ASAP: AUTOMATIC SPECULATIVE ACYCLIC PARALLELIZATION FOR CLUSTERS

HANJUN KIM

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

ADVISOR: PROFESSOR DAVID I. AUGUST

SEPTEMBER 2013

Abstract

While clusters of commodity servers and switches are the most popular form of large-scale parallel computers, many programs are not easily parallelized for clusters due to high internode communication cost and lack of globally shared memory. Speculative Decoupled Software Pipelining (Spec-DSWP) is a promising automatic parallelization technique for clusters that speculatively partitions a loop into multiple threads that communicate in a pipelined manner. Speculation can complement conservative static analysis, making automatic parallelization more robust and applicable. Pipelining allows Spec-DSWP to speculate only rarely occurring dependences while respecting the other dependences through communication among threads. Acyclic communication patterns in pipelining make the parallelized programs tolerant of high communication latency of clusters. However, since Spec-DSWP partitions a loop iteration (a transaction) into multiple sub-transactions across multiple threads according to the pipeline stages, a special runtime system is required that supports multi-threaded transactions (MTXs).

This dissertation proposes the Automatic Speculative Acyclic Parallelization (ASAP) system that enables Spec-DSWP for clusters without any hardware modification. The ASAP system supports various speculation techniques that require different validation and communication costs, and automatically parallelizes sequential loops using the Spec-DSWP transformation with the optimal application of the speculation techniques. The ASAP system efficiently supports MTXs to correctly execute the speculatively transformed programs on clusters. With synergistic combination of speculation, acyclic communication, and runtime system support, this approach achieves or demonstrates a path to achieve scalable performance speedup up to 109× for a wide range of applications on clusters without any hardware modification.

Acknowledgments

First and foremost, I sincerely thank my advisor, Prof. David August, for his assistance and support throughout my years in graduate school at Princeton. He has taught me a great deal about research from reading papers with critical thinking and picking my research topics to conducting my research and presenting the results. In particular, his insightful suggestions and encouragement made me continue my research especially when I felt difficulty in building the proposed system in this dissertation.

I sincerely thank the members of my dissertation committee, Prof. Kai Li, Prof. David Wentzlaff, Prof. Shard Malik, and Prof. Jaswinder Singh. In particular, I thank my advisor and my readers, Prof. Kai Li and Prof. David Wentzlaff, for carefully reading this dissertation and providing insightful and detail comments. Their collective wisdom and feedback have improved this dissertation. I thank Prof. Shard Malik and Prof. Jaswinder Singh for serving on my committee and for their support and feedback.

This dissertation would not be possible without the support of the Liberty research group, including Arun, Yun, Tom, Prakash, Jialu, Jack, Nick, Thomas, Feng, Stephen, Taewook, Deep, Matt, Jordan, and Hao. Tom, Prakash, and Nick have greatly contributed to the LLVM liberty compiler that is used in this research, and I thank for both their friendship and support. I also specially and sincerely thank my closest collaborators, Arun, Nick, and Jae. Many of the ideas described in this dissertation from the compiler to the runtime system were made and refined through long discussions with them over the years. I thank Taewook for many interesting discussions on my future research topics.

I would like to thank Intel Corporation and Siebel Scholars Foundation for supporting my work with the Graduate Research Fellowship and the Siebel Scholars Program. I also want to acknowledge that the evaluation in this dissertation was performed on computational resources supported by the PICSciE-OIT High Performance Computing Center and Visualization Laboratory.

I have been fortunate to have had great administrative support from the entire staff of the Department of Computer Science and Princeton University. I especially thank Melissa Lawson for her assistance on handling travel grants, fellowship applications, conference organization, and a host of other things. She helped me be accustomed to Princeton, and made my life in Princeton a lot easier.

Finally, I thank my friends and family. I thank my parents, my parents-in-law, my sister Soowoo, and my brother-in-law Hyukmin for their support throughout this process with their love and encouragement. I thank my parents again for their unconditional love, encouragement, and confidence in me, since I was young. I cannot image how much they sacrificed to allow me to achieve what I have today. Above all, my wife KyungSun has my sincere thanks for her love, care, and understanding throughout not only writing of this dissertation, but also most of my life.

For my lovely wife, KyungSun

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Clusters of commodity servers and switches are one of the most popular large-scale parallel computing forms to speed up the execution of programs beyond the performance achievable on a single-board computer. While clusters provide scalable hardware resources such as processor cores, memory, and I/O bandwidth, programs need to be parallelized to efficiently utilize these parallel hardware resources for its performance improvement. As a result, clusters are primarily used for scientific programs or web services, which consist of units of work that are mostly independent.

However, extracting scalable parallelism from sequential programs on clusters is challenging for two main reasons. First, commodity clusters do not provide shared memory. This requires the parallelizer (programmer or compiler) to identify shared data and explicitly insert communication primitives between producers and consumers. Second, clusters have high inter-node communication latency. Without careful communication optimization, the inter-node communication cost easily becomes a performance bottleneck.

There are two main strategies for scalable, efficient parallelization on clusters: parallel programming methods and automatic parallelization methods. Explicit parallel programming using a message passing protocol (e.g., MPI) is one potential solution to the problem, but it can severely limit the programmer's productivity by requiring a deep knowledge of

```
a: Loop_A:
b: for (int i=0; i<N; i++)
c:  regular[i] += foo(i);

d: Loop_B:
e: for (int i=0; i<N; i++) {
f:   irregular[idx[i]] += foo(i);
g:   if (irregular[idx[i]] > error)
h:     printf("I/O operation!");
i: }
```

```
j: Loop_C:
k: while(node) {
l:   node = node->next;
m:   res = boo(node);
n:   printf("result: %d", res);
o: }
```

Figure 1.1: Sequential code example with three loops

concurrency, domain expertise, and platform-specific performance tuning. Parallelization APIs such as OpenMP [1] can help programmers parallelize sequential programs. Using the APIs, the programmers annotate what and how to parallelize the sequential programs. Then, the compiler generates parallel codes according to the annotation. However, the APIs still require the programmers to analyze the data and control dependences of the program to find effective parallelization strategies.

Automatic parallelization is an attractive alternative to the time-consuming, error-prone manual parallelization. Parallelizing compilers can automatically parallelize affine loops [5, 12, 14, 47]. `Loop_A` in Figure 1.1 shows such an example code. If a compiler proves that all the memory variables in the body of the function `foo` do not alias the array `regular` via inter-procedural analysis, the compiler can parallelize the loop. Therefore, the utility of an automatic parallelizing compiler is largely determined by the quality of its memory dependence analysis.

In some cases, static analysis may be imprecise. For example, within the function `foo`, assume that there is a read from or write to the array element `regular[i+M]`, (and the size of the array is greater than (`M+N`)), where `M` is an input from the user. In this case, `Loop_A` may not be DOALL-able depending on the value of `M`. If `M` is greater than `N`, the loop is DOALL-able; otherwise, it is not. Some research compilers such as SUIF [5] and Polaris [12, 70] integrate low-cost run-time analysis capabilities to insert a small test code to check the value of `M` at run-time to select either a sequential or parallel version of the

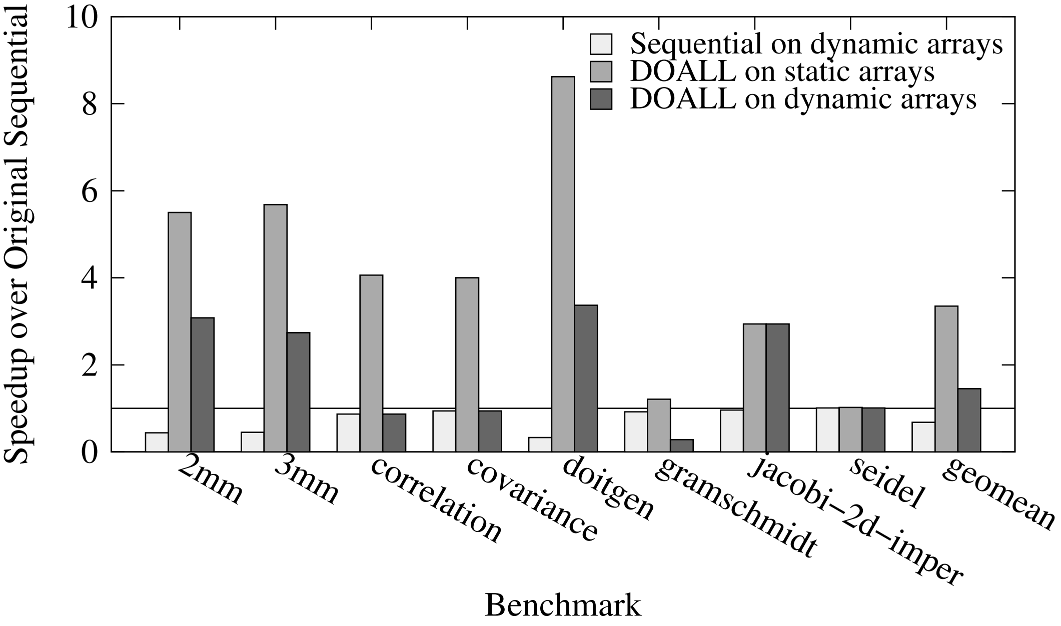


Figure 1.2: Performance sensitivity due to memory analysis on a shared-memory machine. Details about each program are described at Chapter 6

loop accordingly. However, the coverage of these techniques is mostly restricted to the cases when a predicate can be extracted outside the analyzed loop and a low cost run-time test can be generated [70]. They cannot be applied to `Loop_B` in Figure 1.1, for example, where an index array is used to access the array `irregular` and a simple predicate cannot be extracted outside the loop due to the `if` condition within the loop body.

Another issue with automatic parallelization is the fragility of static analysis. Figure 1.2 illustrates how fragile static analysis can be with a small change in the program. In this example, the Liberty compiler [78] without speculation support can easily parallelize the unmodified array-based PolyBench benchmarks [61] using static arrays. However, if programmers replace the static arrays with dynamically allocated arrays, the replacement does not only suppress some of the optimizations previously applied, but also blocks parallelization for several benchmarks since heap objects are generally more difficult to analyze. This shows that the optimization path and run-time performance are highly affected by how a program is implemented.

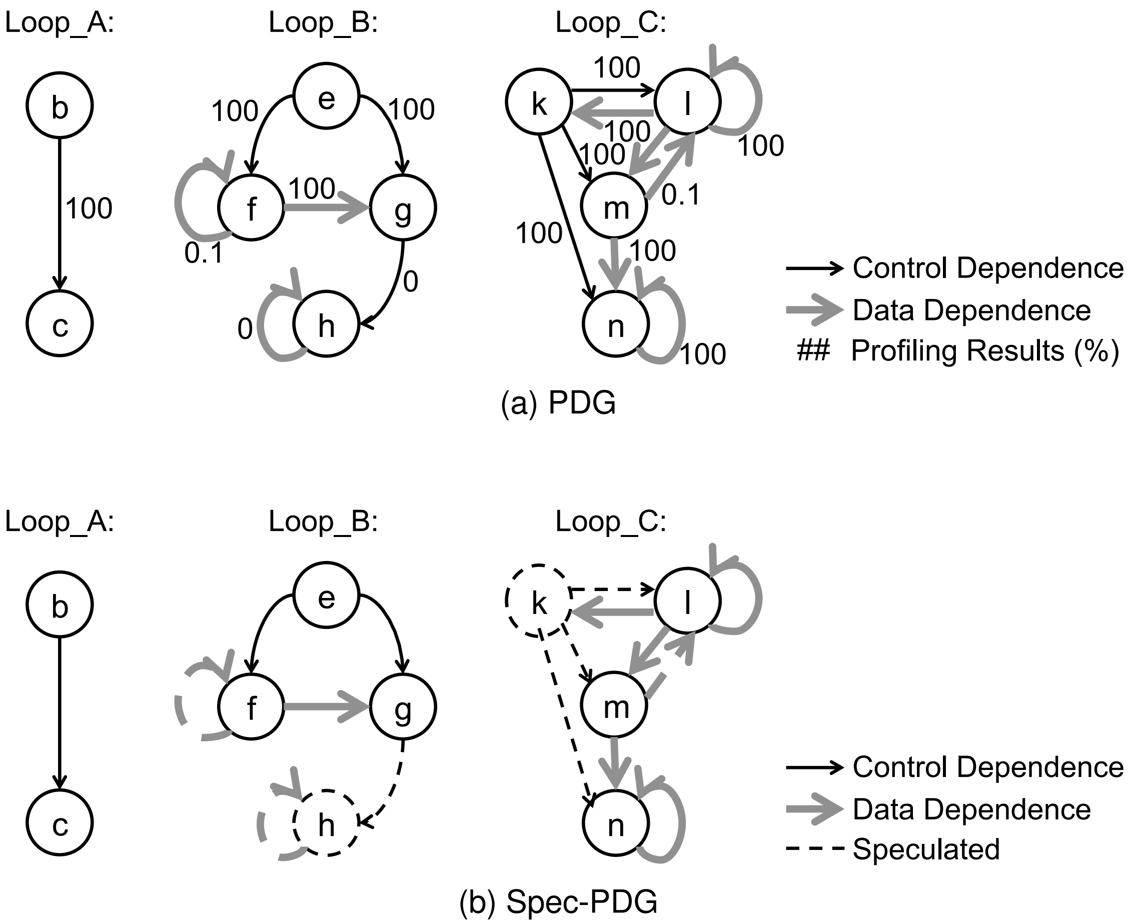
Therefore, analysis-based approaches, both static and dynamic, are not sufficient for parallelization of even array-based applications, let alone pointer-based ones, having irregular memory accesses and complex control flows. Moreover, recursive data structures, dynamic memory allocation, and frequent accesses to shared variables pose additional challenges. Imprecise, fragile static analysis has severely limited the applicability of conventional automatic parallelization.

Automatic speculative parallelization [49, 52, 64, 76, 89] can overcome the limitations of static compiler analysis. These compilers speculatively remove memory or control dependences among instructions, and optimistically parallelize loops. In addition, the compilers can speculatively remove irregular dependences that manifest infrequently at run-time. Speculation complements the imprecision and the fragility of the conservative static analysis, and makes automatic parallelization more robust and applicable. For example, the compiler can apply Speculative DOALL (Spec-DOALL) parallelization to `Loop_B` in Figure 1.1, speculating that no cross-iteration dependence violation occurs via concurrent array accesses and that the error condition in Line `g` does not happen at run-time. This approach requires runtime support for misspeculation detection and recovery in either hardware or software to ensure correctness.

1.1 Speculative Loop Parallelization with Communication Support

Since misspeculation penalizes the performance, highly accurate speculation is crucial for scalable performance. To increase speculation hit ratio, it is necessary to speculate only rarely occurring dependences while respecting the other dependences through communication among threads.

DOALL and Spec-DOALL parallelizations partition the iteration space into groups that are executed concurrently with no inter-thread communication. The Program Dependence



(a) PDG

(b) Spec-PDG

Figure 1.3: PDG with profiling results and Spec-PDG based on the profiling results for the example in Figure 1.1

Graph (PDG) in Figure 1.3(a) shows control and memory data dependences between statements for the code in Figure 1.1. Since statement c in Loop_A is executed only when the for loop condition at b is true, there is a control dependence from b to c. For the same reason, there are control dependences from e to f and g in Loop_B, and from k to l, m and n in Loop_C. Since statement h in Loop_B can be executed only when the if condition at g is true, there is a control dependence from g to h. Since different iterations in Loop_B can access the same memory address of the irregular array at f, there is a loop carried data dependence on f. There is an intra-iteration data dependence from f to g because g reads the updated irregular array at f in the same iteration. Due to the linked list access, there are loop carried data dependences from l to k and l in Loop_C, and an intra-iteration data dependence from l to m. There is a loop carried data dependence on m because function boo may touch the same memory space at different iterations. Due to I/O operations, there is a loop carried data dependence on statements h and n.

If a PDG for a loop does not have inter-iteration dependences that indicated by cycles in the graph, each iteration in the loop can be independently executed without communication, so the loop is DOALL-able. For example, Loop_A is DOALL-able because its PDG does not have inter-iteration dependences. Although Loop_B is not DOALL-able due to the two inter-iteration dependences, Loop_B is Spec-DOALL-able because the inter-iteration dependences rarely manifest according to profiling results, and compilers or programmers can speculatively remove the dependences as shown in the Speculative PDG (Spec-PDG) in Figure 1.3(b). Since f rarely touches the same memory space, and the if condition at g is rarely true, the compilers or programmers can speculatively remove the dependences assuming they will not occur at run-time. Loop_A and Loop_B can be parallelized with DOALL and Spec-DOALL because their PDG and Spec-PDG do not have any inter-iteration dependence.

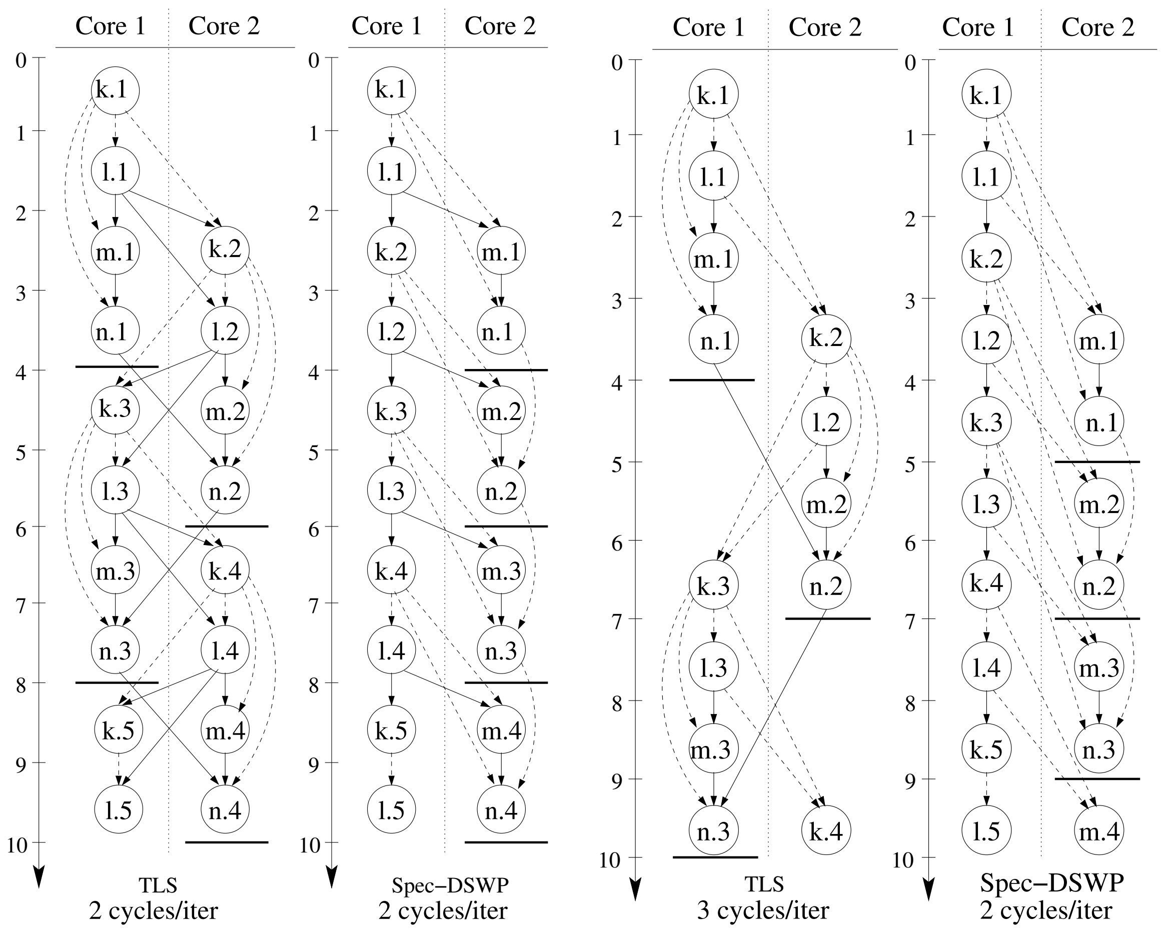
However, not all inter-iteration dependences can be removed. Loop_C in Figure 1.1 is such an example. Speculating that boo does not modify the linked list can remove memory dependences. However, predicting the values of node and the return values of boo on each iteration is extremely difficult in general. The inter-iteration dependences on statement l and statement n manifest every iteration according to the profiling results in Figure 1.3(a). If the dependences are speculatively removed, misspeculation will occur every iteration. The high misspeculation rate will degrade the performance, so the dependences should be respected with communication.

1.1.1 TLS and Spec-DSWP

There are two prominent schemes in speculative parallelization that support communication among threads: thread level speculation (TLS) [10, 58, 68, 75, 76, 90] and speculative decoupled software pipelining (Spec-DSWP) [82]. Both schemes handle inter-iteration dependences by means of communication among threads. TLS and Spec-DSWP break only *some* of the inter-iteration dependences using speculation, while respecting the others. The corresponding dependences are synchronized using communication. This approach selectively breaks the inter-iteration dependences with high confidence, thereby resulting in higher success rates.

Although the two techniques are comparable in applicability, Spec-DSWP provides more robust performance than TLS because Spec-DSWP is more tolerant to increases in inter-core communication latency. The difference is attributed to their inter-core communication patterns: TLS schedules the entire loop body iteration by iteration on alternate threads, so it exhibits a cyclic communication pattern among threads. However, Spec-DSWP partitions the loop body into multiple pipeline stages, and each thread executes each stage over all iterations making an acyclic communication pattern. The pipeline organization of Spec-DSWP keeps dependence recurrences local to a thread, avoiding communication latency on the critical path of program execution.

