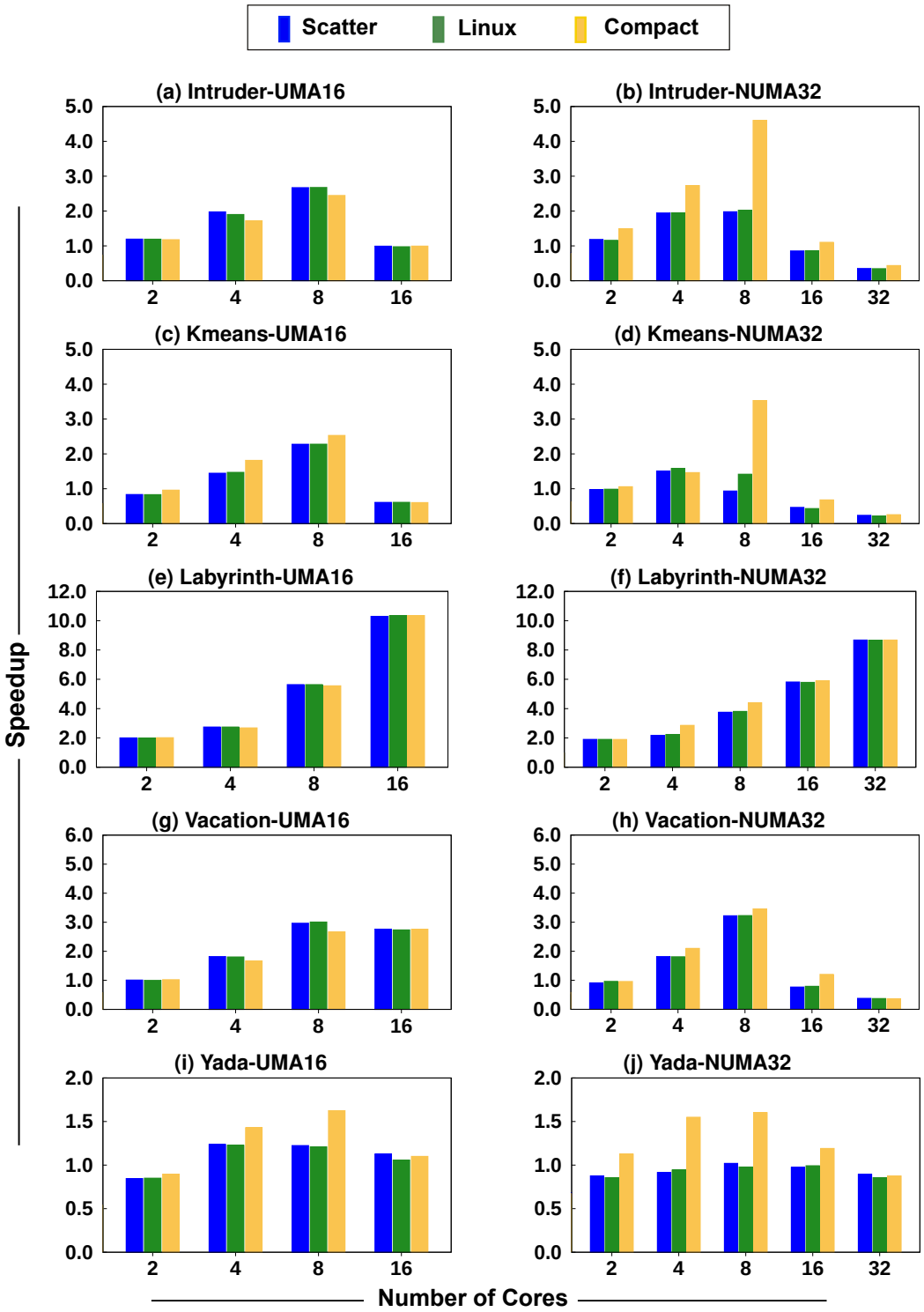

Figure 7.4: Comparison between different thread mapping strategies for the baseline version on the STAMP benchmark applications for the UMA16 and NUMA32 platforms.

Then *Yada* and *Kmeans* observed an increase in performance over the baseline version of up to 31% and 11% respectively.

For *Vacation* and *Intruder*, *scatter* is a better strategy than *compact* since they have short memory intensive transactions. *Labyrinth* is also a memory intensive application. However, as it scales up to the maximum number of cores, this thread mapping strategy does not affect its best performance. Finally, the abort ratio is not shown due to the fact that these thread mapping strategies do not affect it. This is expected since they are system optimizations that do not tackle TM conflicts.

7.2.2 Evaluating Physical Memory Affinity

As aforementioned, the *bind* and *cyclic* memory allocation policies were implemented in OpenSkel. Figure 7.5 reports the speedup of the selected benchmarks on both NUMA16 and the NUMA32 platforms. It shows the performance of the baseline version which uses the default Linux memory allocation policy compared to the implemented ones. The Linux operating system uses the *first-touch* policy which places data on the node of the thread that first touches it [6]. Overall, *bind* and *cyclic* policies perform 8% better than the baseline, on average. In particular, these policies improved the performance of *Kmeans* and *Vacation* up to 14% and 46% respectively on the NUMA16 platform as depicted in Figures 7.5c and 7.5g.