Figure 1.4 illustrates respective execution plans and their performance on different inter-core communication latency. Each node represents a dynamic instance of a statement in Figure 1.1, where the number indicates the iteration to which it belongs. The original sequential codes of `Loop_C` take 4 cycles to execute an iteration. As Figure 1.4(a) shows, ignoring the pipeline fill time, both TLS and Spec-DSWP execute the loop taking 2 cycles per iteration, so they yield a speedup of 2× using 2 threads in the steady state when the inter-thread communication latency is one cycle. When the inter-core latency increases from one to two cycles as in Figure 1.4(b), the speedup with TLS reduces to 1.33×, but the speedup with Spec-DSWP remains 2× in the steady state. TLS puts the inter-core communication latency on the critical path thus negating most parallelism benefits on multicore architectures, which have non-unit communication latency. In contrast to TLS, the Spec-DSWP approach selectively allows those cycles that contain hard-to-predict



(a) Comm. latency = 1 cycle

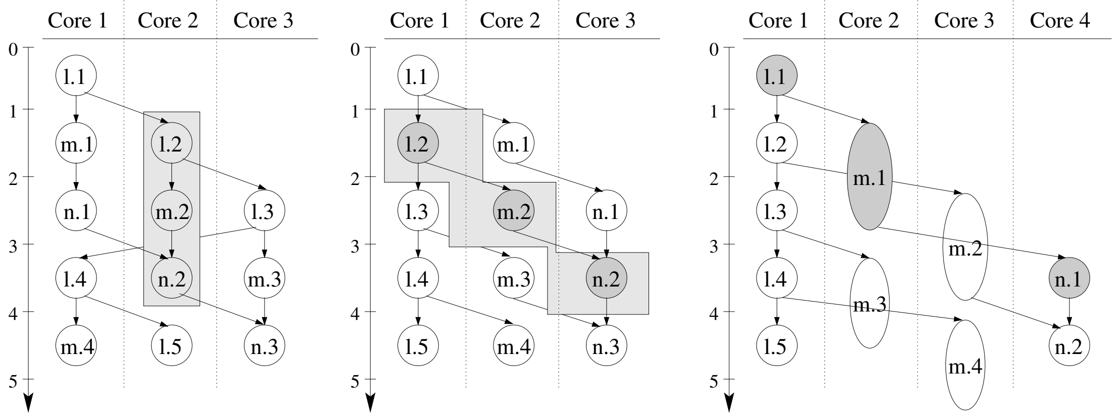
(b) Comm. latency = 2 cycles

Figure 1.4: DSWP keeps critical-path dependences thread-local and communication unidirectional; thus it is tolerant to increase in communication latency. The alphabets mean statements in Figure 1.1 and the numbers mean the iteration counts.

edges to remain thread-local, and is thus not penalized by inter-core communication latency [17, 41, 66, 82].

1.1.2 Multi-threaded Transaction

While Spec-DSWP shows more robust performance improvement, it requires a special runtime system to support speculative execution. TLS and Spec-DSWP are iteration-centric in that they remove inter-iteration dependences, so a loop iteration is the unit of atomic work (transaction). In TLS, each loop iteration is executed by a single thread. Consequently, the unit of atomic execution is single-threaded as shown as a lightly-shared box



(a) TLS (b) Spec-DSWP (c) Spec-DSWP+[S, DOALL, S]

Figure 1.5: The need for Multi-threaded Transactions (MTXs): `Loop_C` in Figure 1.1 can be parallelized with TLS (a) and with Spec-DSWP (b). While an atomic unit (shown as a lightly-shaded region) is confined to a single thread in (a), it spans multiple threads in (b). The darkly-shaded regions correspond to sub-transactions in an MTX. Spec-DSWP can help replicate stages without any cross-iteration dependences to use additional cores (c).

in Figure 1.5(a). Conventional transactional memory or TLS runtime systems that guarantee single-threaded atomicity may be used to support TLS. However, in Spec-DSWP, each loop iteration is executed in a staged manner by multiple threads, making the atomic unit multi-threaded as shown in Figure 1.5(b). Therefore, the conventional transactional memory systems cannot support Spec-DSWP.

To support the atomic units in Spec-DSWP, Multi-threaded Transactions (MTXs) [41, 66, 81] are required. An MTX represents an atomic set of memory accesses like its single-threaded counterpart, but may contain many sub-transactions (subTXs) each of which is executed by only one thread. In Figure 1.5(b), `l.2`, `m.2` and `n.2` are a MTX that has three subTXs. The MTX runtime system executes subTXs in the original sequential program order, and MTXs according to the sequential loop iteration order. Details about how the ASAP system supports MTXs is described in Section 3.2. An MTX with only one subTX degenerates to a single-threaded transaction. This allows the MTX runtime system to implicitly support TLS in addition to Spec-DSWP.

Poor scalability of DSWP due to a limited number of and imbalance among pipeline stages is not a problem. Huang et al. proposed DSWP+ that intentionally creates unbalanced pipeline stages to expose opportunities for scalable parallelization such as DOALL [36]. The example cannot be parallelized directly with DOALL due to the high misspeculation rate and the poor performance. DSWP+ can extract a DOALL stage (Stage m) that accounts for most of the execution time, and can exploit the scalability of DOALL. As more threads are assigned to this stage, the pipeline balance naturally improves. The *DSWP+[...]* notation describes the hybrid parallelization technique. Within square brackets, parallelization techniques applied to each stage are specified. The parallelized example in Figure 1.5(c) is expressed as Spec-DSWP+[S, DOALL, S] because the loop is parallelized with the Spec-DSWP scheme first, and then the second stage is parallelized with the DOALL scheme. Here, *S* indicates a stage that is sequentially executed, whereas *Spec-* indicates speculation between stages.

1.2 Parallel Computers without Cache-Coherent Shared Memory

Despite the relative ease and simplicity of programming shared memory machines, commodity clusters dominate for large-scale parallel processing. A cluster node, with one or more processor cores, typically has its own private physical memory address space and constitutes an independent domain of cache coherence; multiple nodes communicate via explicit message passing through I/O channels. Unlike Symmetric Multiprocessors (SMP), clusters have scalable per-processor memory and I/O bandwidth. In addition, commodity clusters have a cost advantage over cache-coherent Non-Uniform Memory Access (cc-NUMA) multiprocessors whose cost for cache coherence in hardware increases dramatically as the number of nodes increases. Because of their low cost, commodity clusters are the most widespread example of large-scale parallel computers that enable network services and high-performance computing today [60].

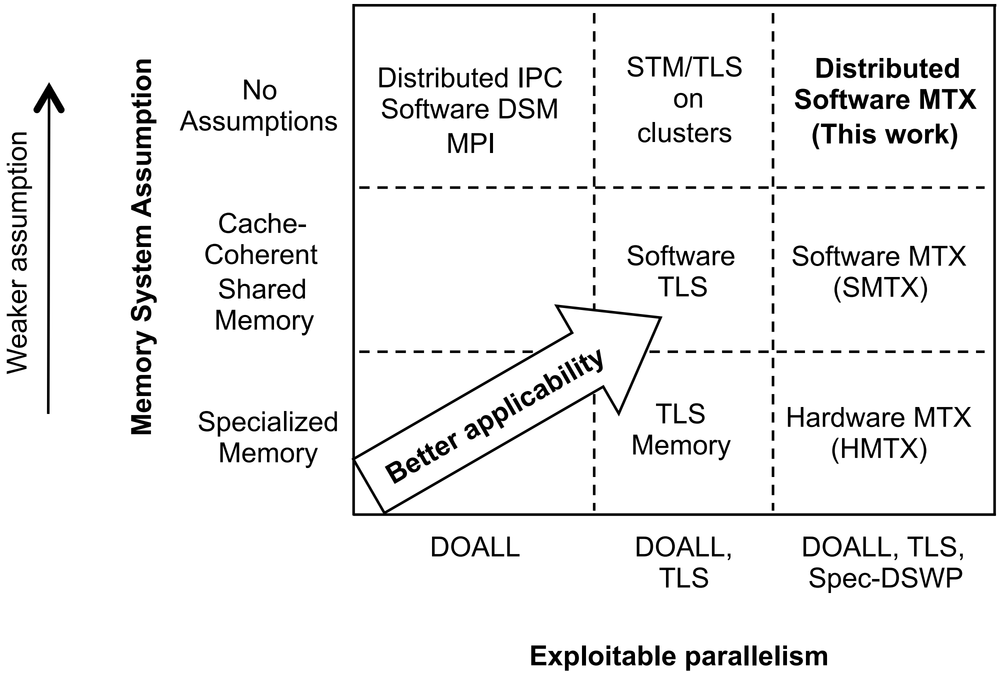


Figure 1.6: ASAP enables the widest variety of parallelization paradigms, while making the fewest assumptions about the underlying hardware.

Since the memory system of a cluster is physically distributed across multiple nodes without a globally shared address space, remote data must be explicitly sent and received between a producer-consumer pair using a message passing protocol such as MPI. The difficulty of message passing-style programming, combined with high inter-node communication latency, has limited the use of clusters to applications such as scientific codes, many of which have little communication and can be easily parallelized. Although distributed transactional memory systems [13, 24, 33, 42, 51, 87] can be modified to enable TLS on clusters, they do not support MTXs and hence cannot be used by Spec-DSWP+.

Vachharajani et al. and Raman et al. proposed MTX runtime systems that support Spec-DSWP [66, 81, 82]. However, the systems cannot run on commodity clusters because they require either hardware modification or cache-coherent shared memory. Unlike the runtime systems, this dissertation proposes a new system that explicitly inserts communication codes to share data between threads on distributed memory systems. To overcome high communication costs of clusters, the system supports additional speculation techniques that have low validation and communication overheads to achieve scalable performance improvement.

In addition to clusters, there are emerging multicore architectures targeted for certain workloads that discard even chip-wide cache coherence to minimize hardware cost and maximize energy efficiency. For example, Intel's 48-core architecture does not support hardware cache coherence [35]. Such processors rely on explicit message passing for efficient inter-core communication and face the same programming challenges as clusters, with the main difference being lower communication latency. A runtime system that exposes additional parallelization opportunities adds great value to these platforms that lack shared memory.

Figure 1.6 summarizes the motivation of this dissertation. This dissertation aims to improve the applicability and scalability of parallelization technology by supporting the widest variety of parallelization techniques while making the fewest assumptions about the underlying hardware.

1.3 Contributions

To support MTXs on commodity clusters enabling scalable automatic parallelization, this dissertation proposes the Automatic Speculative Acyclic Parallelization (ASAP) system that consists of a speculative acyclic parallelizing compiler and a transaction runtime system that supports MTXs. The ASAP compiler automatically identifies parallelizable loops in a program via dynamic profiling runs and static dependence analysis at compile-time. Speculation enabled by profiling and static analysis complement each other; speculation allows the compiler to overcome the limitation of conservative static analysis while static analysis reduces speculatively removed dependences and their validation overheads in the transaction runtime system. The ASAP compiler provides different speculation techniques that have low validation overheads, and transforms the sequential loops to Spec-DSWP codes with optimized communication among threads.

The ASAP runtime system executes the Spec-DSWP codes on clusters without any hardware modification. Like single-threaded transaction runtime systems, the ASAP runtime system allows threads to speculatively execute iterations in a loop in advance, and commit the speculative execution if they are well-speculated, or rollback the execution if misspeculated. To support MTXs beyond single-threaded transactions, the ASAP runtime system should provide two additional features; group transaction commit and uncommitted value forwarding. Since a transaction is decomposed into multiple sub-transactions across multiple threads in Spec-DSWP scheme, the sub-transactions must be committed together (group transaction commit). In addition, since the sub-transactions execute multiple pipeline stages on different threads as a transaction, updated values from a stage should be forwarded to later stages on other threads (uncommitted value forwarding). With these two additional features, the ASAP runtime system can support Spec-DSWP schemes.

With synergistic combination of speculation, acyclic communication, and runtime system support, the ASAP system enables scalable performance improvement for a wide range of applications on clusters without any hardware modification. The ASAP system automatically parallelizes 17 sequential C programs for a 12-core 10-node (120 total core) cluster without any annotation about parallelization, and achieves a geomean speedup of 8.91× with 109.5× maximum performance improvement. In addition, this dissertation manually parallelizes 11 sequential C programs with the ASAP system, and demonstrates a path to achieve scalable performance improvement for general-purpose applications.

This dissertation proposes the first fully automatic speculative acyclic parallelization (ASAP) system for commodity clusters. The contributions of the dissertation are:

- The first fully automatic Spec-DSWP compiler for commodity clusters. The compiler supports new speculation techniques that have low communication and validation overheads, and automatically parallelizes sequential loops using the Spec-DSWP scheme with the optimal application of the speculation techniques. The compiler inserts and optimizes communication between pipeline stages

- The first transaction runtime system that supports MTXs on clusters. The runtime system supports group transaction commit and uncommitted value forwarding to execute Spec-DSWP codes. The runtime system efficiently manages inter-process communication and distributed memory accesses, thus enabling scalable performance for clusters

- Synergistic combination of the ASAP compiler and the ASAP runtime system. The co-design of the compiler and the runtime system enables various optimization techniques that a stand-alone compiler or a runtime system cannot support, and realizes scalable performance improvement for clusters with speculation techniques that have low validation overheads.

1.4 Dissertation Organization

The dissertation introduces the ASAP system that speculatively parallelizes programs with acyclic communication on clusters. Chapter 2 surveys related research of the ASAP system. Chapter 3 illustrates the overall architecture and the execution model of the proposed system. Chapter 4, in which automatic Spec-DOALL parallelization for clusters was published in [40], describes speculation techniques and implementation of the ASAP compiler. Chapter 5, published in [41], describes detail implementation of the ASAP runtime system. Chapter 6 evaluates the ASAP system on a cluster and analyzes the results. Chapter 7 discusses the future work of this dissertation and summarizes the conclusion of this work.

Chapter 2

Related Work

The ASAP system consists of the ASAP compiler and the ASAP runtime. This chapter describes related research of each component.

2.1 ASAP Compiler

Automatic Parallelization: Research about automatic parallelization has a long history. Boulet et al. [15] surveys various parallelization algorithms [3, 25, 27, 28, 44, 48, 84] and code generation techniques [7, 16, 22, 23, 37, 39, 46, 63]. The rest of this section presents recent representative examples of automatic parallelizing compilers. Table 2.1 compares this work with the automatic parallelization systems.

Rus et al. proposes Hybrid Analysis (HA), which exploits runtime support for dependence analysis in statically indeterminate cases [70]. Although their system potentially improves the applicability of automatic parallelization, heavyweight run-time analysis can significantly slow down program execution because there is no overlap between the analysis and the execution phases.

Campanoni et al. proposes HELIX, a new automatic loop parallelization technique that assigns successive iterations of a loop to separate threads [19]. HELIX reduces communication overheads with signal prefetching and code balancing, and achieves stable performance speedups without any slowdown across thirteen C benchmarks from SPEC CPU2000 [73] with a loop selection algorithm and a speedup model. However, its scalability is limited to single shared memory machine, and its applicability is limited due to lack of speculation support.

Automatic Speculative Parallelization: Traditional loop parallelization schemes such as DOALL and DOACROSS rely on regular structure in a program [4]. These schemes perform well for scientific programs but are less profitable for general-purpose applications, where irregular control flow and data access patterns are the norm. Some speculative parallelization schemes, loosely classified as Thread-Level Speculation (TLS), have been developed to mitigate this inherent irregularity [10, 68, 76, 89, 90].

There are research compilers that parallelize applications using speculation [29, 49, 52, 64, 89]. However, these compilers assume the availability of specialized hardware or cache-coherent shared memory, and their performance is evaluated using a small number of cores (typically fewer than 32). Software transactional memory systems have suffered from large validation overhead [20]; consequently they may not scale to the large number of cores. To achieve scalable performance on a large number of cores, it is crucial to optimize communication because the commit bandwidth easily becomes a performance bottleneck.

For example, the POSH compiler [49] is capable of automatically parallelizing complex, general-purpose programs, but it requires TLS hardware support; hence it cannot be used on a commodity machines. STMlite [52, 89] is a speculative parallelization system that consists of an automatic parallelizing compiler and a low-cost software transactional runtime. Although STMlite can execute programs on real hardware, it is implemented and evaluated on small-scale shared-memory machines with 8 cores, and its scalability with a large number of cores has not been demonstrated. The ASAP system has fewer assumptions on the target system showing better scalability.

Automatic Parallelization for Clusters: Intel's Cluster OpenMP [34] extends OpenMP, a parallel programming API for shared-memory multiprocessors, to clusters with distributed

System	Fully Automatic	Applying Speculation	Comm. b/w Threads	Commodity Hardware	Commodity Clusters	Evaluated # of Cores
Polaris [12, 70]	Yes	No	No	Yes	No	8, 16
HELIX [19]	Yes	No	Yes	Yes	No	6
POSH [49]	Yes	Yes	Yes	No	No	4
STMlite [52, 89]	Yes	Yes	No	Yes	No	8
OpenMP [1, 34]	No	No	No	Yes	Yes	-
SUIF [5]	Yes	No	No	Yes	Yes	32
Cluster Spec-DOALL [40]	Yes	Yes	No	Yes	Yes	120
ASAP [This work]	Yes	Yes	Yes	Yes	Yes	120

Table 2.1: Comparison of automatic parallelization systems

memory systems. Although the Cluster OpenMP compiler transforms sequential programs to parallel codes automatically, programmers are still required to specify what and how to parallelize them with programmer annotations.

SUIF [5, 74] parallelizes a sequential program without any programmer annotation for clusters. However, the applicability of SUIF is limited to array-based scientific applications, and SUIF relies on programmer hints to decompose shared data across multiple nodes on a cluster [5]. In contrast, the ASAP system does not require any programmer annotation for shared data since the ASAP runtime handles this via copy-on-access and unified virtual address space, effectively hiding platform details. The SUIF project proposes automatic speculative parallelization [57, 58], but the system requires special hardware systems not supporting commodity clusters.

Cluster Spec-DOALL [40] speculatively parallelizes a sequential program without any programmer annotation for clusters that have more than 100 cores. However, the compiler does not allow communication between threads, so its applicability is limited.

2.2 ASAP **Runtime System**

Thread-Level Speculation (TLS): Runtime support for speculative parallelism has been an active area of research, and there are a number of proposals including Transactional Memories (TM) and TLS memory systems. The runtime systems track every speculative memory operation within a transaction or task (i.e., region of code executed speculatively) to determine if any atomicity violation (in TM) or dependence violation (in TLS) occurs at commit time. Proposals for TM or TLS memory systems can be divided into two classes: hardware-based approaches [32, 53, 77, 85, 88] and software-only approaches [13, 24, 41, 42, 51, 52, 56, 66, 80]. Hardware-based approaches depend on special hardware to buffer speculative state, detect misspeculation and recover from it. In contrast, the ASAP runtime system is a software-only system that enables speculative parallelization on commodity clusters. Software-only approaches can be further divided depending on whether they require cache-coherent shared memory or not. Most existing proposals for software-only speculative runtime systems target only small-scale shared-memory computers with tens of cores at most [52, 56, 66, 80].

Speculative Pipelining: Speculative pipelining schemes [17, 36, 79, 80, 82] have many benefits over TLS. The execution model presented in this dissertation is inspired by the success of similar models described in [17, 36, 79, 82]. Although they speculatively remove dependences, create the pipeline structure, and extract DOALL-style parallelism in some stages of the pipeline, they cannot be executed on most existing TM and TLS memory systems because transactions (loop iterations) are split across multiple threads. This work provides the runtime system that supports the speculative pipelining.

Tian et al. [80] proposed a Copy-Or-Discard (CorD) execution model for speculative parallelization. They achieve excellent speedup (3.7× to 7.8× on 8 cores) on six benchmarks. CorD does not support MTXs. CorD's execution model is such that all the worker threads and the main thread are synchronized on every iteration, putting the cross-thread communication latency on the critical path. CorD uses a multi-threaded approach that partitions the virtual address space. This penalizes every load and store operation by necessitating a table lookup to determine whether the object being accessed exists in the speculative worker's memory partition. By maintaining the same virtual address space across all workers, the ASAP runtime system can guarantee that the load/store addresses will be valid across all of them.

Vachharajani et al. and Raman et al. proposed hardware MTX system (HMTX) and software MTX systems (SMTX) to support Spec-DSWP execution [66, 81, 82]. Although the MTX systems can execute parallelized programs with the Spec-DSWP scheme, they cannot run on commodity clusters because they require either hardware modification or cache-coherent shared memory. The ASAP system inserts and optimizes communication codes for shared data, thus enabling Spec-DSWP codes to be executed on clusters. In addition, clusters have higher communication latency and lower communication bandwidth than the MTX hardware system and cache-coherent shared memory systems, so naïve application of speculation causes high communication and validation overheads. The ASAP system supports additional speculation techniques that have low validation and communication overheads to achieve scalable performance improvement on clusters.

Speculative Runtime Systems for Clusters: There have been proposals for TM and TLS memory systems on clusters [13, 24, 31, 41, 42, 51, 71, 72], but only Cluster-STM [13], DSMTX [41] and Snake-DSTM [71, 72] have demonstrated their scalability on platforms with over 100 cores. None of these systems except DSMTX implements MTX semantics, so they cannot execute Spec-DSWP codes. In addition, among the proposals, there is no known automatic speculative parallelizing compiler targeting them. The ASAP system is the first fully-automatic speculative parallelization system that supports Spec-DSWP and scales to hundreds of cores without requiring hardware support or cache-coherent shared memory.

Herlihy and Sun discuss a DSTM design based on global cache coherence [33], and Zhang and Ravindran propose a location-aware distributed cache-coherence protocol [86, 87]. Their works are largely theoretical and lacks evaluation with a concrete implementation.

TM^{2}C [31] is a distributed transactional memory system for a single machine with non-coherent many-core processors. The system creates transactional memory threads that grant a data access through the distributed locking to each application thread. The system achieves the performance improvement on the many-core system, but it may not be scalable enough to support clusters beyond single machine because the system relies on low network-on-chip communication latency.