In the NUMA16 platform, NUMA-aware memory polices presented significant performance gains as opposed to the NUMA32 platform. Since the NUMA16 platform does not have shared cache memories and nodes are not fully connected, the impact of the memory policies is more significant. In the NUMA32, remote data requests rarely lead to an access to a remote memory bank. This stems from the fact that each node has a large shared L3 cache and they are interconnected through high speed communication channels. Instead of accessing a remote memory bank directly on a data request, a core checks if the data is available on its local and remote caches. Due to this, the impact of physical allocation memory policies in the NUMA32 platform is smaller.

Intruder, *Kmeans* and *Vacation* benchmarks which experience more memory contention have presented better performance with the *cyclic* memory allocations policy in the NUMA16 platform. This policy makes more bandwidth available per core due to its distributed nature of placing memory pages. Therefore, lower latencies to access the worklist are expected with this memory policy. In particular, the *cyclic* policy de-

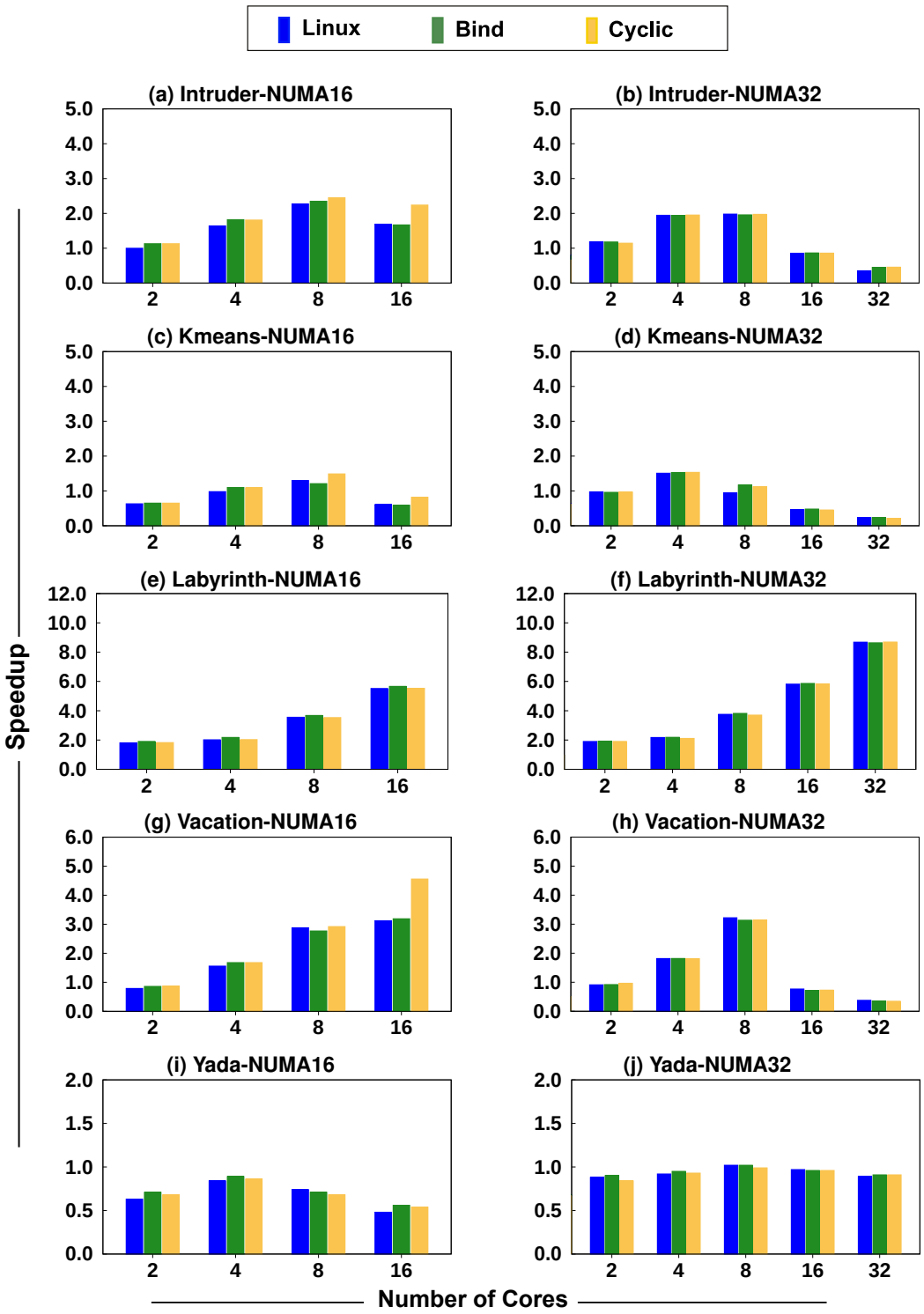

Figure 7.5: Comparison between different memory allocation policies for the baseline version on the STAMP benchmark applications for the NUMA16 and NUMA32 platforms.

livers a significant performance improvement to *Vacation* as presented in Figures 7.5c. This application has very small transactions increasing contention to access the worklist. The *cyclic* policy thus alleviates that contention since it distributes the pages that compose the worklist among the memory banks.