D^{2}STM [24] uses the Bloom Filter Certification (BFC) to reduce validation overheads in transactional memory, but with an increase in the probability of transaction abort. The ASAP runtime system reduces the validation overheads by keeping a separate validation thread and comparing speculatively read and write values, so the ASAP runtime system does not suffer from false-positive detection.

DiSTM [42] is a DSTM system which builds on Java Remote Method Invocation (RMI). DiSTM detects and resolves conflicts at object granularity. In DiSTM, the main node keeps the committed state of a program, and worker nodes execute transactions using private cached data. The ASAP runtime system has a commit unit that keeps committed data, and workers which execute MTXs in their private physical memories. Although DiSTM allows parallel commits using the multiple leases protocol, the workers are tightly coupled through the validation/commit process because a worker cannot start the next transaction until the current transaction commits. By contrast, The ASAP runtime system decouples transaction execution from validation/commit, allowing a worker to start a new transaction before the commit process of the current one finishes.

Distributed Multiversioning (DMV) [51] modifies a software distributed shared memory system (SDSM) to support transactions. DMV and the ASAP runtime system expose a unified virtual address space to programs. DMV performs transaction validation and commit at the page granularity by means of page diffing. This static batching of spatially adjacent words may result in unnecessary diffs and excessive communication for memory access patterns that access pages in a sparse fashion. The ASAP runtime system eliminates this problem by performing transactional operations at the word granularity and batches up the words according to dynamic access patterns.

Nested Parallelism in Transactional Memory: Like the ASAP runtime system, some nested transactional memory [2, 9, 65, 83] can support nested speculative parallelism allowing multiple threads in a transaction. In the nested transactional memory systems, write sets in a sub-transaction are either updated directly to main memory (open nested transaction) or merged to their parent transaction (closed nested transaction). However, in the ASAP runtime system, transactions are ordered, and the write sets are not only forwarded to following sub-transactions in the same parent transaction in different threads, but also optimistically forwarded to following sub-transactions in the same stage in different parent transactions without updating main memory. Unlike closed nested transactions, the ASAP runtime system allows that uncommitted values can be viewed by other threads and transactions. This allows the ASAP runtime system to exploit pipeline parallelism with sequential stages, which would be otherwise hidden. Programmers can locate frequently occurring dependences into sequential stages to deliver partially updated values to the following sub-transactions in the next iterations before commit. Unlike open nested transactions, the ASAP runtime system can safely rollback all the misspeculated updates in sub-transactions without causing any correctness problem because the write sets are not updated to main memory until the parent transaction is committed.

Partitioned Global Address Space (PGAS) and Software Distributed Shared Memory (SDSM): The ASAP runtime system is influenced by PGAS [21, 26, 50] and SDSM [6]. Like PGAS and SDSM, the runtime system provides a unified virtual address space to all threads. Like PGAS, the runtime system partitions the address space into several non-overlapping regions each of which is associated with a different worker. However, while PGAS is a language-based approach to concurrency in distributed systems, the runtime system is a library-based approach which does not require code modifications. Compared to SDSM, the runtime system is customized and optimized to support speculative parallel execution by selectively supporting coherence through Copy-On-Access.

Chapter 3

Overview of ASAP

The ASAP system consists of a parallelizing compiler and a runtime system. The compiler profiles a sequential program with a set of profilers, and parallelizes loops in the program using the profiling results and dependence analysis. The runtime system executes the parallelized loops safely and efficiently on clusters.

3.1 ASAP Compiler

The compiler takes sequential C/C++ source codes as input to generate parallelized codes targeting the ASAP runtime system. As Figure 3.1 illustrates, the compiler framework is composed of the following components: a set of profilers, a parallelization strategy manager with various dependence analyzers, a speculation manager, a communication optimizer, and a code optimizer. The rest of this section briefly explains the functionality of each component, which will be discussed in more detail in Chapter 4.

Profilers: The profilers gather dynamic information by executing the sequential code with training input sets. More specifically, the ASAP compiler uses four profiles: control flow profile, loop aware memory dependence profile, loop profile, and speculative privatization profile. Dependence analyzers in the parallelization strategy manager use these profiling results to apply various speculation techniques.

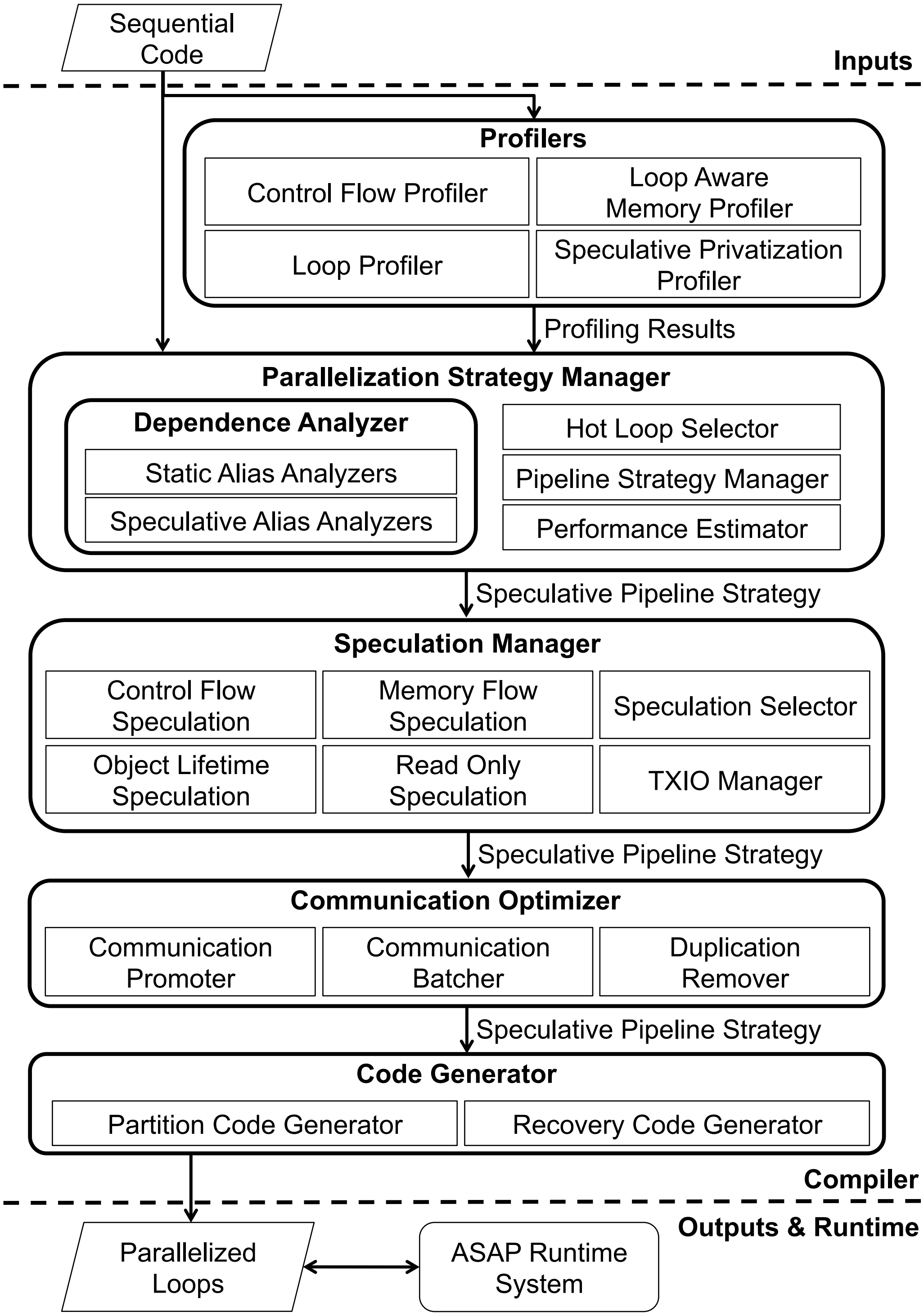


Figure 3.1: Overall ASAP system

Parallelization Strategy Manager: The parallelization strategy manager generates speculative pipeline strategy with the dependence analyzer. The dependence analyzer creates a speculative program dependence graph (Spec-PDG) via static and speculative alias analysis, which includes both data and control dependences. The parallelization strategy manager controls speculative alias analyzers either to speculatively remove or to respect inter-iteration dependences depending on target speculation hit ratio. If there is no inter-iteration dependence without speculation in the Spec-PDG, the strategy manager makes classical DOALL parallelization strategy. If the Spec-PDG has speculated dependence, the strategy is speculative DOALL parallelization. If the Spec-PDG has an inter-iteration dependence, the manager plans DSWP or Spec-DSWP parallelization for the target loop.

Speculation Manager: The speculation manager finds appropriate speculation for each speculated dependence to reduce validation overheads. There can be various speculation techniques to remove an inter-iteration dependence, which have different validation costs. The speculation manager chooses the most efficient speculation technique for each speculated dependence in the Spec-PDG. If a speculated dependence is in the same stage or does not affect the parallelization strategy due to other speculation, the manager removes the unnecessary speculation because the speculation increases validation overheads.

Communication Optimizer: Clusters have high communication latency, so communication optimization is important to achieve scalable performance improvement. The communication optimizer promotes and batches explicit message passing operations for live-out and validation, and hoists them out from inner loops.

Code Generator: The code generator transforms the sequential loops to the speculative parallel loops gathering all information; pipeline strategy, applied speculation techniques, and optimized communication operations. The code generator makes partitions according to the pipeline strategy, and inserts runtime function calls to validate speculated dependences and to deliver messages between partitions. The code generator also generates recovery codes to re-execute a misspeculated iteration, which is invoked after rollback.

3.2 ASAP Runtime

The parallelized loops, either with speculation or not, are executed on the ASAP runtime system, which supports speculative memory accesses, misspeculation detection and recovery, live-in and live-out handling, and process management. As a transaction runtime system, the ASAP runtime system validates memory speculation, and manages rollbacks in the event of misspeculation. Unlike other transaction runtime systems, the ASAP runtime system needs to support not only Spec-DOALL and TLS, but also Spec-DSWP. The ASAP runtime system provides two additional features; group transaction commit and uncommitted value forwarding to support Spec-DSWP, in which a loop iteration (a transaction) is partitioned into multiple partitions as sub-transactions (subTXs) forming a MTX [81].

Group Transaction Commit: All the subTXs within an MTX must commit together. The subTXs originally belong to a transaction that is one atomic unit, so they should be managed as one atomic unit. While each subTX should have its own private memory space for speculative execution, all the updates should be committed together and merged as one committed memory version.

Uncommitted Value Forwarding: Speculative stores in an earlier subTX must be visible in a later subTX within the same MTX. The runtime system should allow a later subTX to load a stored value in an earlier subTX because the subTXs are partitioned from the same loop iteration. Therefore, the value needs to be passed to the later subTX although the value is not committed yet. This allows the subTXs to be synchronized, and reduces misspeculation ratio by respecting some dependences across multiple subTXs.

3.2.1 The Execution Model

In order to support the simultaneous execution of multiple MTXs, the ASAP runtime system creates worker processes that access to different physical memory spaces. To take the overhead of speculation management off the critical path, the ASAP runtime system creates separate commit units such as a “validator” that is responsible for validating transactions and a “master” that is responsible for committing transactions. These also execute in different physical memory spaces. The memory updates by a worker in an MTX are forwarded to other workers that participate in the same MTX via communication channels. Combined, the physical memories of the workers and the commit units, and the communication channel buffers allow each MTX to have the illusion of a private memory, and also allow multiple MTXs to be outstanding in the system.

Figure 3.2 illustrates the life cycle of an MTX in the context of loop parallelization, from initialization to commit.

MTX Initialization: The first MTX in the program is initialized with the non-speculative memory state. This state is generated by the sequential, non-transactional code prior to the parallel section. One option is to have each worker generate the state by redundant execution of the sequential code. However, this may result in replicated side effects. Another option is that only the master executes the sequential, non-transactional code to generate the initial non-speculative memory state. Each worker initializes its memory state referring the memory of the master.

MTX Execution: All speculative loads and stores in an MTX happen in the private memories of the workers. Stores by an earlier subTX in an MTX must be visible to loads in a later subTX so that there is no intra-MTX misspeculation. Since subTXs are executed by different worker processes in different memories, each worker must forward its speculative stores to workers executing later subTXs. This *uncommitted value forwarding* happens via the communication channels between processes. Since *only those processes that participate in the same MTX are connected,* the number of communication channels in the system does not grow quadratically in the number of processes. As Figure 3.2 shows (communication boxes in workers), the uncommitted values are explicitly forwarded at the end of a subTX. A later subTX refreshes its memory with the uncommitted values before commencing execution.

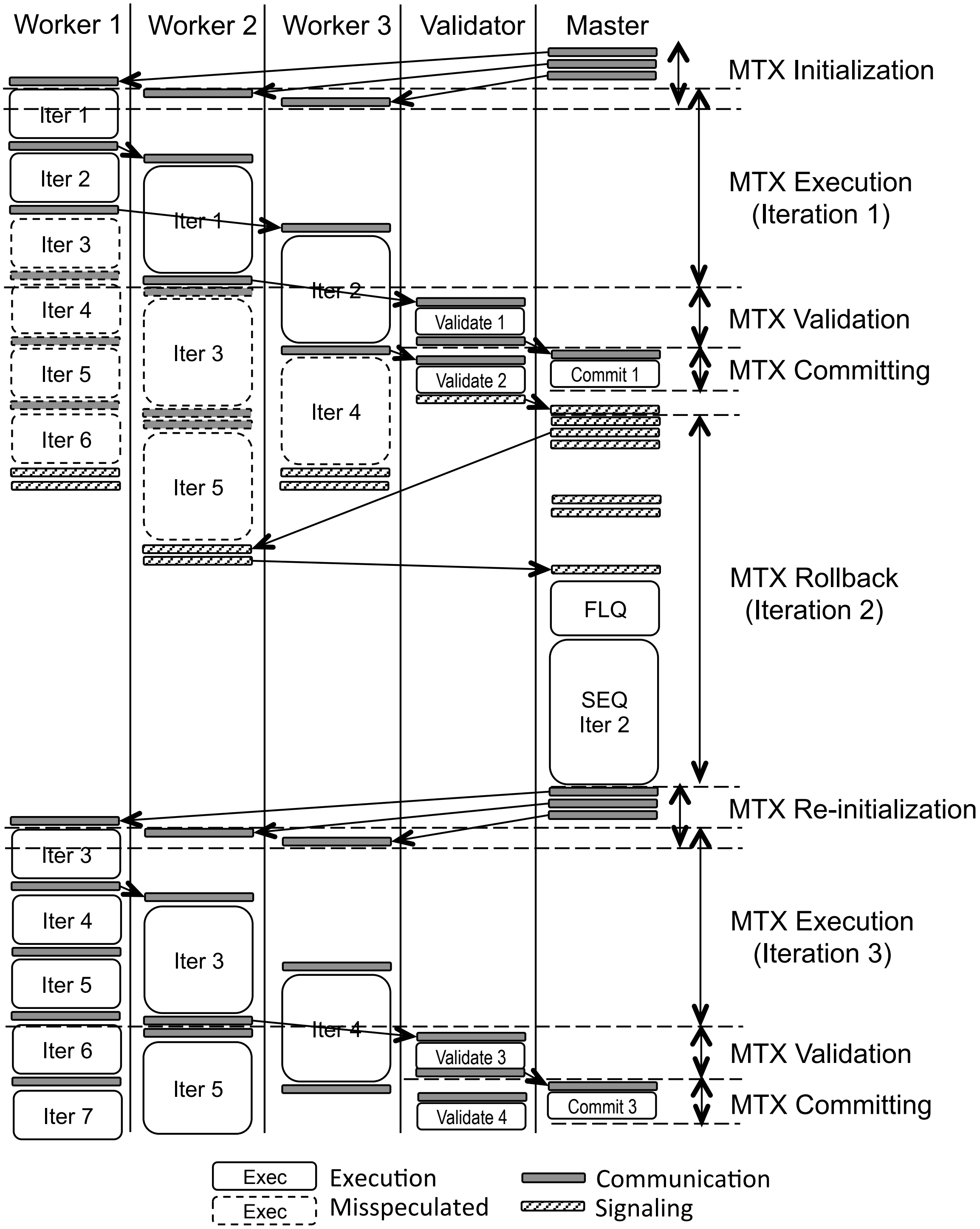


Figure 3.2: MTX execution model

The communication channels also serve to decouple the workers and the commit units. Workers can simultaneously execute subTXs of different MTXs without waiting for validation and commit of the prior MTX. Figure 3.2 illustrates the decoupled execution. Worker 1 is executing a subTX of MTX_5, while worker 2 is executing a subTX of MTX_3, the validator is validating MTX_2, and the master is still committing MTX_1.

MTX Validation: An MTX is deemed to be free of conflict by means of a unified value prediction and checking mechanism. For control dependences, misspeculation is detected if the predicted value of the branch condition does not match the actual value at run-time. False memory dependences are automatically broken by means of memory versioning, so there is no need to check for their manifestation. A true memory dependence between a load and a store operation is checked by comparing the speculatively loaded value (predicted value) with the actual value stored by the store operation when that store is ready to be committed. This check is done by the validator. In all misspeculation cases, a signal is sent to the master which orchestrates recovery.

MTX Committing: After all the subTXs in an MTX are deemed to be free of conflicts, the master commits the entire MTX atomically. Through the same mechanism as uncommitted value forwarding, all stores in subTXs are forwarded from the workers to the master. The master commits the subTXs in a transaction by updating its memory with the forwarded values. The updates are done in order of subTX (which is the program order); if a memory location is updated in two different subTXs, the last update takes effect. This is how the MTX runtime system supports *group transaction commit*. Reiterating, since the master's operations are decoupled from the worker processes by means of the communication channels, the overhead of commit does not impact the workers' execution.

MTX Rollback: When an MTX is detected to conflict with an earlier MTX, it must be re-executed. Figure 3.2 illustrates how the ASAP runtime system recovers misspeculated states. First, in the event of a conflict between MTXs, the master signals the workers to restart the logically later MTX. After receiving the misspeculation signal, each process goes into recovery mode and hits a barrier to ensure that the others have also entered recovery mode. (Signaling in Figure 3.2) Second, each process flushes communication channels than contains speculative state, and hit a barrier again. (FLQ in Figure 3.2) Third, all the processes but the master discards the speculative state in their private memory spaces. (Dashed boxes in Figure 3.2) The master executes the loop iteration corresponding to the aborted MTX in single-threaded fashion. (SEQ in Figure 3.2) During the execution of this iteration, the master may produce data via the communication channels to the workers; this explains why the barrier in step two is necessary. Finally, all the processes hit a barrier to ensure that parallel execution may recommence. The MTX runtime system is reinitialized with the committed memory state as before, and speculative execution resumes.

3.2.2 The Runtime System Structure

The ASAP runtime system mainly consists of three components; MTX manager, communication manager, and memory manager. This section briefly introduces each component. Detail description is presented in Chapter 5.

MTX manager: The MTX manager controls the life cycle of an MTX for each role, from creation to commit as Figure 3.2 shows. The manager allows the workers to speculatively execute programs in advance, the validator to check if the speculative execution is correct, and the master to update correctly speculated results on its memory. The manager recovers the speculative execution if misspeculation occurs.

Communication manager: The communication manager controls all kinds of communication between processes such as message passing, synchronizing processes and signaling. The manager is implemented on top of MPI system [54]. Instead of directly using MPI_Send and MPI_Recv, the manager batches data to transfer in local queues, and amortizes the communication overheads.

Memory Manager: The memory manager handles memory related operations. First, it provides the unified virtual address (UVA) space to each process on distributed memory systems. The UVA space allows data communication without address translation. Second, the manager supports copy-on-access (COA) mechanism that initializes a memory page only which a worker process accesses at run-time, to reduce communication overheads from memory initialization. For the first page access, the master copies the page from the master to the worker. Third, the memory manager assists the MTX manager to validate memory speculation and recovers misspeculated execution. The memory manager provides shadow memory for speculative memory accesses that removes duplicated validation. The MTX manager exploits the shadow memory, thus reducing validation overheads. Finally, the manager tracks memory writes of the master process in the non-parallel region, and checks dirty pages. At the next parallel execution, the memory manager initializes only the dirty pages to reduce memory initialization overheads especially for nested loops.

Chapter 4

ASAP **Compiler**

The ASAP compiler automatically identifies parallelizable loops in sequential C/C++ programs, and transforms the loop to parallel codes with DOALL, Spec-DOALL, DSWP, and Spec-DSWP parallelization scheme. The thesis implements the ASAP compiler on top of Liberty compiler [78] and LLVM infrastructure [45]. The compiler is composed of the following components: a set of profilers, a parallelization strategy manager that includes various dependence analyzers, a speculation manager, a communication optimizer, and a code optimizer. Among the components, this dissertation contributes parts of speculative alias analyzers in parallelization strategy, speculation manager, communication optimizer, and code generator. The dissertation uses the existing profilers and parallelization strategy of the Liberty compiler with small modification. This section describes each component in detail.

4.1 Profilers

The profilers execute the original sequential program with training input sets, and gathers various dynamic information. The ASAP compiler uses four profilers in the Liberty compiler infrastructure; **control flow profiler** that collects the traversal count of every edge in the control flow graph, **loop aware memory profiler** that observes the flow of values from stores to loads, **loop profiler** that measures the execution time of each loop, and **speculative privatization profiler** that tracks memory allocation and its usage.

4.2 Parallelization Strategy Manager

The parallelization strategy manager makes speculative pipeline strategy such as DOALL, Spec-DOALL, DSWP, and Spec-DSWP for each loop based on profiling information and speculative program dependence graph (Spec-PDG) from the dependence analyzer. The ASAP compiler uses the parallelization strategy manager of the Liberty compiler infrastructure with additional speculative alias analyzers such as object lifetime analyzer, read only analyzer, and TXIO analyzer that this thesis newly implements. The parallelization strategy manager generates the strategy with four components; hot loop selector, dependence analyzer, pipeline strategy manager, and performance estimator.

The **hot loop selector** finds hot loops that execute more than the minimum number of iterations per invocation. The iteration number per invocation is important for the ASAP compiler to generate a profitable parallel loop because a hot loop may suffer from huge invocation overheads more than parallelism benefits if the loop is invoked many times but iterates a small number of times per invocation. The hot loop selector relies on information about loop execution time and iteration counts per invocation that the loop profiler and the control flow profiler generate respectively.

The **dependence analyzer** creates Spec-PDG for the hot loops via static and speculative alias analysis. Since the ASAP runtime system provides private memory spaces for each worker process, the dependence analyzer ignores loop-carried anti- and output- dependences in Spec-PDG. To support various speculation techniques, the ASAP compiler uses three additional speculative analyzers beyond the existing analyzers such as object lifetime analyzer for object lifetime speculation, read only analyzer for read-only speculation, and TXIO analyzer for I/O operations in transactions.

The **pipeline strategy manager** generates pipeline plans for the hot loops using the Spec-PDG. The pipeline strategy manager makes strongly connected components (SCCs) from the Spec-PDG, and makes partitions for each loop [59, 82]. The SCC is a set of instructions that participate in a dependence cycle of the Spec-PDG. A SCC is replicable if all the instructions in the SCC do not have any side effect. If all the SCCs in a loop are replicable, the pipeline strategy manager creates either DOALL or Spec-DOALL strategy for the loop. If not, the manager allocates non-replicable SCCs to sequential stages, thus making DSWP or Spec-DSWP strategy.

Finally, the **performance estimator** decides whether the parallelization strategy is profitable or not. For example, if a two-stage pipeline strategy has a large sequential stage with a small parallel stage, the pipeline strategy may not be scalable due to the large sequential stage, or sometimes suffer from slowdown due to the communication costs. The performance estimator removes the non-profitable parallelization strategies.

4.3 Speculation Manager

The speculation manager makes the optimal speculation plan for the speculative pipeline strategy, and updates the strategy to be correctly validated and recovered without side effects. The ASAP system provides the four types of compile-time speculation techniques:

- **Control Flow Speculation:** The control flow profiler collects the traversal count of every edge in the control flow graph. For each control flow edge, the profiler computes the ratio of the number of taken times to total loop iterations. When the ratio is smaller than a static threshold, the control flow speculation alias analyzer marks the control flow edge as *speculated*, and all basic blocks that are dominated by the edge as *speculatively dead*. Control speculation does not require any internode communication except for misspeculation recovery, so the speculation manager preferentially applies control speculation over the other forms of speculation.

- **Memory Flow Speculation:** Memory flow speculation relies on the loop aware memory profiler which observes the flow of values from stores to loads. This information is stronger than alias information because two memory operations may alias even when there is no flow between the operations. The memory flow speculation alias analyzer identifies loop-carried memory flow dependences which occur less frequently than a static threshold, and marks them as *speculated*. No speculation is applied to intra-iteration memory flow dependences because such dependences do not affect parallelism applicability. Inter-node communication must be inserted to detect a memory flow misspeculation at run-time, so memory flow speculation has higher overhead than other speculation. To reduce validation overheads, the speculation manager avoids applying the memory flow speculation if another speculation technique is available.

- **Object Lifetime Speculation:** Object lifetime speculation is guided by the speculative privatization profiler to identify dynamic objects that are private to a single loop iteration. The profiler reports allocation sites whose object is not freed in the same iteration of a loop, and deallocation sites whose object is not allocated in the same iteration. Since updates to iteration-private objects are independent across iterations, the object lifetime speculation alias analyzer removes inter-iteration dependences on the private objects. In addition, the updates on the private objects are not live-out of the loop, so the speculation manager marks memory writes on private objects as non-live-out, and reduces the amount of inter-node communication. Specialized versions of `malloc` and `free` automatically test for misspeculation without inter-node communication.

- **Read Only Speculation:** The speculative privatization profiler also provides memory access patterns for a memory allocation unit. If a memory allocation unit is not written at all in a loop, the read only speculation can assume that the allocated memory is not changed in the loop, and speculatively remove all the memory dependences from and to the memory space. Speculation can be validated without inter-node communication by tracking all the memory write addresses.

Speculation removes inter-iteration dependences that are unlikely to manifest at runtime, to increase parallelism opportunities. Since the dependences are optimistically removed without static proof, the ASAP runtime system should validate the speculated dependences at run-time for correct execution, and the validation operation affects the overall performance. An inter-iteration dependence can be removed with multiple speculation techniques that have different validation costs. Among the available speculation techniques, the **speculation selector** chooses the most efficient speculation technique to minimize the validation overheads, and makes the optimal speculation plan. For example, since validation of the memory flow speculation is the most expensive among the four available speculation techniques, the selector chooses other speculation techniques if applicable. Since control flow speculation can speculatively remove all the instructions that are dominated by speculated control flow edge, the selector priorly applies control flow speculation.

It addition, the selector removes unnecessary speculation. For example, although a memory inter-iteration dependence between two instructions is speculatively removed, the two instructions may be located in the same sequential stage due to other dependences. If a speculation does not affect parallelization structure, the speculation selector removes the dependences from speculation sets to reduce validation overheads.

The **TXIO manager** enables speculation on loops that may have side-effective instructions like I/O operations. If a loop has an instruction with side effects, it is difficult to apply speculation to the loop because the side effects may not be recovered. The TXIO manager resolves the problem and allows the ASAP compiler to speculatively parallelize the loop, delaying the execution of the side effective instructions after commits.

4.4 Communication Optimizer

When scaling the ASAP system to a large number of cores, the limited communication bandwidth becomes a bottleneck of the whole system. Minimizing the number of messages and volume of communication between processes is crucial for scalable performance. In many programs, memory operations within inner loops of a parallelized loop claim the largest portion of the communication bandwidth to handle live-out and speculative memory accesses. To reduce the amount of communication generated by the inner loops, the communication optimizer performs three optimizations: promotion, batching, and duplication reduction.

When validating a memory access in a speculative iteration, the validator calls `specStore` function to reflect the memory update to the validator's memory version, and `specLoad` function to check if the memory access reads the correct version. Within a single transaction, multiple accesses to the same address cause redundant communication; only the first load from and the last store to the address affect the validation result. Exploiting this feature, the communication promoter hoists function calls to `specStore` and `specLoad` out of the inner loop to the loop preheader and loop exits respectively, if the target memory address is loop-invariant. Unlike conventional store/load promotion, this optimization is insensitive to the existence of other instructions that may overwrite or reload the same address, because only the first load and the last store within a transaction matter for validation. Similarly, calls to `produce` for live-outs can be moved to the inner loop's exits.

The service bandwidth is limited not only by the communication volume in bytes but also by the number of messages. Batching is optimization which gathers dense reads from or writes to a chunk of memory into a single jumbo read or write. The communication batcher is applicable to memory operations in a counted inner loop whose pointer operands are induction variables of the inner loop. If applicable, the batcher removes the calls to `specStore` (`specLoad`) from the inner loop, and places the calls to `specStoreRange` (`specLoadRange`) after (before) the inner loop. In this way, the ASAP runtime system

```
for(i=0; i<ni; i++) {     // Loop i
 for(j=0; j<nj; j++) {    // Loop j
  specLoad(&C[i][j]);
  C[i][j] *= beta;
  specStore(&C[i][j]);
  for(k=0; k<nk; ++k) {  // Loop k
   specLoad(&C[i][j]);
   C[i][j] += alpha*A[i][k]*B[k][j];
   specStore(&C[i][j]);
  }
 }
}
```

(a) Before Optimization

```
for(i=0; i<ni; i++) {     // Loop i
 for(j=0; j<nj; j++) {    // Loop j
  specLoad(&C[i][j]);
  C[i][j] *= beta;
  specStore(&C[i][j]);
  specLoad(&C[i][j]);
  for(k=0; k<nk; ++k) {  // Loop k
   C[i][j] += alpha*A[i][k]*B[k][j];
  }
  specStore(&C[i][j]);
 }
}
```

(b) After Promotion

```
for(i=0; i<ni; i++) {     // Loop i
 specLoadRange(&C[i][0], nj);
 specLoadRange(&C[i][0], nj);
 for(j=0; j<nj; j++) {    // Loop j
  C[i][j] *= beta;
  for(k=0; k<nk; ++k) {  // Loop k
   C[i][j] += alpha*A[i][k]*B[k][j];
  }
 }
 specStoreRange(&C[i][0], nj);
 specStoreRange(&C[i][0], nj);
}
```

(c) After Batching

```
for(i=0; i<ni; i++) {     // Loop i
 specLoadRange(&C[i][0], nj);
 for(j=0; j<nj; j++) {    // Loop j
  C[i][j] *= beta;
  for(k=0; k<nk; ++k) {  // Loop k
   C[i][j] += alpha*A[i][k]*B[k][j];
  }
 }
 specStoreRange(&C[i][0], nj);
}
```

(d) After Removing Duplication

Figure 4.1: An example of communication optimization for benchmark `gemm`

can deliver the same number of bytes in fewer messages. A batched function call may be further promoted higher in a loop nest in a way analogous to the promotion of a speculative load or store. In other words, batching not only reduces the number of messages but also exposes hidden opportunities for communication optimization by transforming loop-variant `specStore` and `specLoad` into loop-invariant `specStoreRange` and `specLoadRange`.

The compiler inserts validation function calls to `specStore` and `specLoad` for all the speculative memory instructions. If multiple instructions access the same memory address, the same values are delivered multiple times, thus increasing communication overheads without affecting the validation results. If one of the instructions dominates the others, the duplication remover erases the duplicated function calls for the same memory address. The duplication reduction is similar to promotion but different. The duplication

```
Loop_A:
// Master Process
beginInvocation(DOALL);
produceLiveIns();
consumeLiveOuts();
endInvocation();
```

```
// Worker Process
beginInvocation(DOALL);
consumeLiveIns();
for(int i=0; i<N; i++) {
  if(i%NP==tid) {
    regular[i]+=foo(i);
    produce(&regular[i]);
} }
endInvocation();
```

Figure 4.2: Code generation: DOALL (Loop_A)

reduction erases duplicated static instructions, but the promotion moves a static instruction of an inner loop to the outer loop, and reduces duplicated dynamic instructions. The duplication reduction removes batched or promoted function calls.

Figure 4.1 shows a real example about how the communication optimizer optimizes communication in benchmark `gemm` through promotion, batching, and duplication reduction. Figure 4.1 (a) is the original codes before communication optimization. The dependence analyzers cannot statically remove inter-iteration dependences on memory access `C[i][j]`, so they remove the dependences with memory flow speculation. The speculation manager inserted `specLoad` and `specStore` in `Loop j` and `Loop k` right before loading and after storing `C[i][j]`. First, the communication promoter promotes `specLoad` and `specStore` in `Loop k` to `Loop j` because `C[i][j]` is loop-invariant to the induction variable `k` (Figure 4.1 (b)). Then, since `j` in `C[i][j]` is the induction variable of `Loop j`, and the induction variable is increased by one, the communication batcher batches `specLoad`s and `specStore`s in `Loop j`, and respectively changes them to `specLoadRange`s and `specStoreRange`s with the loop iteration count `nj` (Figure 4.1 (c)). Finally, the duplication remover removes duplicated validation codes on `C[i][j]` because one of the validation codes dominates the others (Figure 4.1 (d)).

4.5 Code Generator

The code generator transforms the sequential loop to the parallel loop according to speculative pipeline strategy. The basic code generator parallelizes a loop without speculation and partitioning. For example, Figure 4.2 shows how the basic code generator transforms the example code in Figure 1.1. The parallelization strategy manager statically proves the absence of loop-carried dependences, and creates DOALL parallelization strategy for `Loop_A` in Figure 1.1. The code generator wraps the loop with function calls to `beginInvocation` and `endInvocation` that initialize and finalize the runtime library for parallel execution, and inserts calls to `produce` and `consume` to explicitly transfer register live-ins. The code generator does not initialize memory live-ins because the ASAP runtime system initializes the memory live-ins with the copy-on-access mechanism provided by the ASAP runtime. The code generator inserts communication codes for register and memory live-outs to deliver them via inter-node communication queues.

4.5.1 Speculative Code Generation

If a loop is planned to be speculatively parallelized, the code generator transforms the sequential loop to the speculatively parallelized loop. While the code generator transforms the sequential loop in the same way as the static parallel loop, it must also insert additional codes to detect and recover misspeculated execution. Algorithm 1 shows how the code generator inserts validation and recovery codes for memory flow and control flow speculation. This algorithm is for the worker processes. Section 4.5.2 describes details about recovery code generation for the master process.

First, the code generator inserts recovery codes for the case of misspeculation (Lines 1–7). The code generator creates a basic block named `recoverBB`, and inserts a function call to `waitRuntimeRecoverMemory()` that makes all the worker processes wait for the ASAP runtime system to restore memory status. Then, the code generator adds func-

Algorithm 1: *makeSpeculative*

```
Data: loop = a parallel loop
Data: controlSpeculationSet = control flow speculated branches
Data: memorySpeculationSet = memory flow speculated dependences
Result: spec_loop = a parallel loop with validation and recovery codes
/* 1. Insert recovery codes */
let header = getLoopHeader(loop);
let recoveryBB = createBasicBlock();
recoveryBB $\leftarrow$ waitRuntimeRecoverMemory();
foreach lv $\in$ loop_carried_local_variables(loop) do
    let lv_idx = getLVIdx(lv);
    recoveryBB $\leftarrow$ lv = loadLV(lv_idx);
    header $\leftarrow$ storeLV(lv_idx, lv);
/* 2. Make an iteration a transaction */
header $\leftarrow$ if(TXBoundary() == isMisspec) goto recoveryBB ;
foreach exitBB $\in$ loop_exits(loop) do
    exitBB $\leftarrow$ if(endInvocation() == isMisspec) goto recoveryBB;
/* 3. Redirect speculated branches */
foreach branchInfo $\in$ controlSpeculationSet do
    let branch = getBranchInst(branchInfo);
    let branchOutBB = getUnlikelyBranchedBB(branchInfo);
    let misspecBB = createBB();
    redirectControl(branch, branchOutBB, misspecBB);
    misspecBB $\leftarrow$ misspec();
    misspecBB $\leftarrow$ goto recoveryBB;
/* 4. Insert validation for memory flow speculation */
foreach edge $\in$ memorySpeculationSet do
    let storeInst = getSrcInst(edge);
    let loadInst = getDstInst(edge);
    before(loadInst) $\leftarrow$ specLoad(getPointerAddr(loadInst));
    after(storeInst) $\leftarrow$ specStore(getPointerAddr(storeInst));
```

```
Loop_B:
// Master Process
beginInvocation(Spec-DOALL);
produceLiveIns();
commitProcess(recoveryFcn);
consumeLiveOuts();
endInvocation();

recoveryFcn:
recoveryFcn() {
  int i=loadLV(idx_i);
  irregular[idx[i]]+=foo(i);
  if(irregular[idx[i]]>error)
    printf(“I/O operation”);
  i++;
  storeLV(idx_i, i);
}
```

```
// Worker Process
beginInvocation(Spec-DOALL);
consumeLiveIns();
executeForLoop();
return;

recoveryBB:
  waitRuntimeRecoverMemory();
  i = loadLV(idx_i);
  goto header;

executeForLoop(){
  for(int i=0; i<N; i++) {
  header:
    if(TXBoundary()==isMisspec)
      goto recoveryBB;
    if(i%NP==tid) {
      storeLV(idx_i, i);
      specLoad(&irregular[idx[i]]);
      irregular[idx[i]]+=foo(i);
      specStore(&irregular[idx[i]]);
      if(irregular[idx[i]] > error){
        misspec();
        goto recoveryBB;
  } } }
  if(endInvocation()==isMisspec)
    goto recoveryBB;
}
```

Figure 4.3: Code generation: Spec-DOALL (Loop_B)

tion calls to `loadLV` that restore local variables because the ASAP runtime system supports rollbacks only for heap-located variables. For the same reason, the code generator inserts calls to `storeLV` at the header of the loop to explicitly pass local variables in each transaction to the master process. Here, the code generator handles only loop-carried local variables in phi nodes; other local variables are either unchanged or unused across iterations, so they do not need recovery support.

Then, the code generator isolates each loop iteration as a separate transaction by inserting calls to `TXBoundary` and `endInvocation` at the loop header and every loop exit (Lines 8–10). These functions return whether the master process has sent a misspeculation signal. The code generator inserts `if` statements to check the return value, and branch instructions to recoveryBB for the worker process to initiate local recovery if the functions return true. A worker process may finish earlier than others that may be misspeculated, so endInvocation blocks the worker process until all the workers finish parallel execution.

To maximize performance improvement, the ASAP system applies different validation methods for different speculation. For control speculation (Lines 11–17), the compiler redirects speculated branches to *misspecBB*, so the runtime system can catch misspeculation early without validating memory accesses. For memory speculation (Lines 18–22), the ASAP system instruments relevant memory operations by inserting specLoad and specStore calls. These calls collect a transaction log, which is used for the runtime system to detect misspeculation. The runtime system to validate speculated memory accesses by looking at the logs.

Figure 4.3 shows how the code generator transforms Loop_B in Figure 1.1 to a Spec-DOALL loop. Inserting calls to TXBoundary and endInvocation, the code generator makes an iteration an isolated transaction. Calls to storeLV and loadLV save and restore an induction variable i. The code generator inserts validation codes for speculated dependences by inserting calls to specLoad and specStore for irregular, and redirecting if statement to the recovery codes.

4.5.2 Recovery Code Generation

If misspeculation occurs, the ASAP runtime system squashes all the following speculative iterations, and makes the master process execute the misspeculated iteration again with committed program state honoring the semantics of the original program. For the master process to execute only the misspeculated iteration, the code generator creates a recovery function that includes a clone of the original loop, and redirects back edges of the loop to a loop exit block. To restore register state, the code generator inserts codes to restore local variables.

Algorithm 2: *makeRecoveryCode*

```
   Data: loop = a sequential loop
   Result: recovery = codes for misspeculation recovery which execute misspeculated
           iteration sequentially
1  let recovery = copy(loop);
2  let header = getLoopHeader(recovery);
3  redirectBackEdgesToExit(recovery);
4  foreach lv ∈ loop_carried_local_variables(recovery) do
5      let lv_idx = getLVIdx(lv);
6      foreach exitBB ∈ loop_exits(recovery) do
7          exitBB ← storeLV(lv_idx, lv);
8      header ← newLV = loadLV(lv_idx);
9      replaceUses(lv, newLV);
```

Algorithm 2 shows the detail process of the recovery code generation. Since the recovery code is executed only for one misspeculated iteration, *makeRecoveryCode* redirects back edges to a loop exit block (Line 3). Since the ASAP runtime system recovers only memory pages, the algorithm inserts recovery codes for register states (Line 4–4). The recovery codes manage loop-carried local variables in phi nodes because the other local variables are not changed nor used in the loop. For each loop-carried local variable, *makeRecoveryCode* inserts calls to `LoadLV` function to restore the local variable at the beginning of the recovery iteration, and inserts calls to `StoreLV` function to store and pass the updated local variables to worker processes at the end of the loop. The generated recovery codes for `Loop_B` and `Loop_C` in Figure 1.1 are located at Figure 4.3 and Figure 4.4.

4.5.3 Pipeline Code Generation

If a pipeline strategy has more than one stage, the code generator splits a loop iteration into multiple partitions according to the pipeline strategy. The pipeline strategy has information only about instructions in each pipeline stage. While allocating instructions to each partition, the code generator should find dependences between instructions in different stages, and insert communication codes for them. Since the ASAP runtime system delivers

Algorithm 3: *makePartitions*

```
Data: loop = a sequential loop
Data: stages[] = instruction sets for each stage in the pipeline strategy
Result: partitions[] = pipelined loops
/* 1. Copy the whole loop to each partition, and make
   remove sets for instructions not in each stage      */
let removeSets[] = new instructionSets(size(stages));
for $idx \gets 0$ to size(stages) do
    let stage = stages[idx];
    let partitions[idx] = copy(loop);
    foreach $instruction \in partitions[idx]$ do
        if $instruction \notin stage$ then
            removeSets[idx] $\gets$ instruction;
/* 2. Insert communication between stages              */
for $idx \gets 0$ to size(stages) do
    let stage = stages[idx];
    foreach $instruction \in removeSets[idx]$ do
        foreach $use \in uses(instruction)$ do
            if $use \in stage$ then
                let srcIdx = findSource(instruction, stages);
                partitions[srcIdx] $\gets$ produce(instruction, idx);
                partitions[idx] $\gets$ newValue = consume(srcIdx);
                partitions[idx].replaceUses(instruction, newValue);
                break;
/* 3. Remove instructions from each partition          */
for $idx \gets 0$ to size(stages) do
    let stage = stages[idx];
    /* Refine control flow                             */
    foreach $basic\_block \in partitions[idx]$ do
        if all the instructions in the basic_block $\in$ removeSets[idx] then
            let succBB = findSuccessor(basic_block);
            foreach $predBB \in pred(basic\_block)$ do
                redirect(predBB, basic_block, succBB);
        else if branch instruction in the basic_block $\in$ removeSets[idx] then
            let succBB = findSuccessor(basic_block);
            basic_block $\gets$ branch succBB;
    foreach $instruction \in removeSets[idx]$ do
        remove instruction from partition[idx];
```

memory updates from prior stages to later stages, the code generator only respects register dependences between local variables.

Algorithm 3, *makePartitions*, is about how the code generator transforms a sequential loop to pipeline codes according to instruction sets for each stage. There are two ways to make partitions; 1) copying only instructions in a stage to the partition, or 2) copying all the instructions in the loop and removing instructions that are not in the stage. This algorithm uses the second method. First, *makePartitions* copies the whole sequential loop for each partition, and makes a remove set for instructions that do not belong to the stage (Line 2–5).