In contrast, Figures 7.5i shows that *Yada* benefits more from the *bind* policy since it has long transactions and much less contention to access the worklist. Due to this, the most suited policy is to allocate memory pages closer to the thread that access them. Additionally, in *Yada*, each thread inserts new work-units in the worklist during the benchmark execution. For that reason, threads have already data close to them, reducing the need of accessing remote nodes.

7.2.3 Evaluating System and Pattern-Oriented Optimizations

This last section investigates the performance improvement of combining system and pattern-oriented optimizations in the UMA16 and NUMA32 platforms. Moreover, it shows that even when these system-oriented optimizations provides a significant speedup, the pattern-oriented optimizations can add up further performance. Basically, the investigated memory allocation policies do not apply to the UMA16 platform. Furthermore, they do not have a significant impact on the NUMA32 as explained in the previous section. As a result, these optimizations did not influenced in the best execution or static oracle of each application as shown in Figure 7.6.

In contrast, the *compact* thread mapping strategy improved performance even further for many applications. Particularly, in the NUMA32 platform, *compact* became the best single optimization for *Intruder*, *Kmeans* and *Yada* as depicted in Figure 7.6b. The exceptions are *Labyrinth* and *Vacation* that scales up to the maximum number of cores, when thread mapping becomes irrelevant. Figure 7.6a shows that it is also the case for most applications in the UMA16 platform, except from *Yada* that scales only up to 8 cores. In particular, the combined performance of *compact* and other optimizations led to a performance improvement of 15% compared to the previous static oracle based only on pattern-oriented optimizations.

For the UMA16 platform, the extended autotuning mechanism did not improved compared to the previous one. This stems from the fact that for UMA platforms the *scatter* thread mapping policy is always used and only *Yada* benefits from employing the *compact* strategy. In contrast, the *compact* strategy at least as efficient as the *scatter* strategy in the NUMA32 platform and is always employed by the autotuning. This

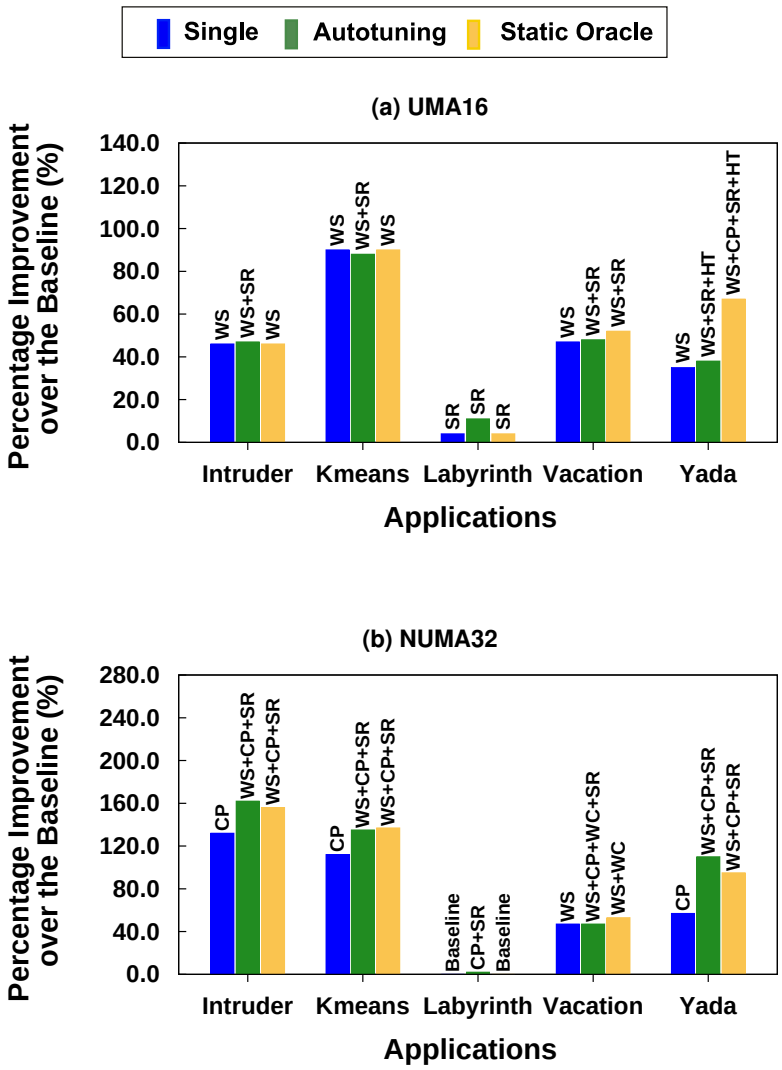

Figure 7.6: Performance improvement of the extended autotuning mechanism, the static oracle including system-oriented optimizations and the single best optimization over the best baseline execution.

enabled the autotuning mechanism to achieve performance improvements that surpass the static oracle by 6% and 15% for *Intruder* and *Yada* respectively.

In summary, the experimental results show that the extended autotuning version combining system-oriented and pattern-oriented optimizations can achieve performance improvements of up to 88%, with an average of 46%, over the baseline version for the UMA16 platform and up to 162%, with an average of 91%, for the NUMA32 platform.