Then, *makePartitions* inserts communication codes for register dependences (Line 8–10). If a value in a remove set is used in a stage, the value should be delivered from a prior stage that generates the value. Function `findSource` finds the owner of the value by iterating stages, and returns the index of the owner. *makePartitions* inserts calls to `produce` and `consume` between the owner and the stage, and replaces all the uses of the value in the partition to the consumed value.

Finally, *makePartitions* removes instructions in the remove set from each partition (Line 18–28). While removing the instructions, *makePartitions* refines control flows. If all the instructions in a basic block are in the remove set, *makePartitions* redirects all the predecessors of the basic block to its successor before removing the basic block. *makePartitions* finds a transitive successor that will not be removed. There exists only one transitive successor because all the successors are not control dependent to the basic block. If they are control dependent to the basic block, the branch instruction of the basic block cannot be removed, so the basic block is not empty. For the same reason, if a branch instruction of a basic block is removed, *makePartitions* inserts a branch instruction to the non-removed successor. After refining the control flow, *makePartitions* removes all the instructions in the remove set from the partition.

Figure 4.4 shows partitioned codes for `Loop_C` in Figure 1.1. The code generator places the instructions to each partition, making three-stage pipelined parallel loops. For exam-

```
Loop_C:
// Master Process
beginInvocation(Spec-DSWP);
produceLiveIns();
commitProcess(recoveryFcn);
consumeLiveOuts();
endInvocation();

recoveryFcn:
recoveryFcn() {
  node = loadLV(idx_node);
  i = loadLV(idx_i);
  node = node->next;
  res = boo(node);
  printf(“results: %d”, res);
  i++;
  storeLV(idx_node, node);
  storeLV(idx_i, i);
}

// Worker Process
beginInvocation(Spec-DSWP);
consumeLiveIns();
if(tid==0) stage1_seq();
else if(tid < NP-1) stage2_pll();
else stage3_seq();
return;

recoveryBB_stage1:
  waitRuntimeRecoverMemory();
  node = loadLV(idx_node);
  goto header1;

stage1_seq(){
  while(node) {
  header1:
    if(TXBoundary()==isMisspec)
      goto recoveryBB_stage1;
    storeLV(idx_node, node);
    produce(node, stage2);
    produce(node, stage3);
    specLoad(&node);
    node = node->next;
    specStore(&node);
  }
  if(endInvocation()==isMisspec)
    goto recoveryBB_stage1;
}
```

```
// Worker Process (Continued)
recoveryBB_stage2:
  waitRuntimeRecoverMemory();
  node = loadLV(idx_node);
  i = loadLV(idx_i);
  goto header2;

stage2_pll(){
  while(node, i++) {
  header2:
    if(TXBoundary()==isMisspec)
      goto recoveryBB_stage2;
    node = consume();
    if(i%(NP-2) == (tid-1)) {
      storeLV(idx_i, i);
      specLoad(&node);
      res = boo(node);
      produce(res, stage3);
      specStore(&node);
    }
  }
  if(endInvocation()==isMisspec)
    goto recoveryBB_stage2;
}

recoveryBB_stage3:
  waitRuntimeRecoverMemory();
  node = loadLV(idx_node);
  goto header3;

stage3_seq(){
  while(node) {
  header3:
    if(TXBoundary()==isMisspec)
      goto recoveryBB_stage3;
    node = consume(stage1);
    res = consume(stage2);
    printf(“result: %d”, res);
  }
  if(endInvocation()==isMisspec)
    goto recoveryBB_stage3;
}
```

Figure 4.4: Code generation: Spec-DSWP (Loop_C)

ple, the code generator allocates instruction `node = node->next;` to the first partition, `res = boo(node);` to the second, and `printf("result: %d", res);` to the third. Since there is a dependence on `node` from the first partition to the others, the code generator inserts `produce(node, stage2)` and `produce(node, stage3)` to the first stage, and does `node = consume(stage1)` to the others. The code generator inserts calls to `produce` and `consume` for the dependence on `res` between the second and the third stages. The code generator changes the second stage to a parallel stage like the DOALL loop generation, and inserts validation and recovery codes for `Loop_C` like the Spec-DOALL loop generation in the previous sections.

The ASAP compiler automatically finds parallelism opportunities from sequential loops via profiling runs and static dependence analysis, and generates optimized parallel codes for the loops. To efficiently execute parallel programs on distributed memory systems of clusters, the ASAP compiler explicitly inserts and optimizes communication codes for share data. Moreover, to achieve high performance speedups on clusters that have high communication overheads, the ASAP compiler supports additional speculation techniques that require low validation and communication costs. With the careful application of speculation techniques and communication optimization, the ASAP compiler generates efficient parallel codes that have scalable performance improvement.

Chapter 5

ASAP Runtime System

The ASAP runtime system executes parallelized loops, either speculated or not, with three main components; MTX manager, communication manager, and memory manager. The MTX manager controls the life cycle of MTXs executing Spec-DSWP codes. The communication manager provides optimized communication channels to the MTX manager and the memory manager. The memory manager assists the MTX manager to efficiently execute MTXs on distributed memory systems. This chapter introduces detail implementation of the components.

5.1 The MTX manager

As Section 3.2.1 describes, there are three roles in the MTX execution; workers that speculatively execute parallel programs, a validator that validates transactions, and a master that commits transactions. The MTX manager plays each role providing different features. The MTX workers have a local variable set (LV Set) that tracks all the speculative updates of the local variables, a read set and a write set that keep speculative memory loads and stores. Instead of the LV set and the read set, the MTX validator has a validator that validates speculative memory accesses of the MTX workers. The validator compares all the read values in workers' read set with local memory values in the validator, and detects memory

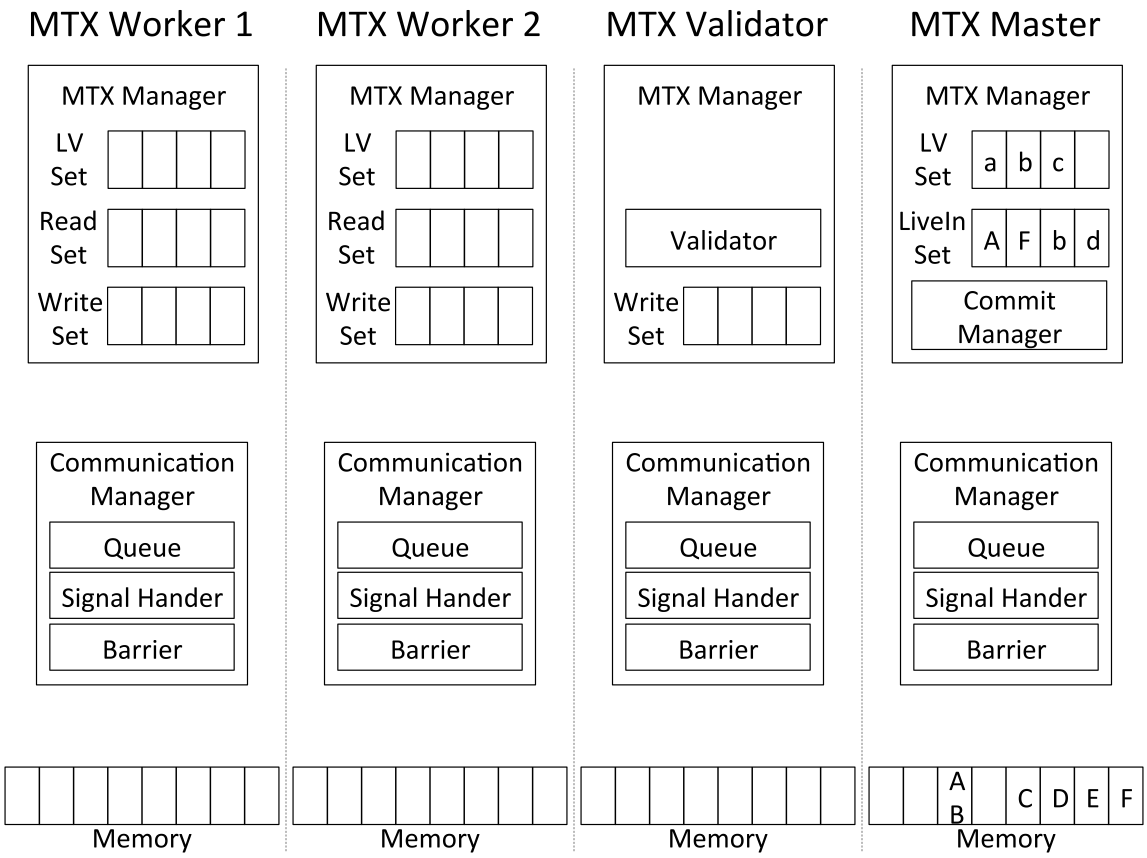


Figure 5.1: MTX manager: Initial state

misspeculation. The write set in the validator is the union of all the workers' write sets. The MTX master has a LV set that keeps the committed local variables, and a live-in set that keeps live-in values for recovery codes. The commit manager executes commits or rollbacks depending on the validation results.

5.1.1 MTX Execution

The MTX manager manages the life cycle of MTXs from initialization to commit. According to the roles, the MTX manager initializes, executes, validates and commits the MTXs on distributed memory system. If one of the MTXs is misspeculated, the manager squashes all the non-committed speculative execution, and does rollback to committed states for all the MTX processes.

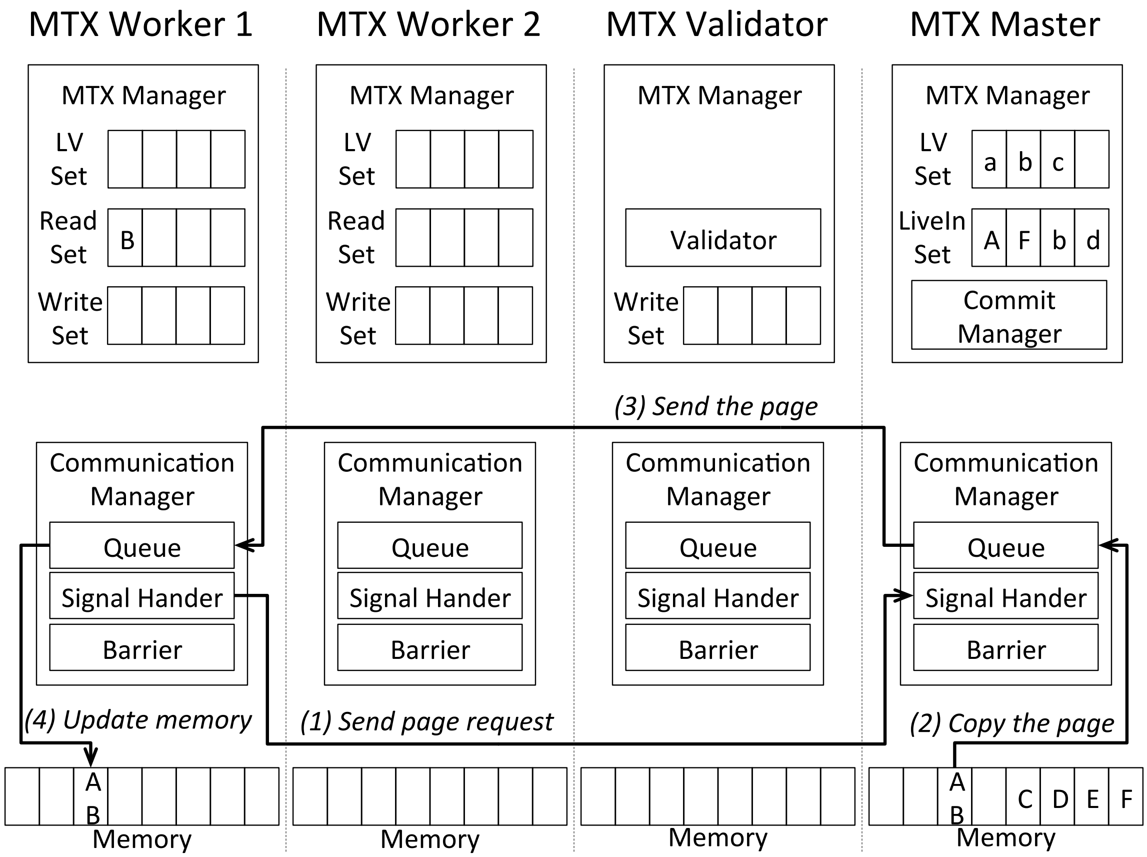


Figure 5.2: MTX manager: Memory initialization

MTX Initialization: There are mainly two ways to initialize the memory state prior to entering the parallel section. One way is that each process generates their own states by redundantly executing the sequential codes, and the other way is that only one of the processes executes the sequential codes. The MTX manager chooses the second option because the first one may not be safe due to replicated side effects. Figure 5.1 shows the MTX managers and memory states right before executing the parallel section. Since only the master process executes the sequential codes of a program, the master has up-to-date memory values while the others do not.

Each worker initializes its memory state referring the memory of the master at the first memory access. When a worker accesses an uninitialized memory, the memory manager causes a page fault signal, and sends a page request to the master. The master sends the requested page to the worker, and the worker updates its memory. Since the MTX man-

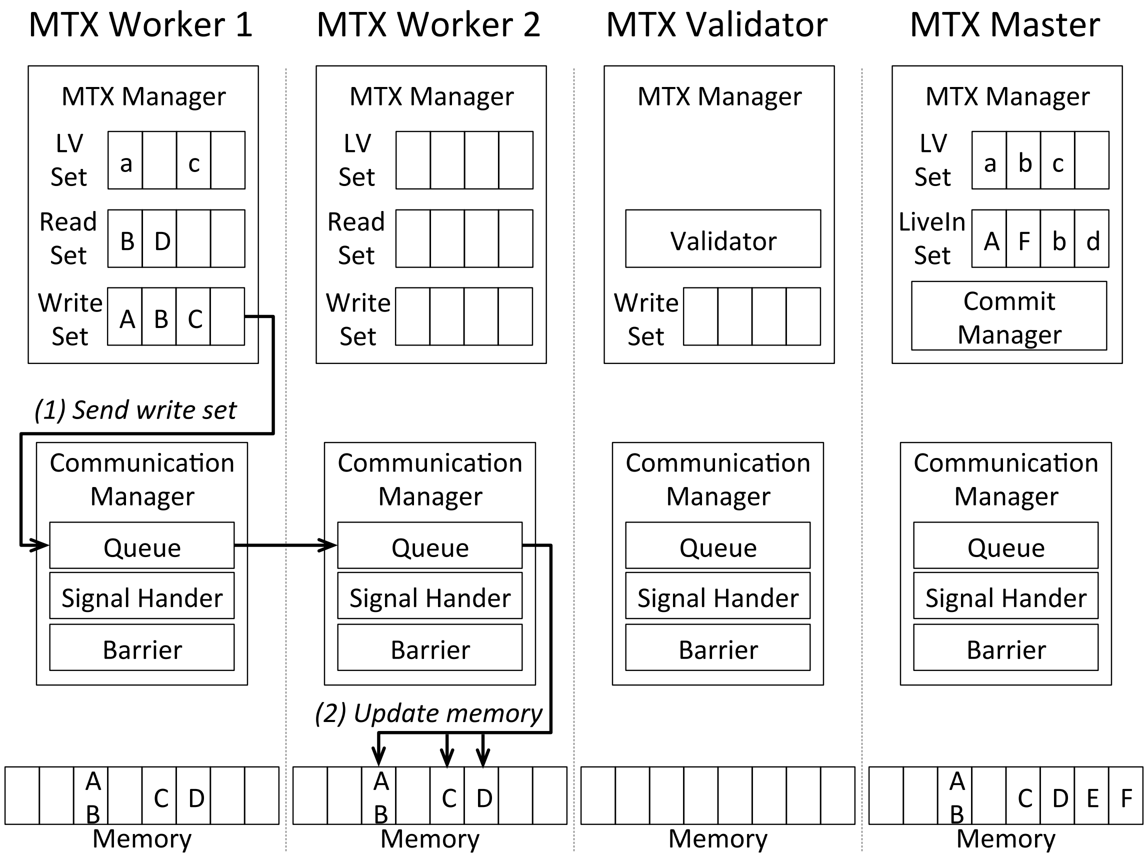


Figure 5.3: MTX manager: Pipeline execution

ager copies the whole page only for the first access, the ASAP runtime can amortize the initialization overheads if the same page is accessed again. In addition, the runtime system does not initialize memory units that are not accessed, so it can reduce communication overheads from the memory initialization. Figure 5.2 illustrates the process of the memory initialization. When the worker 1 accesses B, the MTX manager copies the page from the master to the local memory space. Since A is located in the same page, the worker 1 can access A without additional initialization. Since E and F are not accessed by the worker 1, the two pages are not copied from the master to the worker 1.

MTX Execution: Since each worker executes MTXs in different physical memory spaces, all the memory updates by a worker in an MTX should be forwarded to other workers that participate in the same MTX via the communication manager. The MTX manager tracks all the speculative and non-speculative memory updates, and stores them in a write

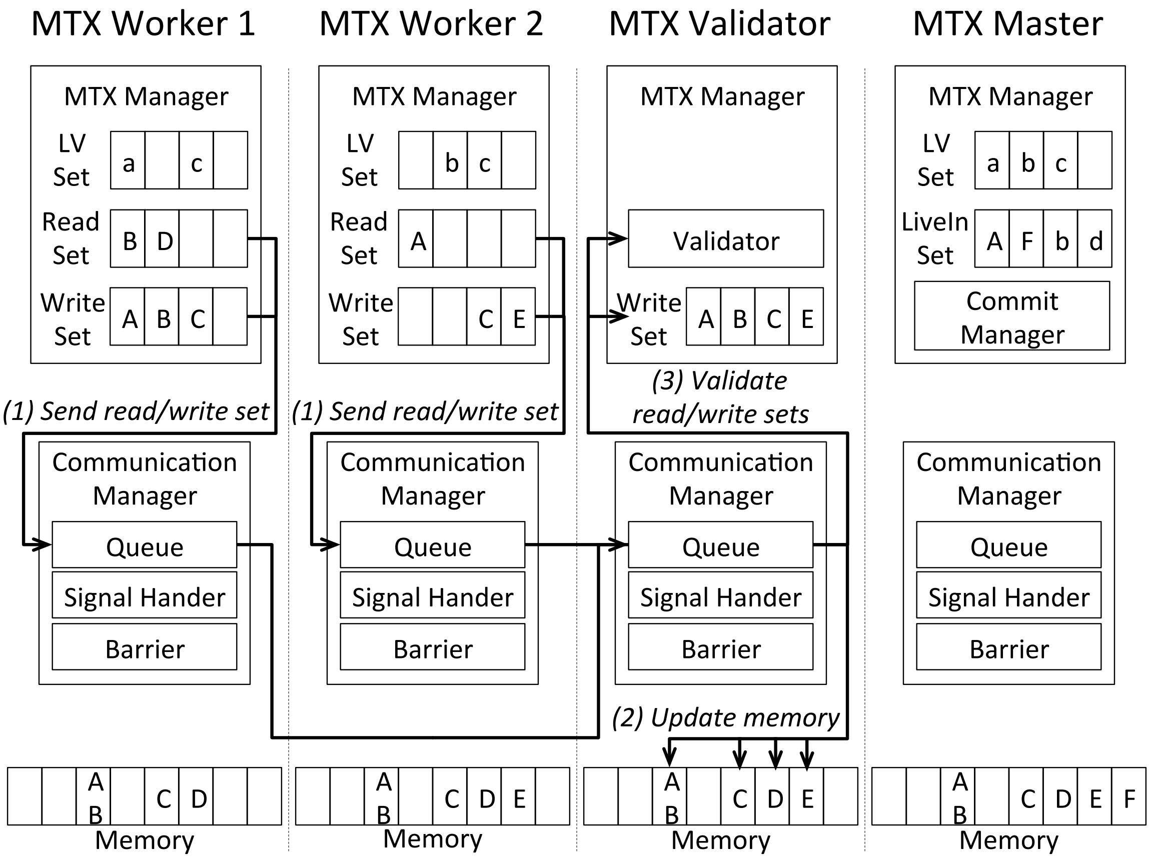


Figure 5.4: MTX manager: Validation

set. When a worker finishes a subTX, it delivers the write set to workers in later stages. Figure 5.3 shows that the worker 1 sends the write set to the worker 2, and continues to execute the next subTX. The worker 2 updates its private memory with the delivered write set, and starts its subTX. Since queues among processes can keep multiple sets from multiple MTX versions, a process can be decoupled from other processes, executing a subTX that is in several MTX versions ahead from others.

MTX Validation: The ASAP supports four types of speculation; control flow speculation, object lifetime speculation, read-only speculation, and memory flow speculation.

For the control flow speculation, the ASAP compiler inserts validation codes that compare predicted values with actual values on the speculated branch conditions. The ASAP runtime system exposes a `SignalMisspeculation` function interface for worker processes to invoke a recovery process from control misspeculation. The ASAP compiler

Algorithm 4: *validation*

```
Data: memorySets = read and write sets from workers
Result: isMisspec = validation result
let size = getWorkerProcessNumber();
for i = 0 to size do
    let memorySet = memorySets[i];
    foreach $memoryOp \in memorySet$ do
        let address = getAddress(memoryOp);
        let value = getValue(memoryOp);
        if isRead(memoryOp) then
            if *address != value then
                return true;
        else
            *address = value;
return false;
```

inserts calls to this function along all speculated control flow edges. If a speculated control flow occurs at run-time, the `SignalMisspeculation` function is called, and the misspeculation signal is sent to the MTX validator or the MTX master.

Object lifetime speculation is applied to allocation and deallocation sites whose object is likely to be private to one iteration of the loop. The ASAP compiler replaces calls to `malloc` and `free` with `specMalloc` and `specFree`. The MTX manager in each worker records a list of speculatively local objects that have been allocated. When a subTX terminates, the manager checks whether the list is empty. If any speculatively local object was not freed by the end of the subTX, the manager signals misspeculation to the validator or the master. Note that this additional bookkeeping occurs locally at each worker node, so inter-node communication is unnecessary to detect misspeculation. Speculatively local objects are allocated in the private memory space of each worker and considered thread-local, hence reducing overhead for validation and live-out communication.