Chapter 8

Related Work

8.1 High Level Parallel Programming

8.1.1 Parallel Languages and Extensions

One approach to support parallelism is to extend existing programming languages with keywords from the parallel programming domain to spawn and synchronize threads and partition data. This alternative is well exploited in languages such as Cilk++ [8] and Charm++ [46]. Particularly, Cilk++ [8] has extensively employed *work stealing* in order to overcome scalability bottlenecks. In [62], several optimizations were applied transparently to the Charm runtime system to support clusters of multi-cores. However, both Charm++ and Cilk++ are designed for non-transactional parallel applications.

Another alternative is to support parallelism in functional programming languages. For instance, Haskell *map* function can express data-parallelism in an elegant manner. In [38], Concurrent Haskell is extended to support transactional memory. It takes advantage of the Haskell type system to provide stronger guarantees. Additionally, transactions can be composed to form larger transactions. Such language solutions aim to improve programmability and enable compiler-driven optimizations. As with sequential languages, their main goal is to be generic. Also, unlike skeleton libraries, they rely on a compiler for analysis and optimizations.

8.1.2 Library-Based Frameworks

In contrast to parallel language extensions, this thesis's proposed approach is based on skeletal programming for software transactional memory applications. An extensive survey of recent parallel skeleton languages and libraries is presented in [32]. Although

many parallel skeletons have been proposed, they are efficient only for regular data and task-parallel applications.

In [49, 51, 52], the authors have identified that new programming abstractions are required to efficiently use speculative execution on a particular class of irregular applications that exhibits amorphous data-parallelism. These applications are mostly worklist-based algorithms that work within a shared graph data structure. They have implemented the Galois system, a Java-based interface that provides a set of iterators to implement the worklist and a runtime system that implements the speculative execution support. Galois follows a data-centric approach where users must plan their code around especially designed recursive data structures, while OpenSkel focuses on an algorithmic abstraction based on computation patterns in which there are no restrictions on the data structures used. Additionally, the proposed framework in this thesis and its optimizations are decoupled from the underlying speculation mechanism. This means that it runs on top of state-of-the-art STM systems, extending the STM paradigm and inheriting the performance achieved by existing STMs.

8.2 Skeleton-Driven Performance Optimizations

8.2.1 Pattern-Oriented Performance Optimizations

Recent work [63] has exploited structure-driven optimizations for irregular applications. In fact, they proposed a technique called *iteration coalescing* that was adapted to *work coalescing* (WC) in the proposed skeleton framework. Both techniques coalesce smaller transactions into larger ones in order to reduce overheads related to the dynamic assignment of work to threads. Differently from [63], WC does not support partial commit since it requires modification in the target speculation system.

The work in [4] proposes a *"steal-on-abort"* mechanism. When concurrent transactions conflict, the aborted transaction is stolen by its opponent transaction and queued to avoid simultaneous execution. Multiple local queues with job stealing for load balancing are also employed. In contrast, rather than stealing a conflicting transaction from another thread, one of the proposed optimizations in this thesis, called *swap retry* (SR), tries to compute a different work-unit from its own stack whenever it aborts. It does not require visible readers and can be applied to any word-based software transactional memory system. Moreover, [4] focuses on a single optimization within a

software transactional memory system, where the proposed skeleton-driven approach has a set of transparent optimizations.

One of the optimizations developed in this thesis exploits the use of helper threads to improve performance. Helper threads have been explored as a means of exploiting otherwise idle hardware resources to improve performance on multi-cores [21]. A compiler framework to automatically generate software helper threads code for profitable loops in sequential applications has been developed [72]. As in our system, it prefetches data to shared cache levels. Finally, in [65] helper threads were manually coded within hardware transactional memory barriers, to improve the performance of a sequential implementation of Dijkstra's algorithm. In this case, a helper thread does useful work rather than just prefetching. In the proposed approach, helper threads are employed to speedup transactional applications rather than sequential applications. Additionally, synchronization between the worker and helper threads is solved by the use of pattern-oriented information.

8.2.2 System-Oriented Performance Optimizations

Mechanisms to enhance memory affinity on data parallel applications such as thread scheduling and NUMA-aware memory allocation have been studied in [6, 11]. In [11], the authors proposed a NUMA-aware runtime implementation for OpenMP interface. Differently to those proposed system-oriented optimizations, OpenSkel runtime relies on thread schedulers to efficiently manage memory affinity. In [6], the authors designed dynamic mechanisms that decide data placement over the physical memory of a NUMA machine. The proposed mechanisms use hardware counters to compute the most suitable data placement (e.g., queuing delays, on-chip latencies, and row-buffer hit-rates of on-chip memory controllers). Due to this, the target architecture must implement the hardware support for these counters. Although these mechanisms showed reasonable performance in simulations, when employed in real systems they led to small gains.

Some studies on thread mapping for MPI and OpenMP applications are relevant to this thesis, since they propose heuristics to map threads on multi-core platforms. In [83], the authors presented a process mapping strategy for MPI applications with collective communications. The strategy uses the graph partitioning algorithm to generate an appropriate process mapping for an application. The proposed strategy was

then compared with *Compact* and *Scatter*. In [27], two thread mapping algorithms are proposed for applications based on the shared memory programming model. These algorithms rely on memory traces extracted from benchmarks to find data sharing patterns between threads. During the profiling phase, these patterns are extracted by running the workloads in a simulated machine. The proposed approach was compared to *Compact*, *Scatter* and the operating system process mapping strategies. In [41], the authors proposed a dynamic thread mapping strategy for regular data parallel applications implemented with OpenMP. The strategy considers the machine description and the application characteristics to map threads to processors. The efficiency of their proposed thread mapping strategy was evaluated using simulation.

In this thesis, memory allocation and thread mapping strategies are applied under a skeleton framework to transactional worklist applications. To the best of our knowledge, this is the first work to implement and autotune these optimizations for the target applications.

8.3 Autotuning Software Systems

8.3.1 Autotuning Mechanisms for Parallel Applications

In [47], the authors propose an autotuning framework for stencil applications. It automatically parallelizes and tunes sequential Fortran stencil applications. The autotuning strategy selects a subset of the optimization search space instantiations based on architectural constraints. Then it runs each instantiation, measures the execution time and chooses the fastest instantiation for the target machine. It is an off-line approach based on exhaustive search as opposed to the proposed autotuning mechanism in this thesis.

Another class of automatic tuning systems is based on *Machine Learning* techniques. This approach involves off-line training to build a predictor used by the runtime system to choose between the available alternatives. It has been applied to partition data-parallel OpenCL tasks [33], to select task scheduling strategies [76] and map threads [78] in OpenMP parallel programs. It has become also common applying ML-based approaches to autotune the Java runtime system. In [71], the authors applied a decision tree to automatically select among various garbage collection strategies in Java for MapReduce applications.

8.3.2 Autotuning Mechanisms for STMs

In [56], the authors use a history-based heuristic to dynamically select the most efficient STM design considering dimensions such as level of indirection and type of non-blocking semantics. Automatic tuning of STM systems is considered in [29], which proposes a hill climbing strategy to adjust the number of locks, the number of shifts in a hash function and the size of the hierarchical array implemented on TinySTM. In contrast, this thesis focuses on the tuning of higher level optimizations rather than STM internal designs and parameters. Transactional Memory contention managers ([35, 73]) dynamically react to contention and try to minimize it by changing the *conflict resolution* policy, for example, in order to improve performance. In this thesis, the proposed scheme does take contention into account, but targets a broader range of optimizations. In [82], the authors propose an adaptive mechanism to transaction scheduling on top of a STM contention manager. It uses contention feedback information to adjust the number of available active transactions running in parallel to avoid high contention. In [3], the authors also propose a concurrency control mechanism that continuously tunes the number of threads based on the transaction commit rate. In contrast, this thesis proposes an autotuning mechanism that follows a hill climbing strategy to select the most suitable number of threads in a few intervals rather than during the whole application execution.

Chapter 9

Conclusions and Future Work

9.1 Summary of Contributions

Although many parallel programming models and systems have been proposed in the recent past, improving performance and programmability of applications is still hard and error-prone. Transactional memory is a recent attractive programming model that simplifies the extraction of coarse-grained parallelism formerly limited by lock programming. In this programming model, critical sections are speculatively executed and data conflicts solved by a runtime system. However, it still resembles lock programming in the sense that the application programmer has to explicitly point out the critical sections and to create and synchronize threads. These are limitations that have been possibly preventing TM to be largely adopted by the larger programming community as an alternative to lock-based programming. Additionally, the TM system is limited to implement optimizations in the transactional level. Although these optimizations are essential to achieve high performance, they do not take into account the algorithmic structure of the application. As a consequence, high-level optimizations that could help to enhance the TM system even further and also the application as a whole are not fully exploited.

Parallel algorithmic skeletons are designed to encapsulate parallelism issues, such as thread management, leaving the programmer with the single task of programming the application kernel. These parallel skeletons can naturally complement transactional memory programming as long as transactional applications present a well-defined pattern. Additionally, these high-level constructs reveal structural information of the applications otherwise not available to the TM system. This information can be used to provide skeleton-driven performance optimizations that are transparent to the applica-

tion programmer.

This thesis presents a new skeleton framework called OpenSkel to improve performance and programmability of a subset of transactional memory applications. The main idea is to enhance transactional memory programming by providing an abstraction that reliefs the application programmer from the burden of dealing with thread management and TM issues such as barriers placement. In addition to simplifying the programming task, this framework also frees the programmer from manually tuning the application by automatically enabling and selecting various pattern-oriented performance optimizations. This framework is also extended to provide automatic system-oriented optimizations such as thread mapping and memory allocation. These platform dependent optimizations also impacts the application performance and should be addressed.

To the best of our knowledge, this thesis is the first to provide a skeleton-based framework to a subset of existing TM applications and to automatic tune a set of pattern-oriented optimizations according to the application behavior. In particular, the proposed autotuning mechanism presents a novel approach that successfully allows our framework to achieve performance improvements that are close to, and sometimes surpass a static oracle. Additionally, this thesis also automates the selection of existing system-oriented optimizations when applied to transactional worklist applications. In addition to simplifying programming of transactional applications, the experimental results presented in this thesis show that ultimately combining system-oriented and pattern-oriented optimizations the OpenSkel framework can automatically achieve performance improvements of up to 88%, with an average of 46%, over a baseline version for a 16-core UMA platform and up to 162%, with an average of 91%, for a 32-core NUMA platform.