For the read-only speculation and the memory flow speculation, one process is dedicated as a validator process. The validator tracks memory accesses and checks memory versions. When a MTX (i.e., single iteration) is finished without misspeculation, the valida-

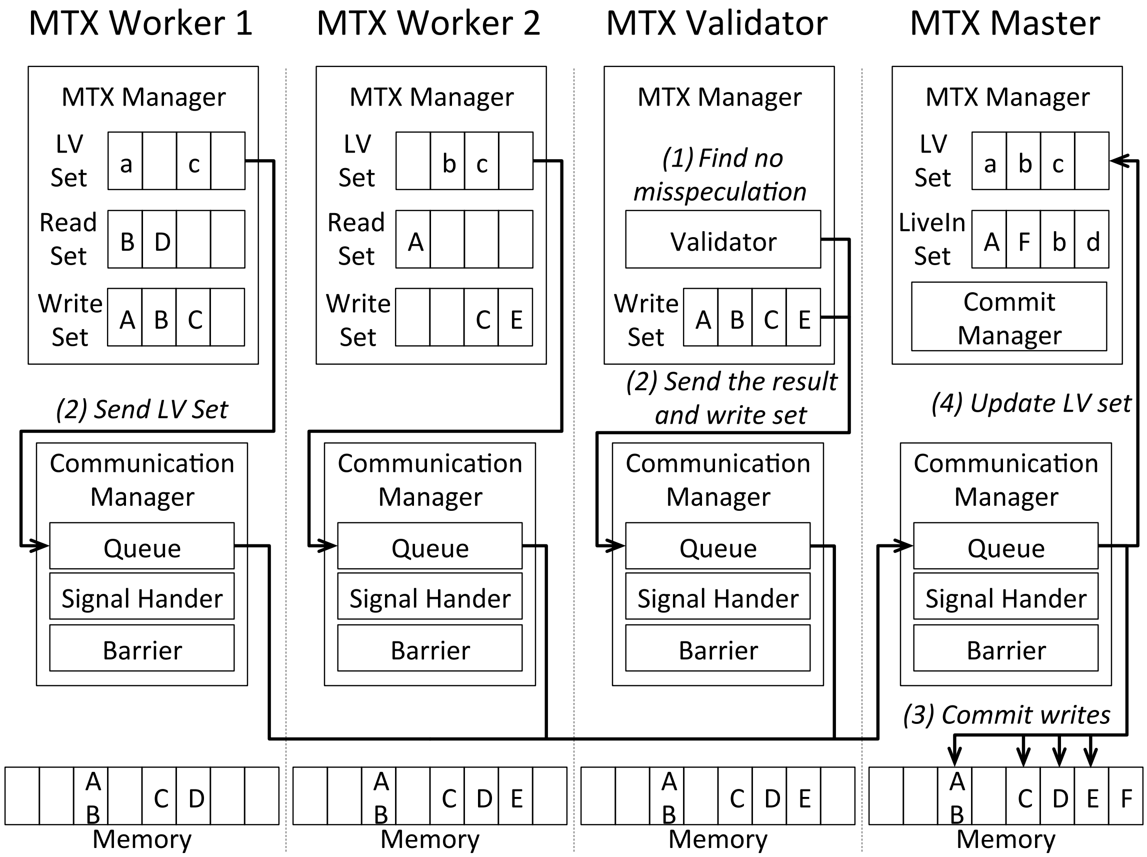


Figure 5.5: MTX manager: Commit

tor forwards all speculated stores to the master process which keeps the committed program state. Figure 5.4 shows how the MTX manager manages validation of the read-only speculation and the memory flow speculation. The workers track all the speculated reads and all the speculated and non-speculate writes, and store the memory addresses and the values in a read set and a write set. Here, the write set is the same write set in the MTX execution. When each worker finishes its subTX in a MTX, the worker sends its read set and write set to the MTX validator, and the validator validates the delivered sets.

Algorithm 4 shows how the validator validates the speculations. Because pipeline parallelism has an execution order among processes, the validator validates the read and write sets from the first process to the last one. For each memory operation, the validator uses its local memory as a scratch memory. The validator updates the values in the write sets to the local memory at the same memory address, and compares the values in the read sets

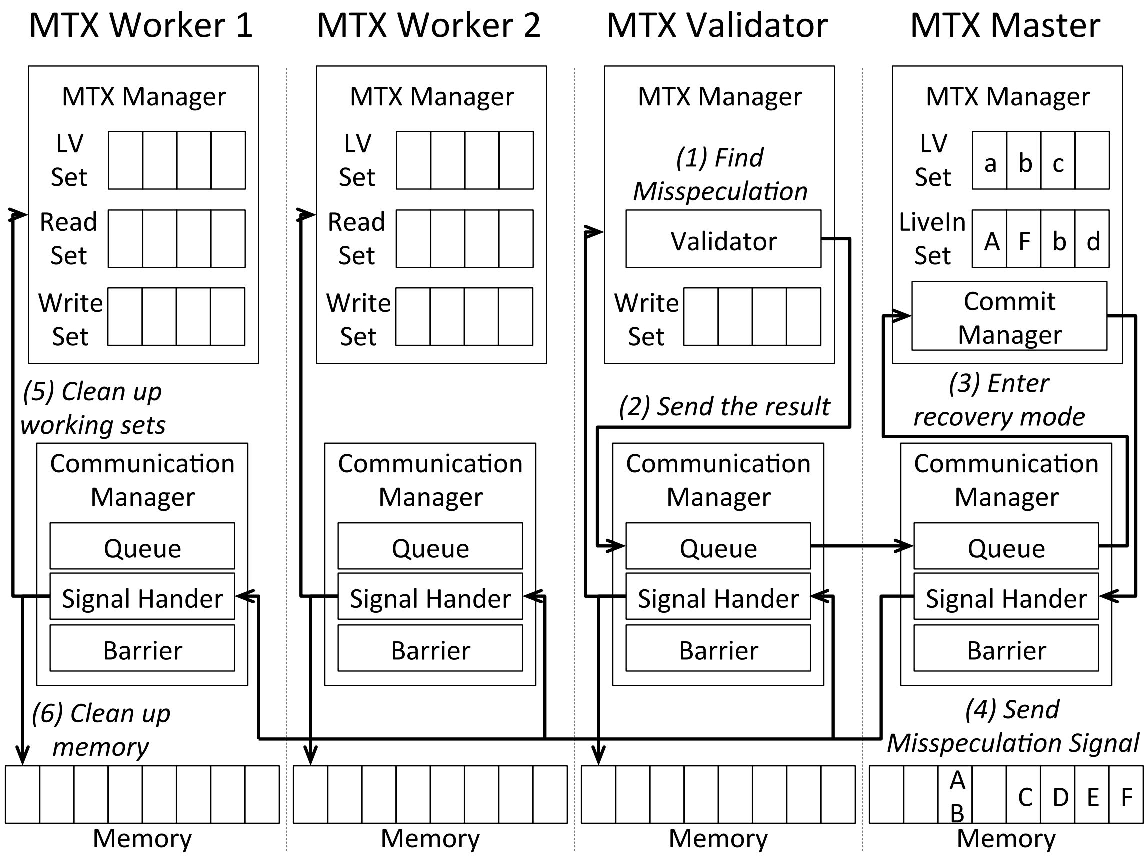


Figure 5.6: MTX manager: Rollback

with the local memory values. If a delivered read value is different from the local memory value, the validator returns the misspeculation result. The validator keeps a write set that unions all the write sets of the workers, and sends the write set to the master if there is no misspeculation.

MTX Committing: If there is no misspeculation detected during validation, the MTX validator sends all the writes in the write set to the MTX master. The MTX workers already sent their local variable updates when they finished subTXs in the same MTX. The MTX master checks the misspeculation signal, and updates the local variable sets and write sets to the local memory if there is no misspeculation delivered. Therefore, the master has only committed values in its local memory. Since there is an execution order among processes in pipeline parallelism, the master updates the local variable sets in order from the first process one to the last one. Figure 5.5 shows how the MTX managers commit a MTX.

Algorithm 5: *rollback*

```
/* 1.  Broadcast misspeculation signal                  */
master→broadcast(MISSPEC);
/* 2.  Flush queues (producer part)                     */
flushAllQueues();
barrier();
/* 3.  Clear queues (consumer part)                     */
clearMisspeculationSignal();
clearAllQueues();
barrier();
/* 4.  Re-execute sequential codes                      */
master→execute_recoveryFunction();
master→broadcastLocalVariables();
/* 5.  Re-initialize memory                             */
workers→clearMemory();
worker→consumeLocalVariables();
```

MTX Rollback: If one of the MTX workers sends a misspeculation signal, or if the validator detects a misspeculated memory operation, the MTX master starts the rollback algorithm. Algorithm 5 shows how the MTX master and MTX workers recover misspeculated execution. First, when the MTX master receives a misspeculation signal, the master broadcasts the misspeculation signal to all the processes. This broadcast allows all the process to enter recovery mode. Some of the worker may be waiting for others to produce some values, so the master and workers flush all the data in their queues to prevent deadlock. After all the processes flush their queues, the processes clear their queues that may have delivered data from misspeculated execution. They also clear the misspeculation signal for the next use. After all the processes clear their queues, the master starts to execute the original sequential codes, and the workers clear their local memory that have misspeculated results. Then, only the master has the committed values while the others do not, like initial memory states. Finally, the master broadcasts correct local variables to workers, workers consume and update the local variables, and all the processes resume the parallel execution.

5.1.2 Recovery Overheads

Figure 5.7 shows the overheads of the ASAP runtime system when it executes a parallel program and recovers misspeculated execution. In well-speculated cases, the communication cost between processes becomes overlapped with later communication, so the system pays only one time cost for communication as pipeline fill time. In addition, since the ASAP runtime system creates a validator and a master as separate processes, the overheads of validation and commit are overlapped with computation work of worker processes, and become off the critical path. Therefore, the ASAP runtime system can have low overheads for communication and speculation management.

However, while the separate validator and master processes allow the ASAP runtime system to have low overheads in speculation management, they can cause additional misspeculation penalty because more work is speculatively executed in advance in multiple processes. As a result, the high accuracy of speculation is important for the ASAP runtime system to achieve scalable performance improvement.

As Figure 5.7 illustrates, the misspeculation penalty of the ASAP runtime system consists of four factors such as Enter Recovery Mode (ERM), FLush Queue (FLQ), SEQuential execution (SEQ), and ReFill Pipeline (RFP). ERM is the period between the time when the master notices misspeculation and the time when the whole system enters the recovery mode. Although each process receives the misspeculation signal either at the begin or at the end of subTXs in the Figure 5.7, it can receive in the middle of the subTX if a function about signal check is called, and the ERM overhead can be reduced instead of causing checking overheads. FLQ is the overhead caused by flushing and clearing communication queues. Since multiple processes execute multiple transactions communicating each other, there can be misspeculated values in the communication queues, so the runtime system should flush and clear the queues. SEQ is the overhead for the master to sequentially execute the misspeculated iteration again. The sequential re-execution allows the master to have correct up-to-date memory without misspeculation. While the master executes the

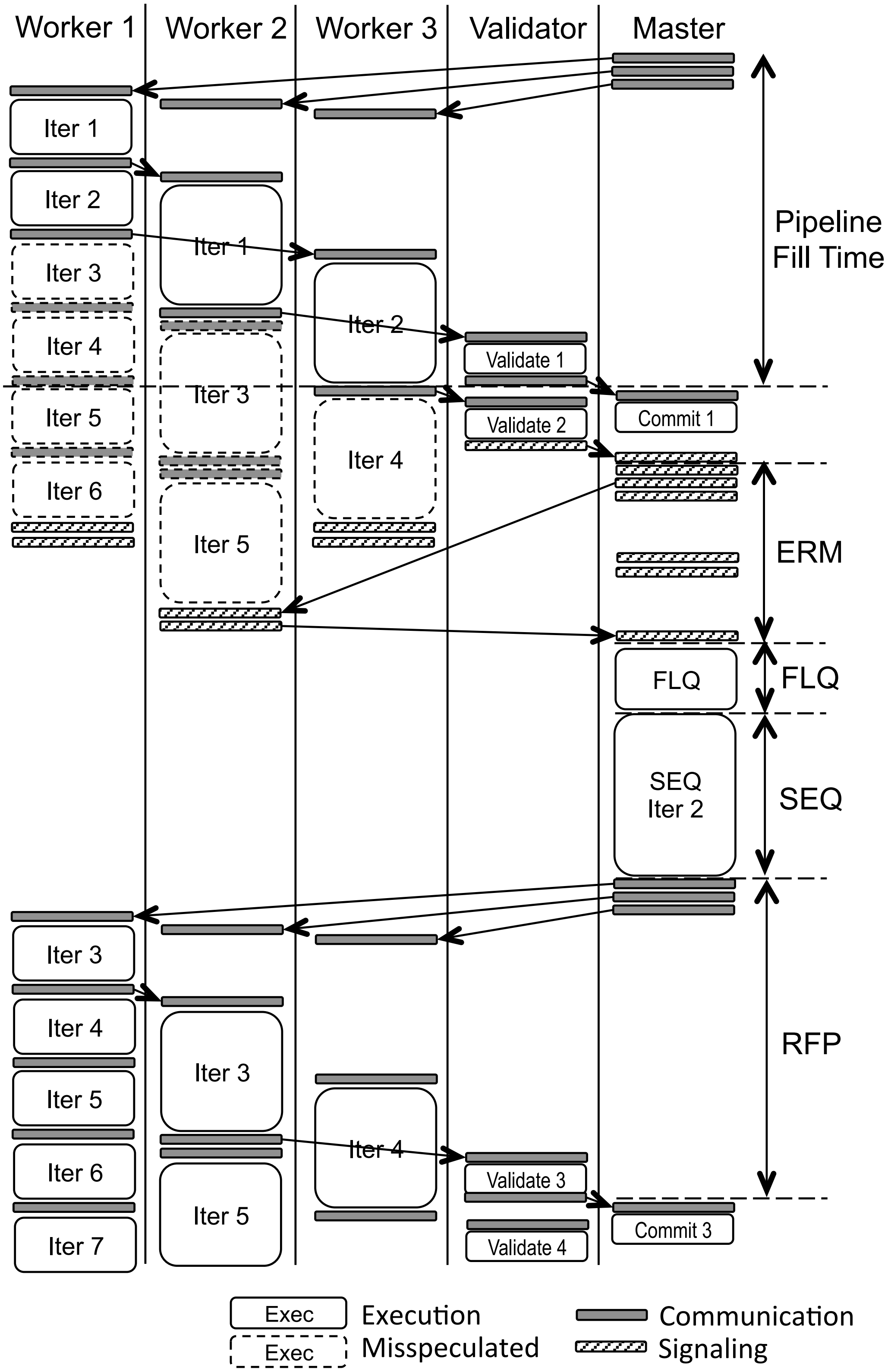


Figure 5.7: Misspeculation recovery overheads (ERM: Enter Recovery Mode, FLQ: FLush Queue, SEQ: SEQuential execution, and RFP: ReFill Pipeline)

sequential iteration, workers and the validator initialize their local memory to clear the misspeculated memory status. Since their overheads are overlapped by the master's SEQ, the overheads are not observable to users. Finally, RFP is the overhead to refill the pipeline after the ASAP runtime system resumes the speculative parallel execution. Since the ASAP runtime system executes iterations in order, it squashes all iterations higher than the misspeculated one, hence causing the work done in the later iterations to go to waste. This squash empties the pipeline after misspeculation, so the dissertation includes RFP to the recovery overheads.

5.2 The Communication manager

The communication manager mainly provides three types of communication for the ASAP runtime system; 1) message queue, 2) page handler, and 3) signal handler. These communication operations are implemented on top of the MPI system [54].

Message Queue: The communication manager provides multiple message queues to execute different runtime operations such as controlling MTXs, delivering read and write sets, and managing I/O operations. The manager reduces communication overheads by batching the produced data in a local queue instead of directly calling MPI functions.

Figure 5.8 illustrates how the communication manager produces and consumes data using the message queues. When a process produces data destined for another process, the data is stored in a local queue. When the amount of stored data exceeds a threshold, or when `flushQueue` function is explicitly called, the communication manager starts delivering the stored data to the opposite process. The manager copies the whole data in the local queue to the system buffer with a tag that indicates the message queue. Since there are multiple message queues between two processes, the tag is unique for each message queue. Then, the communication manager sends the data to the opposite process. When the opposite process receives the data, the data is temporarily stored in its system buffer.

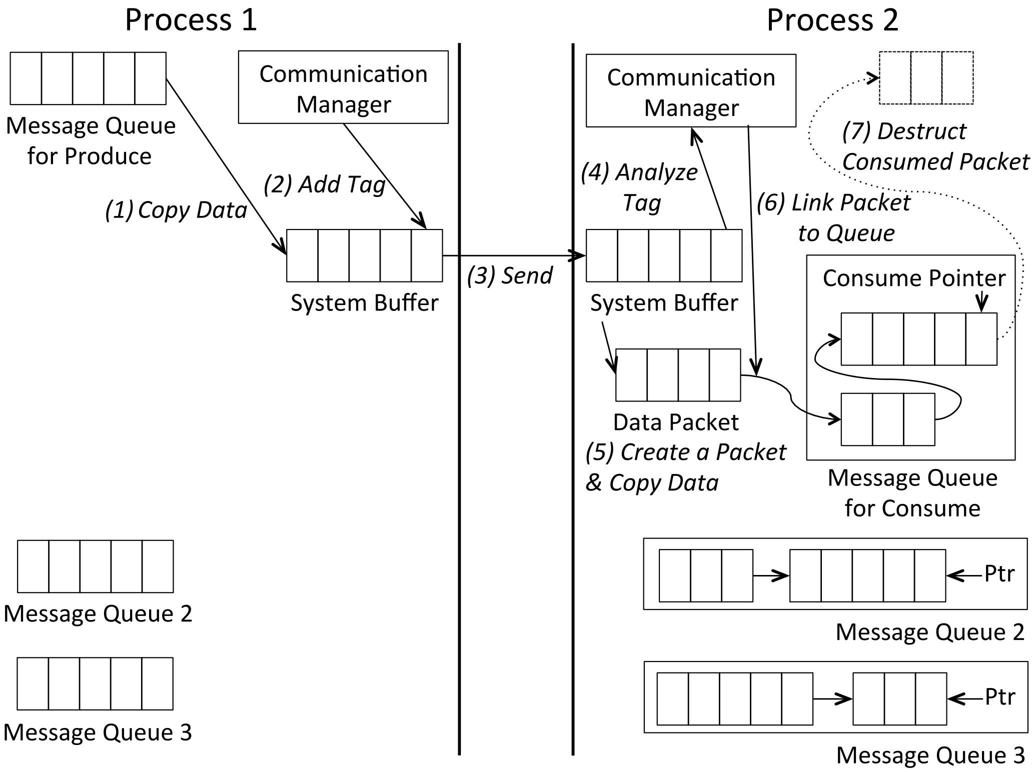


Figure 5.8: Communication manager: Handling message queues

After the communication manager analyzes the tag, the manager creates a data packet, copies the data to the packet, and links the packet to the opposite queue. When the opposite process calls the consume function, the data in the queue is consumed in order. When all the data in the packet is consumed, the queue destroys the data packet.

Page Handler: The communication manager helps the memory manager initialize the local memory. At the beginning of the parallel execution, all the local memories in the workers and the validator are not initialized. When the workers or the validator access a page at the first time, the memory manager starts to initialize the accessed page by copying the page from the master. Figure 5.9 shows how the communication manager copies the page from the master to the worker or the validator. When the memory manager requests a page to the communication manager, the communication manager sends the request to the master. In the request, there are only a page address and a page request tag. After the communication manager in the master analyzes the tag, the communication manager

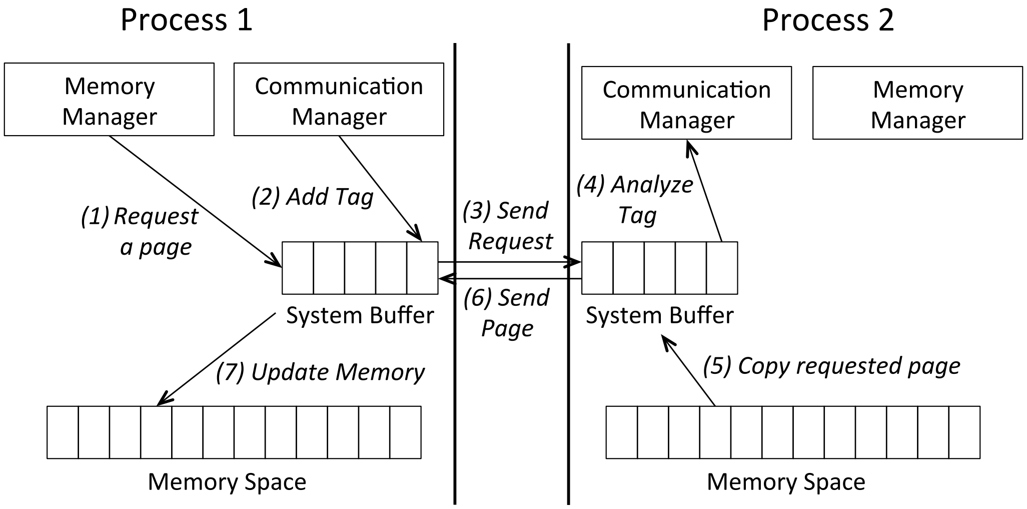


Figure 5.9: Communication manager: Handling page request

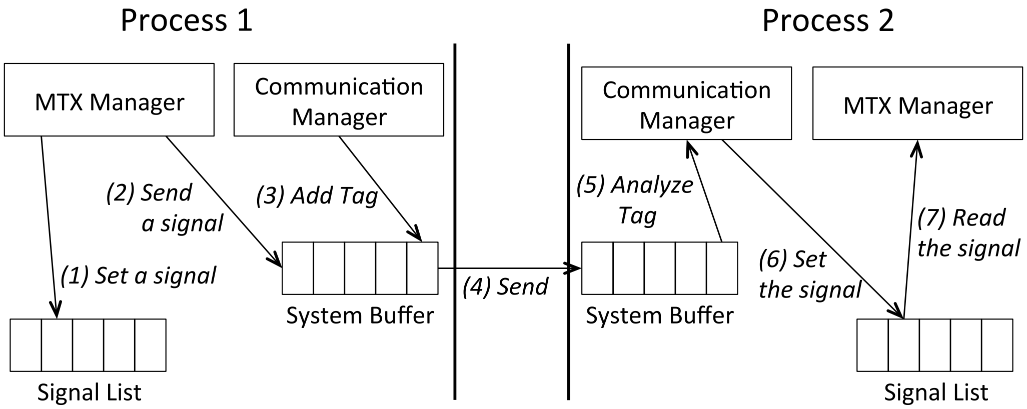


Figure 5.10: Communication manager: Handling signals

directly copies the requested page from the local memory, and sends it to the request sender. Then, the memory manger in the worker updates the delivered page to the local memory.