9.2 Future Work

This research can be extended in many directions. Probably, the most obvious improvement is to add new pattern-oriented and system-oriented optimizations. Particularly, the *helper threads* optimization can be used not only for data prefetching but also as profiling threads. For instance, HTs could record page memory accesses and forward this information to the memory allocator. Then, the allocator could do a smarter memory allocation to avoid remote accesses or even to migrate pages it is profitable. Nevertheless, HTs could also be also applied to non-transactional worklist applica-

tions.

Regarding the autotuning mechanism, it can be enhanced considering new optimizations and STM configurations. For instance, it can choose the best locking and contention management strategies depending on the application behavior. This autotuning mechanism could also apply dynamic tuning to *helper threads* and *thread concurrency level* throughout the application execution if profitable.

Another important continuation to this thesis is to extend OpenSkel to accommodate more patterns and allow composability between them. For instance, the pipeline pattern can be created by composing a sequence of worklists. Finally, it is the intention of the authors to continue the development of OpenSkel, integrating it to existing compilers and extending them to produce helper thread code.

Bibliography

[1] Marco Aldinucci and Marco Danelutto. Skeleton-based Parallel Programming: Functional and Parallel Semantics in a Single Shot. *Computer Languages, Systems and Structures*, 33(3-4):179–192, October 2007.

[2] Marco Aldinucci, Marco Danelutto, and Patrizio Dazzi. MUSKEL: An Expandable Skeleton Environment. *Scientific International Journal for Parallel and Distributed Computing*, 8:325–341, December 2007.

[3] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Advanced Concurrency Control for Transactional Memory using Transaction Commit Rate. In *Euro-Par'08: Proceedings of International Euro-Par Conference on Parallel Processing*, pages 719–728, August 2008.

[4] Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris C. Kirkham, and Ian Watson. Steal-on-Abort: Improving Transactional Memory Performance through Dynamic Transaction Reordering. In *HiPEAC'09: Proceedings of the International Conference on High-Performance and Embedded Architectures and Compilers*, pages 4–18, January 2009.

[5] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Katherine A. Yelick. A View of the Parallel Computing Landscape. *Communications of the ACM*, 52(10):56–67, October 2009.

[6] Manu Awasthi, David W. Nellans, Kshitij Sudan, Rajeev Balasubramonian, and Al Davis. Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers. In *PACT'10: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 319–330, September 2010.

[7] Woongki Baek, Chi C. Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. The OpenTM Transactional Application Programming Interface. In *PACT'07: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 376–387, September 2007.

[8] Robert Blumofe, Christopher Joerg, Bradley Kuszmaul, Charles Leiserson, Keith Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[9] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In *ISCA'08: Proceedings of the International Symposium on Computer Architecture*, pages 127–138, June 2008.

[10] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance Pathologies in Hardware Transactional Memory. In *ISCA'07: Proceedings of the International Symposium on Computer Architecture*, pages 81–91, June 2007.

[11] François Broquedis, Olivier Aumage, Brice Goglin, Samuel Thibault, Pierre-André Wacrenier, and Raymond Namyst. Structuring the Execution of OpenMP Applications for Multicore Architectures. In *IPDPS '10: Proceedings of the International Symposium on Parallel and Distributed Processing*, April 2010.

[12] François Broquedis, Jérôme Clet Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *PDP'10: Proceedings of the International Conference on Parallel, Distributed and Network-based Computing*, pages 180–186, February 2010.

[13] S. Browne, Jack Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications*, 14:189–204, August 2000.

[14] Duncan K. G. Campbell. Towards the Classification of Algorithmic Skeletons. Technical Report YCS 276, Dep. of Computer Science, University of York, December 1996.

[15] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC'08: Proceedings of the International Symposium on Workload Characterization*, pages 35–46, September 2008.

[16] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why is it only a Research Toy? *Communications of the ACM*, 51(11):40–46, November 2008.

[17] Márcio Castro, Luís Fabrício Wanderley Góes, Christiane Pousa Ribeiro, Murray Cole, Marcelo Cintra, and J.-F. Méhaut. A Machine Learning-based Approach for Thread Mapping on Transactional Memory Applications. In *HiPC'11: Proceedings of High Performance Computing Conference*, 2011.

[18] Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen Mcdonald, Chi C. Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun. A Scalable, Non-blocking Approach to Transactional Memory. In *HPCA'07: Proceedings of the International Symposium on High Performance Computer Architecture*, pages 97–108, April 2007.

[19] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press & Pitman, October 1989.

[20] Murray Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, March 2004.

[21] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative Precomputation: Long-Range Prefetching of Delinquent Loads. In *ISCA'01: Proceedings of the International Symposium on Computer Architecture*, pages 14–25, May 2001.

[22] Intel Corporation. *Intel C++ STM Compiler Prototype Edition: Language Extensions and Users Guide*, 2007.

[23] Intel Corporation. *OpenMP and Intel IPP Library*. October 2010.
URL: *http://software.intel.com/en-us/articles/openmp-and-the-intel-ipp-library*.

[24] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, January 1998.

[25] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 137–150, December 2004.

[26] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *DISC'06: Proceedings of the International Symposium on Distributed Computing*, pages 194–208, September 2006.

[27] Matthias Diener, Felipe Madruga, Eduardo Rodrigues, Marco Alves, Jorg Schneider, Philippe Navaux, and Hans-Ulrich Heiss. Evaluating Thread Placement based on Memory Access Patterns for Multi-core Processors. In *HPCC'10: Proceedings of the International Conference on High Performance Computing and Communications*, pages 491–496, September 2010.

[28] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a Research Toy. *Communications of the ACM*, 54(4):70–77, April 2011.

[29] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic Performance Tuning of Word-based Software Transactional Memory. In *PPoPP'08: Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 237–246, February 2008.

[30] Pascal Felber, Torvald Riegel, Christof Fetzer, Martin Subkraut, Ulrich Muller, and Heiko Sturzrehm. Transactifying Applications Using an Open Compiler Framework. In *TRANSACT'07: Proceedings of the Workshop on Transactional Computing*, pages 56–64, August 2007.

[31] Keir Fraser and Tim Harris. Concurrent Programming without Locks. *ACM Transactions on Computer Systems*, 25(2):1–61, May 2007.

[32] Horacio González-Vélez and Mario Leyton. A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers. *Software Practice and Experience*, 40(12):1135–1160, November 2010.

[33] Dominik Grewe and Michael F.P. O'Boyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In *CC '11: Proceedings of the 20th International Conference on Compiler Construction*, pages 286–305, March 2011.

[34] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, November 1999.

[35] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a Theory of Transactional Contention Managers. In *PODC'05: Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 258–264, July 2005.

[36] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional Memory Coherence and Consistency. In *ISCA'04: Proceedings of the International Symposium on Computer Architecture*, pages 102–112, June 2004.

[37] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, December 2010.

[38] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable Memory Transactions. In *PPoPP'05: Proceedings of Symposium on Principles and Practice of Parallel Programming*, pages 48–60, June 2005.

[39] Jim Held, Jerry Bautista, and Sean Koehl. From a Few Cores to Many: A Terascale Computing Research Overview, White Paper, Intel Corporation 2006.

[40] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *ISCA'93: Proceedings of the International Symposium on Computer Architecture*, pages 289–300, May 1993.

[41] S. Hong, S. H. K. Narayanan, M. Kandemir, and Ö. Özturk. Process Variation Aware Thread Mapping for Chip Multiprocessors. In *DATE'09: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 821–826, April 2009.

[42] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computing: Technology, Architecture, Programming*. McGraw-Hill, Inc., September 1997.

[43] Mary J. Irwin and John P. Shen. Revitalizing Computer Architecture Research. In *CRA Conference on Grand Research Challenges*, pages 1–21, December 2005.

[44] Emmanuel Jeannot and Guillaume Mercier. Near-Optimal Placement of MPI Processes on Hierarchical NUMA Architectures. In *EuroPar'10: Proccedings of the International Conference on Parallel Processing*, pages 199–210, September 2010.

[45] Simon P. Jones. Beautiful Concurrency. In *Beautiful Code: Leading Programmers Explain How They Think (Theory in Practice)*, pages 385–406, July 2007.

[46] Laxmikant Kale and Sanjeev Krishnan. CHARM++: A Portable Concurrent Object Oriented System based on C++. In *OOPSLA'93: Proceedings of the Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 91–108, October 1993.

[47] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. An Auto-Tuning Framework for Parallel Multicore Stencil Computations. In *IPDPS'10: Proceedings of the International Symposium on Parallel and Distributed Processing*, pages 1–12, April 2010.

[48] Andi Kleen. A NUMA API for Linux. Technical Report Novell-4621437, August 2005.

[49] Milind Kulkarni, Martin Burtscher, Rajasekhar Inkulu, Keshav Pingali, and Calin Cascaval. How Much Parallelism is There in Irregular Applications? In *PPoPP'09: Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 1–12, February 2009.

[50] Milind Kulkarni, Paul Chew, and Keshav Pingali. Using Transactions in Delaunay Mesh Generation. In *WTW'06: Proceedings of the Workshop on Transactional Memory Workloads*, pages 23–31, June 2006.

[51] Milind Kulkarni, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and Paul L. Chew. Optimistic Parallelism Benefits from Data Partitioning. In *ASPLOS'08: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 233–243, March 2008.

[52] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and Paul L. Chew. Optimistic Parallelism Requires Abstractions. In *PLDI'07: Proceedings of the Conference on Programming Language Design and Implementation*, pages 211–222, June 2007.

[53] Doug Lea. A Java Fork/Join Framework. In *JAVA'00: Proceedings of the International Conference on Java Grande*, pages 36–43, June 2000.

[54] Edward A. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33–42, May 2006.

[55] Mario Leyton. *Advanced Features for Algorithmic Skeleton Programming*. PhD thesis, Universite de Nice - Sophia Antipolis – UFR Sciences, October 2008.

[56] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive Software Transactional Memory. In *DISC'05: Proceedings of the International Symposium on Distributed Computing*, pages 354–368, September 2005.