Signal Handler: The ASAP runtime system has multiple system states such as misspeculation, end of invocation, and exit. The runtime system uses flags to manage the states, and makes the communication manager notify the state to other processes. Figure 5.10 shows how the communication manager notifies signals. For example, when the master process detects misspeculation, it sets the misspeculation flag. Then, the communication manager sends the signal to the other processes to notify the stage change, and the communication manager in the other processes receives the signal and updates their flags.

When the MTX managers in the other processes access the updated flags about misspeculation, they can enter the recovery mode.

5.3 The Memory manager

The memory manager provides four features; memory initialization, unified virtual address space, shadow memory for memory speculation validation, and dirty page check.

The memory manager initializes memory when a page is accessed at the first time. (Copy-On-Access, COA). Figure 5.11 illustrates memory management of the ASAP runtime system for two speculative workers and the master. Initially, only the master executes the sequential, non-transactional codes, so the master has all the pages in the local memory. The others do not have any page, and their page tables are not initialized with access protections. When a worker accesses a memory location, the protection mechanism results in a page fault, causing a transfer of the page from the master to the worker. This COA transfers only data that are actually demanded by each worker, thereby avoiding the transfer of excessive, unnecessary data.

The COA is implemented in a memory page granularity because a word granularity is inefficient on clusters due to its high round-trip latency. By sending a memory page in response for a request for a word, the memory manager aggressively speculates that words near the original location will be accessed by the worker in the future. This serves as a constructive prefetching mechanism that amortizes the round-trip latency cost over accesses to multiple locations in the same page.

In addition, the memory manager provides the unified virtual address (UVA) space to all the processes on distributed memory systems. The whole address space is divided into multiple ranges for each process. The ASAP compiler redirects all the memory allocation and deallocation function calls to the new functions that the memory manager provides. The new functions allow a process to allocate the memory on the target range that the

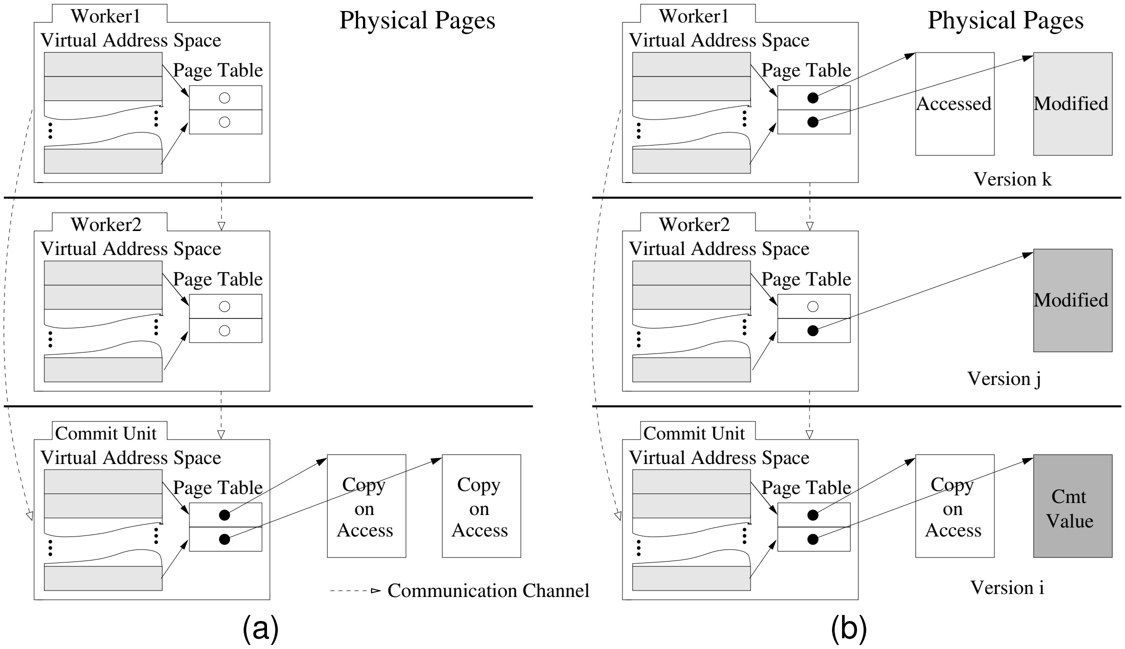


Figure 5.11: Memory layout: (a) Initial state: Only the page table of the master maps the non-speculative pages while the others are uninitialized. (b) MTX execution: The page tables map accessed virtual address range to copied private pages, or keep non-accessed page entry uninitialized.

process owns. When another process accesses the memory address, the COA mechanism copies the page from the owner to the process.

UVA is similar to distributed shared memory (DSM) [6], but it does not support a coherence mechanism unlike DSM systems. Memory coherence requires cyclic communication because a process requests data and the other replies it. The cyclic communication on clusters causes high communication overheads due to high communication latency. Instead, UVA only provides the same virtual address space to all the processes, and the compiler explicitly inserts message passing codes to respect existing data dependences. The pipeline parallelism makes the message passing path acyclic, so the runtime system can be more tolerant of the high communication latency.

When the memory manager initializes a page, it also allocates a shadow page for the page to track read and write operations. When a process writes data at a memory address, the memory manager checks the shadow memory. If the address is first written, the MTX manager adds the write to the write set. When a process speculatively reads a memory, the memory manager checks the shadow memory. If the address has not been read nor written, the MTX manager adds the read to the read set. Here, the manager also checks the write for speculative reads because the read operation on a written memory in the same transaction do not cause any misspeculation. Since the shadow memory allows the MTX manager to avoid inserting duplicated reads and writes to the read and write sets, it reduces communication overheads from validation. To reduce the amount of memory usage due to the shadow memory, the ASAP runtime allocates the shadow memory only for the accessed memory in a transaction.

The read set can be empty if no speculative reads occur, or if all the speculative reads operate at the already written memory spaces. If all the read sets in a MTX are empty, the validator does not cause misspeculation. The ASAP runtime system optimistically does not send the read and write sets to the validator if all the read sets are empty, and reduces communication overheads. If the read set is empty for a series of MTXs, the MTX masters send the read and write sets after a threshold to reduce rollback overheads.

Finally, the memory manager tracks memory accesses during the sequential execution of the master, and marks pages dirty. If the program executes parallel codes again, the memory manager initializes only the dirty pages because the others are not modified. This dirty page check reduces initialization overheads especially when inner loops are parallelized.

The ASAP runtime system executes Spec-DSWP codes on clusters without any hardware support. The MTX manger provides two features such as group transaction commit and uncommitted value forwarding, and safely executes MTXs in the Spec-DSWP codes. The communication manager and the memory manager allow the MTX manager to efficiently execute Spec-DSWP codes without worrying about hardware structure of clusters. With the MTX, communication and memory managers, the ASAP runtime system realizes the speedup potential of various parallel codes on commodity clusters.

Chapter 6

Evaluation

The ASAP system is evaluated with benchmarks from PolyBench/C [61], SPEC CPU92, CPU95, CPU2000, CPU2006 [73], and PARSEC [11] written in C. Table 6.1 lists the selected benchmarks along with information such as benchmark suite, benchmark description, and parallelization paradigms. Details of each benchmark can be found in [11, 61, 73]. The selected benchmarks exhibit diversity in terms of parallelization paradigms, types of speculation required, and communication characteristics.

The parallel programs are evaluated on a commodity cluster with 10 nodes, 12 cores per node (120 cores in total). Each node has two Intel 6-core Westmere X5650 processors running at 2.67 GHz with 48 GB of memory. It runs 64-bit RedHat Enterprise Linux v5. The inter-node communication link is Mellanox ConnectX Infiniband x4 QDR. OpenMPI (v1.4.5 with gcc v4.1.2, -O3) is used as the underlying communication layer. The ASAP compiler builds on the LLVM compiler infrastructure (r164307) [45].

The evaluation uses two different input sets; profiling inputs and evaluation inputs. The ASAP system profiles the sequential programs with profiling inputs. To use different inputs for profiling and evaluation, and to accept a problem size from the command line, this dissertation changes statically allocated fixed size arrays to dynamically allocated variable size arrays in the PolyBench/C benchmarks. This modification makes the benchmarks more

Benchmark	Source Suite	Description	Automatic Parallelization	Manual Parallelization
2mm	PolyBench/C	2 matrix multiplications	✓	
3mm	PolyBench/C	3 matrix multiplications	✓	
cholesky	PolyBench/C	Cholesky decomposition	✓	
correlation	PolyBench/C	correlation computation	✓	
covariance	PolyBench/C	covariance computation	✓	
doitgen	PolyBench/C	multiresolution analysis	✓	
dynprog	PolyBench/C	dynamic programming (2D)	✓	
fdtd-2d	PolyBench/C	2-D finite different time domain	✓	
gemm	PolyBench/C	matrix-multiply	✓	
reg_detect	PolyBench/C	2-D image processing	✓	
symm	PolyBench/C	symmetric matrix-multiply	✓	
syr2k	PolyBench/C	symmetric rank-2k operations	✓	
syrk	PolyBench/C	symmetric rank-k operations	✓	
052.alvinn	SPEC CFP 92	neural network	✓	✓
130.li	SPEC CINT 95	lisp interpreter		✓
164.gzip	SPEC CINT 2000	file compressor		✓
179.art	SPEC CFP 2000	image recognition		✓
197.parser	SPEC CINT 2000	English parser		✓
256.bzip2	SPEC CINT 2000	file compressor		✓
456.hmmer	SPEC CINT 2006	gene sequence database search		✓
464.h264ref	SPEC CINT 2006	video encoder		✓
crc32	Ref. Impl.	polynomial code checksum	✓	✓
blackscholes	PARSEC	option pricing	✓	✓
swaptions	PARSEC	portfolio pricing	✓	✓

Table 6.1: Benchmark details

difficult for the compiler to analyze because the memory allocation size is not decided at compile-time.

The evaluation inputs are chosen for the original sequential programs to run longer than one hour to observe performance scalability on a large number of cores. Eight benchmarks from PolyBench/C are not used for evaluation because their execution time is too short to be parallelized even with large input sets. Among remaining 22 benchmarks, the ASAP compiler does not parallelize 9 benchmarks such as `adi`, `floyd-warshall`, `gramschmidt`, `jacobi-1d-imper`, `jacobi-2d-imper`, `lu`, `ludcmp`, `seidel-2d`, and `trmm` because the performance estimator predicts the parallel loops are not profitable although they can be parallelized.

Since the ASAP runtime can support speculative manual parallelization, this chapter additionally evaluates the runtime system with manually parallelized CPU-intensive benchmarks that require speculation for efficient parallelization. Codes are manually parallelized in a systematic manner. Due to the difficulty of manual application of compiler algorithms, benchmark selection was influenced by source code tractability.

6.1 Performance Speedup

Table 6.2 shows how the ASAP compiler parallelizes benchmarks with different kinds of speculation and communication optimization. Although speculation is not necessary to manually parallelize PolyBench/C benchmarks, the ASAP compiler employs speculation to overcome imprecise and fragile static alias analysis. Figure 6.1 shows the performance speedup. Base is the execution time of the original sequential program. In this graph, the horizontal axis shows the number of cores, and the vertical axis shows full application speedups. All execution times were averaged over five runs. The evaluated benchmarks are categorized into two groups; automatically parallelized and manually parallelized benchmarks.

6.1.1 Automatically Parallelized Benchmarks

The ASAP compiler automatically finds parallelism opportunities and parallelizes the sequential program. Among 17 benchmarks, the ASAP system achieves speedups on 13 benchmarks with a synergistic combination of three design choices.

First, **speculation makes fully-automatic parallelization possible.** Imprecision of static analysis limits classical automatic parallelization. Table 6.2 shows that only 3 benchmarks such as `3mm[A]`, `correlation[A]`, and `covariance[A]` among the 13 benchmarks with performance speedup are parallelized without any speculation. Without speculation, the ASAP compiler cannot parallelize loops in the other 10 benchmarks, thus losing performance speedup opportunities. Figure 6.2 compares performance speedups between non-speculative parallelization and speculative parallelization. Speculation increases the

Benchmark	Parallelized Loops			Speculation				Comm. Opti.			Coverage of P'llized Loops
	DOALL	Spec-DOALL	Spec-DSWP	Mem	Ctrl	Obj	Read	P	B	D	
2mm[A]	1	1	0	4	2	0	0	0	3	2	>99.9%
3mm[A]	3	0	0	0	0	0	0	0	3	3	>99.9%
cholesky[A]	0	0	1	1	0	0	0	0	0	0	>99.9%
correlation[A]	1	0	0	0	0	0	0	0	0	0	99.5%
covariance[A]	1	0	0	0	0	0	0	0	0	0	99.7%
doitgen[A]	0	1	0	6	2	0	0	1	3	0	97.3%
dynprog[A]	0	0	1	20	3	0	0	1	0	0	>99.9%
fdtd-2d[A]	0	0	1	7	0	0	0	0	4	0	98.9%
gemm[A]	0	1	0	4	2	0	0	0	2	1	99.4%
reg_detect[A]	0	1	0	18	4	0	0	0	4	0	>99.9%
symm[A]	0	0	1	0	1	0	0	0	1	0	>99.9%
syr2k[A]	0	1	0	2	1	0	0	0	1	0	99.3%
syrk[A]	0	1	0	2	1	0	0	0	1	0	99.6%
052.alvinn[A]	0	1	0	264	0	384	0	0	0	0	85.5%
052.alvinn[M]	0	1	0	7	0	0	0	3	6	0	85.5%
130.li[M]	0	0	1	87	3	0	0	9	3	0	>99.9%
164.gzip[M]	0	0	1	140	0	0	0	13	3	0	98.4%
179.art[M]	0	0	1	40	0	0	0	9	0	0	>99.9%
197.parser[M]	0	0	1	172	8	0	0	5	3	0	>99.9%
256.bzip2[M]	0	0	1	64	1	0	0	16	7	0	99.3%
456.hmmer[M]	0	0	1	59	0	0	0	1	0	0	>99.9%
464.h264[M]	0	0	1	455	0	0	0	15	0	0	93.6%
crc32[A]	0	0	1	4	3	5	0	1	0	0	>99.9%
crc32[M]	0	0	1	0	2	0	0	1	0	0	>99.9%
blackscholes[A]	1	1	0	12	0	84	382	2	0	0	>99.9%
blackscholes[M]	0	0	1	10	1	0	0	0	0	0	>99.9%
swaptions[A]	0	1	0	4	5	8761	868	1	0	0	>99.9%
swaptions[M]	0	1	0	4	2	19	0	2	0	0	>99.9%

Table 6.2: Parallelization details: DOALL, Spec-DOALL, and Spec-DSWP show the number of parallelized loops with the parallelization scheme. Mem is the number of applied memory flow speculation, Ctrl is the number of speculated branches, Obj is the number of memory flow dependences removed by object lifetime speculation, and Read is the number of memory flow dependences by read-only speculation. In communication optimization, P, B and D stand for the number of promoted, batched and removed duplicated function calls, respectively. Coverage shows the execution time ratio of parallelized loops over the entire program. [A] and [M] in the benchmark names stand for automatically and manually parallelized programs.

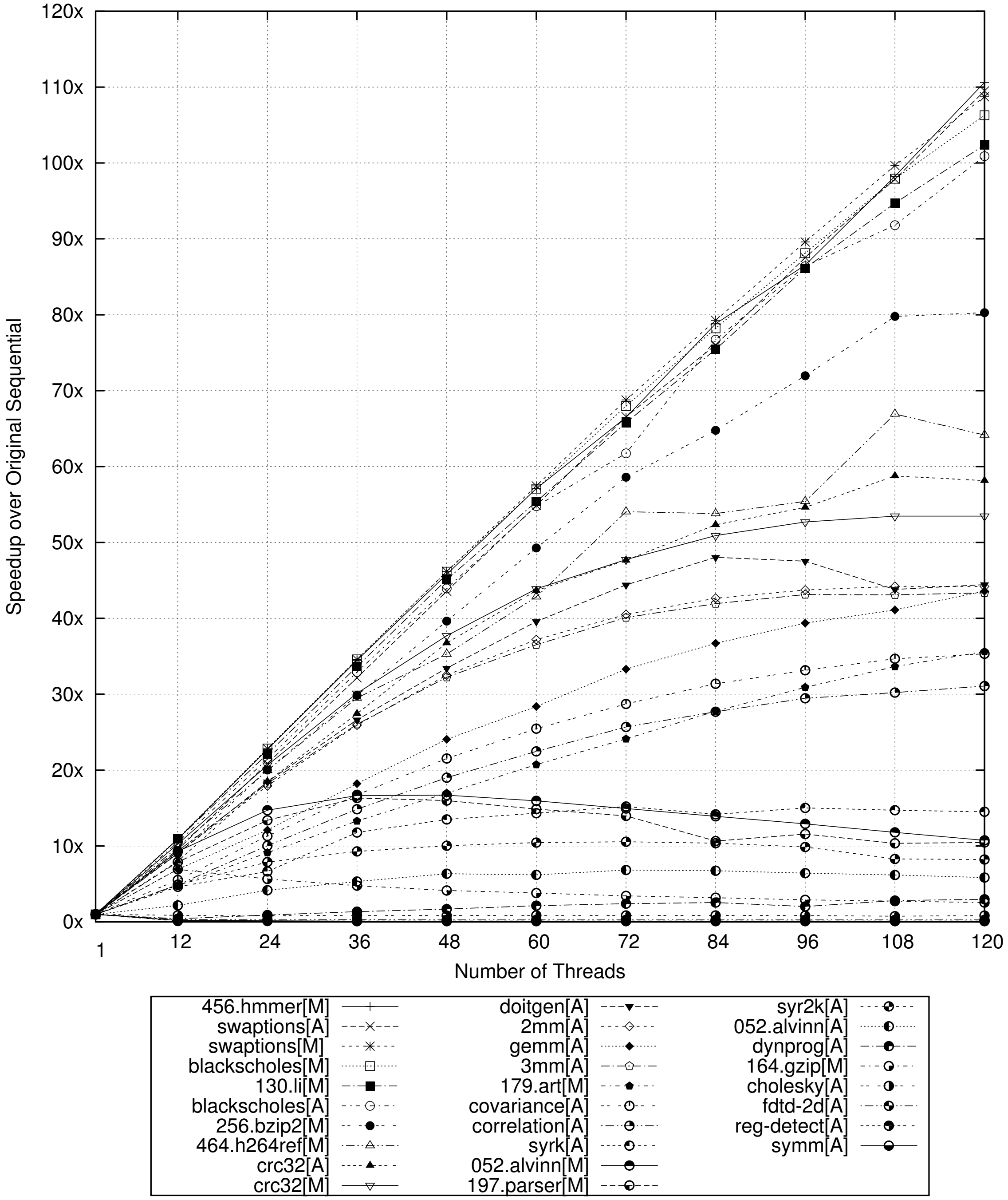


Figure 6.1: Overall speedup on 120 core cluster (Benchmarks in the legend are ordered from highest to lowest speedup)

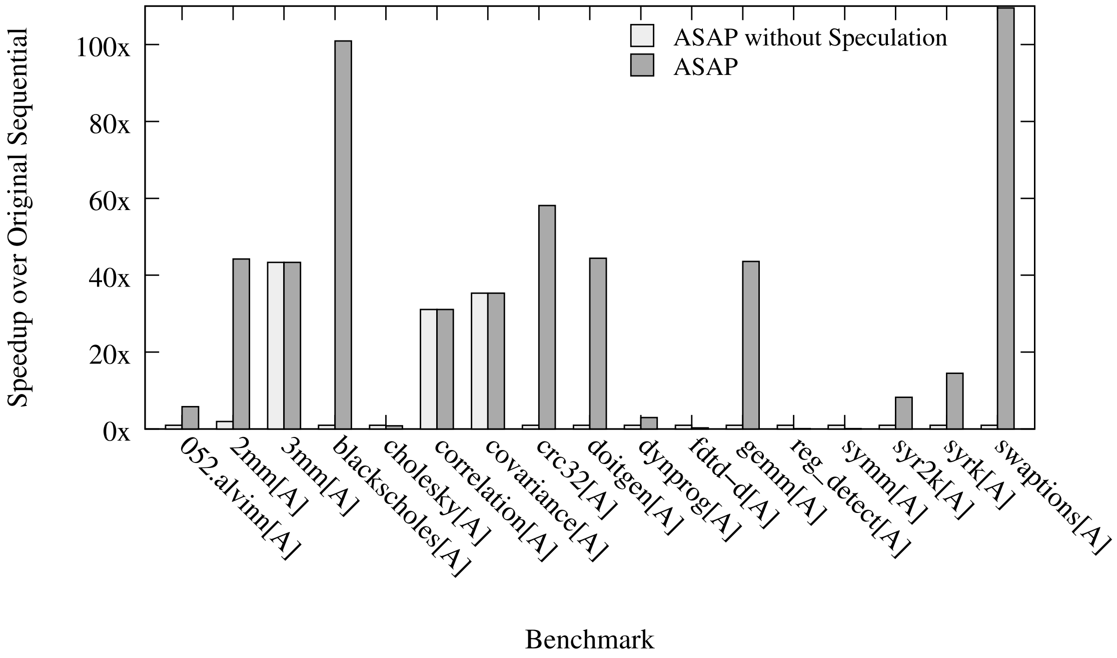


Figure 6.2: Performance Effect of Speculative Parallelization

applicability of the ASAP system to more loops in the benchmarks, and leads performance speedups of the benchmarks.