[57] Virendra J. Marathe and Michael L. Scott. A Qualitative Survey of Modern Software Transactional Memory Systems. Technical Report TR839, University of Rochester, June 2004.

[58] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer, and Michael L. Scott. Lowering the Overhead of Software Transactional Memory. In *TRANSACT'06: Proceedings of the Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[59] Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Scalable Techniques for Transparent Privatization in Software Transactional Memory. In *ICPP'08: Proceedings of the International Conference on Parallel Processing*, pages 67–74, September 2008.

[60] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Pearson Education, September 2004.

[61] Michael McCool. Structured Parallel Programming with Deterministic Patterns. In *HotPar'10: Proceedings of the USENIX Conference on Hot Topics in Parallelism*, pages 25–30, June 2010.

[62] Chao Mei, Gengbin Zheng, Filippo Gioachin, and Laxmikant V. Kalé. Optimizing a Parallel Runtime System for Multicore Clusters: A Case Study. In *TG'10: Proceedings of the TeraGrid Conference*, pages 12:1–12:8, August 2010.

[63] Mario Méndez-Lojo, Donald Nguyen, Dimitrios Prountzos, Xin Sui, M. Amber Hassaan, Milind Kulkarni, Martin Burtscher, and Keshav Pingali. Structure-Driven Optimizations for Amorphous Data-Parallel Programs. In *PPoPP'10: Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 3–14, January 2010.

[64] Kevin Moore, Jayaram Bobba, Michelle Moravan, Mark Hill, and David Wood. LogTM: Log-based Transactional Memory. In *HPCA'06: Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 254–265, February 2006.

[65] Konstantinos Nikas, Nikos Anastopoulos, Georgios Goumas, and Nectarios Koziris. Employing Transactional Memory and Helper Threads to Speedup Dijkstra's Algorithm. In *ICPP'09: Proceedings of the International Conference on Parallel Processing*, pages 388–395, September 2009.

[66] Christiane Pousa Ribeiro, Márcio Castro, Alexandre Carissimi, and Jean-François Méhaut. Improving Memory Affinity of Geophysics Applications on NUMA platforms Using Minas. In *VECPAR'10: Proceedings of the International Conference on High performance Computing for Computational Science*, pages 279–292, June 2010.

[67] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing Transactional Memory. In *ISCA'05: Proceedings of the International Symposium on Computer Architecture*, pages 494–505, May 2005.

[68] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA'07: Proceedings of the International Symposium on High Performance Computer Architecture*, pages 13–24, February 2007.

[69] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, July 2007.

[70] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is Transactional Programming Actually Easier? In *PPoPP'10: Proceedings of the International Symposium on Principles and Practice of Parallel Programming*, pages 47–56, January 2010.

[71] Jeremy Singer, George Kovoor, Gavin Brown, and Mikel Luján. Garbage Collection Auto-Tuning for Java MapReduce on Multi-Cores. In *ISMM'11: Proceedings of the International Symposium on Memory Management*, pages 109–118, June 2011.

[72] Yonghong Song, Spiros Kalogeropulos, and Partha Tirumalai. Design and Implementation of a Compiler Framework for Helper Threading on Multicore Processors. In *PACT'05: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 99–109, August 2005.

[73] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A Comprehensive Strategy for Contention Management in Software Transactional Memory. In *PPoPP'09: Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 32–40, February 2009.

[74] Michael F. Spear, Michael Silverman, Luke Dalessandro, Maged M. Michael, and Michael L. Scott. Implementing and Exploiting Inevitability in Software Transactional Memory. In *ICPP'08: Proceedings of the International Conference on Parallel Processing*, pages 59–66, September 2008.

[75] Fuad Tabba, Cong Wang, and James R. Goodman. NZTM: Nonblocking Zero-Indirection Transactional Memory. In *SPAA'09: Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, pages 204–213, June 2007.

[76] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning based Mapping. In *PLDI '09: Proceedings of the Conference on Programming Language Design and Implementation*, pages 177–187, June 2009.

[77] Christoph von Praun, Luis Ceze, and Calin Cascaval. Implicit Parallelism with Ordered Transactions. In *PPoPP'07: Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 79–89, February 2007.

[78] Zheng Wang and Michael F.P. O'Boyle. Mapping Parallelism to Multi-Cores: A Machine Learning based Approach. In *PPoPP'09: Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 75–84, February 2009.

[79] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, May 2009.

[80] Polychronis Xekalakis, Nikolas Ioannou, and Marcelo Cintra. Combining Thread Level Speculation, Helper Threads and Runahead Execution. In *ICS'09: Proceedings of the International Conference on Supercomputing*, pages 410–420, June 2009.

[81] Gagarine Yaikhom, Murray Cole, Stephen Gilmore, and Jane Hillston. A Structural Approach for Modelling Performance of Systems Using Skeletons. *Electronic Notes in Theoretical Computer Science*, 190(3):167–183, September 2007.

[82] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive Transaction Scheduling for Transactional Memory Systems. In *SPAA '08: Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, pages 169–178, June 2008.

[83] Jin Zhang, Jidong Zhai, Wenguang Chen, and Weimin Zheng. Process Mapping for MPI Collective Communications. In *Euro-Par'09: Proceedings of the International Euro-Par Conference on Parallel Processing*, pages 81–92, August 2009.