Moreover, speculation makes the ASAP compiler robust enough to overcome fragility of static alias analysis. For example, the evaluated benchmarks have been slightly updated from the earlier version. Although the two versions execute the same tasks, the update makes the ASAP compiler require speculation to parallelize a loop in `2mm[A]` that did not require in the earlier version that was used in [40]. If the ASAP compiler relies only on static alias analysis, the compiler could not parallelize the loop, hence losing performance speedup. Speculation allows the ASAP system to have stable applicability and scalability.

Second, **communication optimization realizes the high performance speedup potential in the** ASAP **system.** Some programs have a high ratio of memory accesses to computation. For example, each iteration in `2mm[A]`, a matrix multiplication benchmark, requires two loads and one store to execute only one floating-point multiplication. This high rate of memory accesses requires a large amount of communication, degrading performance. When applicable, based on the communication pattern, the ASAP compiler

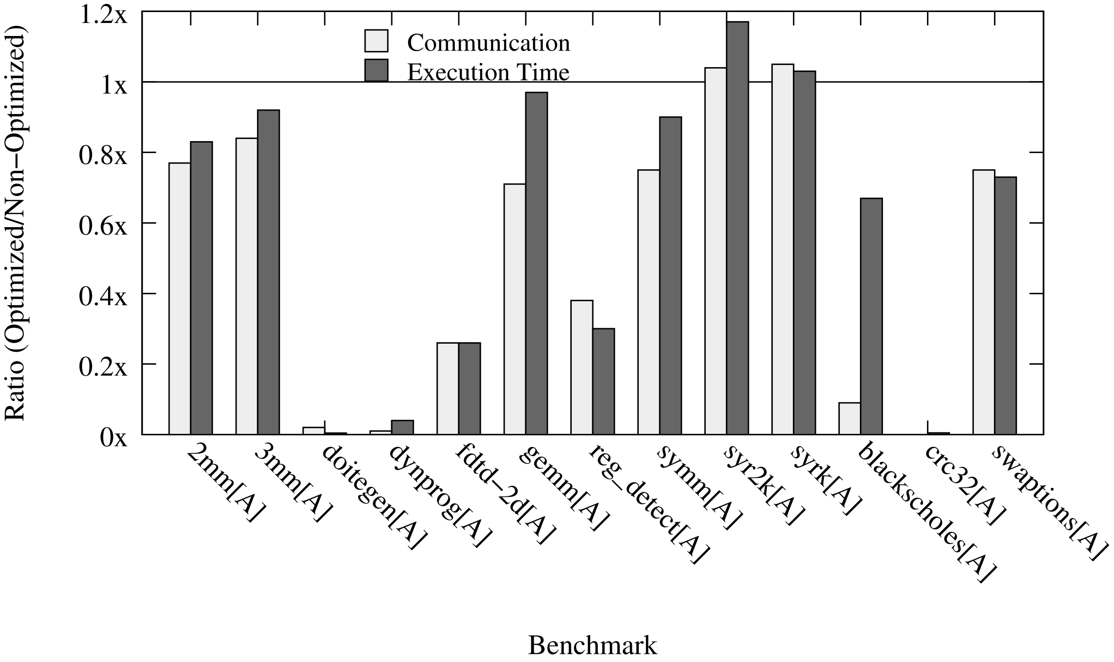


Figure 6.3: Effects of communication optimization on 12 cores: The execution time and the communication amount are normalized to non-optimized values. Lower ratio means more optimized communication.

optimizes communication by promoting, batching and removing communication function calls such as `produce`, `specLoad` and `specStore`. Figure 6.3 shows at most over 99% of the communication is optimized away, thus reducing execution time. Small input sets and 12 cores are used for this result because the unoptimized versions explode execution time.

In addition, the ASAP compiler privatizes dynamically allocated memory objects if the objects are speculated to be iteration-local. Its performance impact is separately evaluated using `swaption[A]` benchmark and two versions of the runtime with and without privatization on 12 cores. A small input set is used because the execution time explodes without privatization. With privatization, the volume of communication in bytes decreases by 99.6%.

Third, **static analysis and separate validation and commit processes reduce validation overhead.** Software transactional memory systems have suffered from large validation overhead [20]. Once the validation cost exceeds a certain threshold relative to the execution time of a loop iteration, the validation process becomes a performance bottleneck. Although the task of validation is effectively offloaded to a separate process to overlap validation with computation in the worker processes, it is still important to reduce the amount of speculative memory accesses, and develop high quality static alias analysis.

However, there are four benchmarks that experience slowdown although the performance estimator predicts speedup: `cholesky[A]`, `fdtd-2d[A]`, `reg_detect[A]`, and `symm[A]`. Based on the parallelization strategy, these benchmarks are divided into two classes.

For the first class, which is exemplified by `cholesky[A]`, `fdtd-2d[A]` and `symm[A]`, the performance speedup is limited by Amdahl's Law. The loops are parallelized with Spec-DSWP scheme, but most instructions in the loop are sequentially executed in the sequential stages, so the performance estimator estimates that the loops are not scalable. Although the performance estimation assumes that the benchmarks show performance speedup on a small number of cores, they suffer from the slowdown because the communication and validation overheads overwhelm the performance profits.

Unlike the performance estimation, `syrk[A]` shows performance speedup. One reason is that the benchmark has longer execution time for large input sets than the expected one. In addition, the loop is parallelized with Spec-DOALL scheme without a sequential stage, so the performance speedup is not limited by Amdahl's Law.

For the second class of benchmarks such as `reg_detect[A]`, inter-node communication bandwidth limits the performance. Due to imprecise static dependence analyzer, the ASAP compiler speculatively parallelizes the loop, and inserts validation codes that require communication between processes. However, the ASAP compiler cannot fully optimize the validation codes because of non-uniform memory access patterns, thus requiring huge communication bandwidth for validation beyond hardware supports.

6.1.2 Manually Parallelized Benchmarks

Since the ASAP compiler has limited applicability in making parallelization strategy, this dissertation executes additional evaluation for the ASAP runtime system with manually parallelized programs. Although the programs are manually parallelized, the parallelization method is the same as the ASAP compiler's.

`052.alvinn[M]`: The parallelized loop is at the second level in a loop nest. The runtime system initializes all the processes with data from the master at the beginning of each invocation of the parallelized loop. In addition, the runtime system delivers all live-out data from the processes to the master at the end of each invocation. These synchronizations from initialization and finalization limit the speedup.

`130.li[M]`: The parallelization speculates that each script is independent of the others and does not change the interpreter's environment nor cause the interpreter to exit altogether. Accesses to the memory state corresponding to the interpreter's environment are executed transactionally. Control flow speculation is used to break the dependences from the program exit condition.

`164.gzip[M]`: Compression works in three stages: 1) read block from the input file, 2) compress block in parallel, and 3) write compressed block. `164.gzip[M]` uses a variable block size, with the starting point of the next block being known only after the current block is compressed. This dependence prevents parallelization. The Y-branch [17] is used to break the dependence, and new blocks are started at fixed intervals. Multiple versions of the arrays used for holding the blocks are automatically created by the runtime system. Speedup is limited by communication bandwidth.

`179.art[M]`: The execution times of iterations in the parallelized loop are highly unbalanced due to the varying trip count of the inner loops. A round-robin assignment of iterations to each worker process results in wasted execution time due to the imbalance. To address this, the first stage distributes work based on queue occupancy as a proxy for load on each parallel-stage process.

197.parser[M]: The values of various global data structures are speculated to be reset at the end of each iteration and control flow speculation for error cases is applied. An entire dictionary must be copied from the master process on access by the worker processes, and sentences must be transferred from the first stage to later stages. As a result, the communication bandwidth becomes a performance bottleneck as the number of processes increases beyond 32.

256.bzip2[M]: Like 164.gzip[M], the second stage compresses blocks in parallel. Unlike 164.gzip[M], the Y-branch is not necessary because the block size is known in the first stage. The runtime system creates multiple versions of the block array. Control flow paths to handle error conditions are speculated as not taken.

456.hmmer[M]: The first stage calculates scores in parallel, while the second stage computes a histogram of the scores sequentially. Max-reduction is applied to compute the maximum score, and commutativity annotation is applied to random number generation.

464.h264ref[M]: Groups of Pictures (GoPs) are encoded in parallel. Dynamic memory versioning enabled by the runtime system breaks false memory dependences in the parallel stage and allows the parallel encoding of GoPs. The source and destination of the synchronized dependences are inside an inner loop. DSWP moves the dependence cycle to a separate stage allowing other stages to execute without waiting. Speedup is limited primarily by the number of GoPs available.

crc32[M]: On a cluster with a network file system, the original program spends most of its execution time reading the files. To reduce this effect, block read is used instead of character read by replacing getc with fread. The program is speculatively parallelized assuming errors do not occur in the CRC computation. Its speedup is limited by the number of input files.

blackscholes[M]: The hottest loop prints error messages if a computed price is different from its reference price. Speculation is applied on the error condition assuming the computed price is close to the reference price in the input file.

swaptions[M]: As in blackscholes[M], the outermost loop is parallelized with speculation on an error condition during price calculation. Object lifetime speculation is applied to the dynamically allocated matrices and vectors in each iteration. Scalability is limited by the input size.

The ASAP systems achieves scalable performance on most benchmarks in the evaluation. The performance improvement comes from a synergistic combination of parallel resources, DSWP, and speculation. While clusters provide many processors, communication latency can be a barrier to good use of them. DSWP addresses this issue. Speculation increases the applicability of DSWP to more applications. The ASAP compiler parallelizes the sequential program combining these features, and the ASAP runtime systems enable the combination by supporting multi-threaded transactions on clusters.

6.2 Comparison of Automatic and Manual Parallelization

Automatic parallelization is an attractive alternative to the time-consuming manual parallelization. However, although the ASAP compiler automatically parallelizes various programs without any programmers' annotation, its applicability is still limited compared to the manual parallelization. This section analyzes the missing features of the ASAP compiler that cause limited applicability comparing to the manual parallelization, and demonstrates a path for the compiler to generate scalable programs for a wide range of programs.

Section 6.1 shows the performance speedups of automatically and manually parallelized programs. Among the manually parallelized programs, the ASAP compiler successfully parallelizes blackscholes and swaptions without any annotation, and achieves scalable performance speedups. However, the compiler fails to parallelize the other programs. This section analyzes how the compiler generates scalable parallel codes for blackscholes and swaptions, and why the compiler fails to parallelize the others. Many features simultaneously affect the failures, but this section is focused on the representative factors.

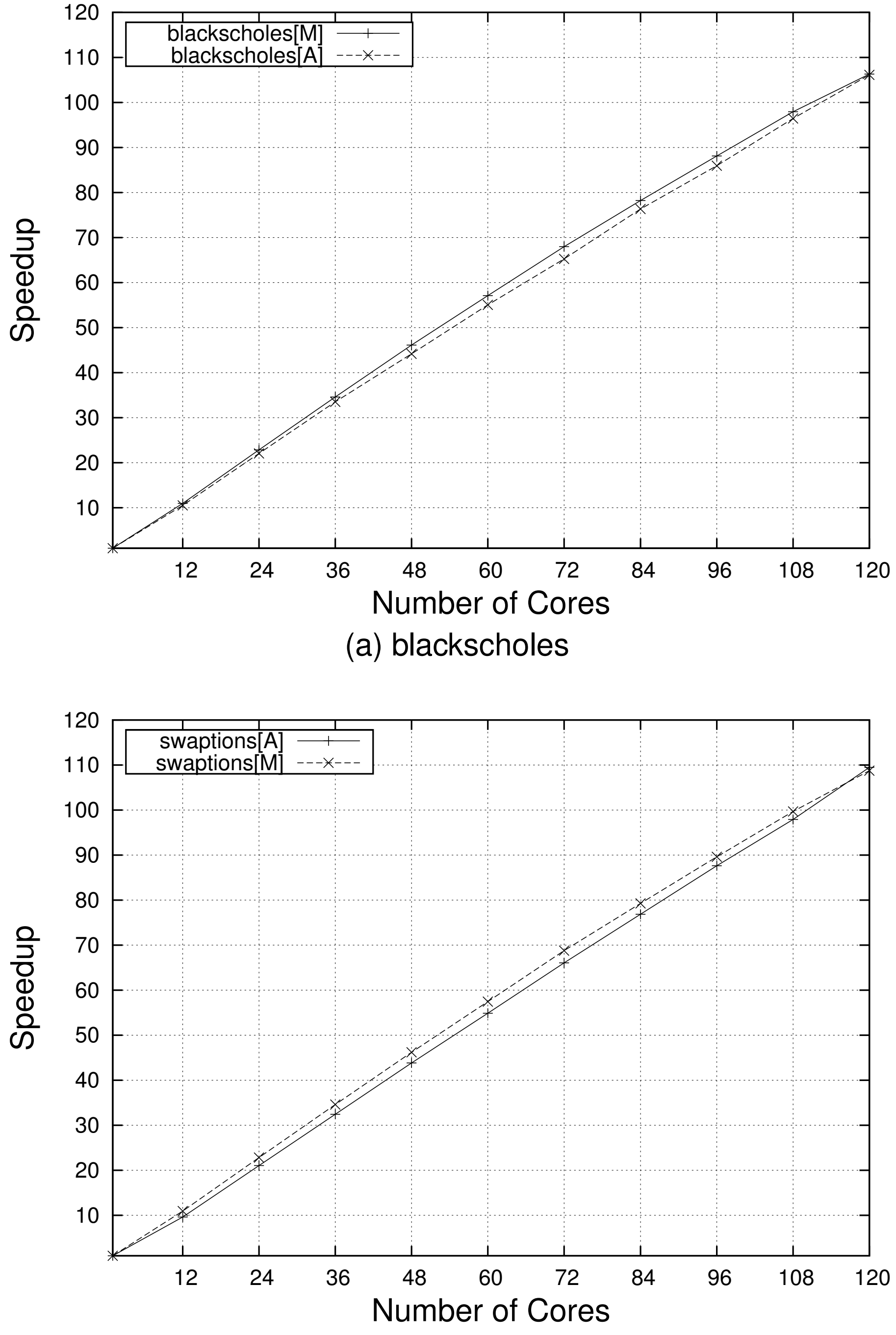
6.2.1 Scalable automatic parallelization

With the right parallelization strategy and speculation plan, the ASAP system can automatically generate speculative parallel codes as scalable as manual ones. Figure 6.4 shows that the performance speedups of automatically parallelized codes for `blackscholes` and `swaptions` are scalable and very close to the manual ones.

The main factor of the scalable speedups is that the ASAP compiler automatically makes the same parallelization strategy as the manual one. For example, Figure 6.5 shows the parallelization strategy that the compiler automatically generates for `swaptions`. Each box is a strongly connected component (SCC), the white box is a SCC in a parallel stage, and the light gray box is a SCC for induction variables that can be replicated to all the stages. Since all the iterations in the loops are in the white boxes or the gray boxes within one stage, the loop can be parallelized in Spec-DOALL scheme that is the same to the manual one.

The ASAP compiler parallelizes `blackscholes` with Spec-DOALL scheme while the manual parallelization uses Spec-DSWP scheme. The parallelized loop in `blackscholes` has an output operation, `printf`. The TXIO manager in the ASAP compiler removes inter-iteration dependences on the `printf`, and sequentially executes the output operation after commit, allowing Spec-DOALL parallelization. However, the manual parallelization allocates the output operation in the sequential stage, thus making a two-stage pipeline parallel loop. Although the automatic and manual parallelizations generate two different codes applying different parallelization schemes, the two codes execute in the same way.

In addition, as Table 6.2 describes, the ASAP compiler removes most loop-carried dependences with object lifetime speculation and read-only speculation that do not require communication between processes, so the compiler can overcome imprecise static alias analysis without paying huge validation and communication costs. The right parallelization strategy and the optimal speculation plan allow the ASAP compiler to automatically generate the same scalable parallel codes as timing-consuming manual parallelization.



(a) blackscholes

(b) swaptions

Figure 6.4: Performance comparison between automatic parallelization and manual parallelization for `blackscholes` and `swaptions`

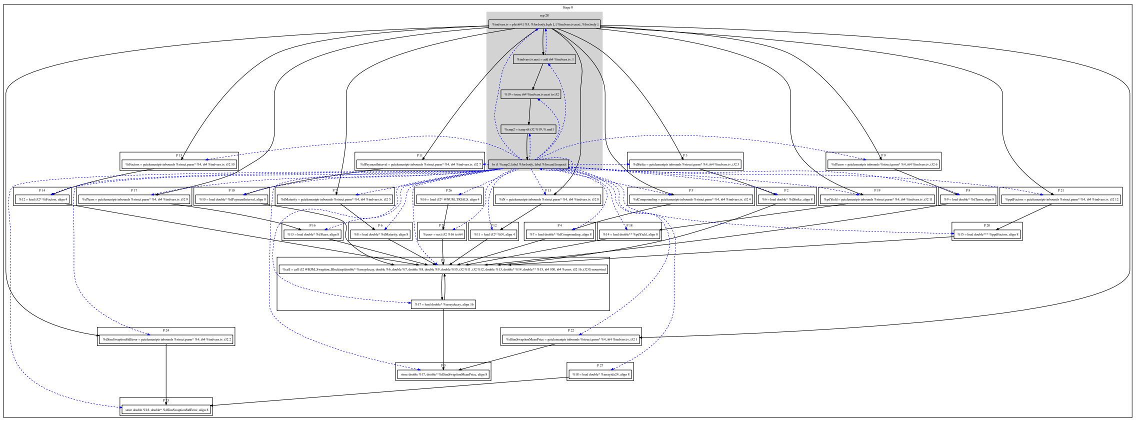

Stage 0
rep 28
%indvars.iv = phi i64 [%5, %for.body.lr.ph], [%indvars.iv.next, %for.body]
%indvars.iv.next = add i64 %indvars.iv, 1
%19 = trunc i64 %indvars.iv.next to i32
%cmp2 = icmp slt i32 %19, %.mul1
br i1 %cmp2, label %for.body, label %for.end.loopexit
P 15
%iFactors = getelementptr inbounds %struct.parm* %4, i64 %indvars.iv, i32 10
P 11
%dPaymentInterval = getelementptr inbounds %struct.parm* %4, i64 %indvars.iv, i32 7
P 3
%dStrike = getelementptr inbounds %struct.parm* %4, i64 %indvars.iv, i32 3
P 9
%dTenor = getelementptr inbounds %struct.parm* %4, i64 %indvars.iv, i32 6
P 14
%12 = load i32* %iFactors, align 4
P 17
%dYears = getelementptr inbounds %struct.parm* %4, i64 %indvars.iv, i32 9
P 10
%10 = load double* %dPaymentInterval, align 8
P 7
%dMaturity = getelementptr inbounds %struct.parm* %4, i64 %indvars.iv, i32 5
P 26
%16 = load i32* @NUM_TRIALS, align 4
P 13
%iN = getelementptr inbounds %struct.parm* %4, i64 %indvars.iv, i32 8
P 5
%dCompounding = getelementptr inbounds %struct.parm* %4, i64 %indvars.iv, i32 4
P 2
%6 = load double* %dStrike, align 8
P 19
%pdYield = getelementptr inbounds %struct.parm* %4, i64 %indvars.iv, i32 11
P 8
%9 = load double* %dTenor, align 8
P 21
%ppdFactors = getelementptr inbounds %struct.parm* %4, i64 %indvars.iv, i32 12
P 16
%13 = load double* %dYears, align 8
P 6
%8 = load double* %dMaturity, align 8
P 25
%conv = sext i32 %16 to i64
P 12
%11 = load i32* %iN, align 4
P 4
%7 = load double* %dCompounding, align 8
P 18
%14 = load double** %pdYield, align 8
P 20
%15 = load double*** %ppdFactors, align 8
P 1
%call = call i32 @HJM_Swaption_Blocking(double* %arraydecay, double %6, double %7, double %8, double %9, double %10, i32 %11, i32 %12, double %13, double* %14, double** %15, i64 100, i64 %conv, i32 16, i32 0) nounwind
%17 = load double* %arraydecay, align 16
P 24
%dSimSwaptionStdError = getelementptr inbounds %struct.parm* %4, i64 %indvars.iv, i32 2
P 22
%dSimSwaptionMeanPrice = getelementptr inbounds %struct.parm* %4, i64 %indvars.iv, i32 1
P 0
store double %17, double* %dSimSwaptionMeanPrice, align 8
P 27
%18 = load double* %arrayidx24, align 8
P 23
store double %18, double* %dSimSwaptionStdError, align 8

Figure 6.5: Parallelization strategy for `swaptions[A]`. The white boxes are strongly connected components (SCCs) that do not have any loop-carried dependence, and the light gray box is a SCC that can be replicated across all the threads. Each line between boxes means dependences, and blue dashed lines are loop-carried dependences.

6.2.2 Imprecise performance estimation

Imperfect performance estimation may lead the ASAP compiler to parallelize an outer loop in an inefficient way, missing profitable parallelism opportunities in inner loops. Since parallelizing an outer loop is generally more profitable than its inner loops due to invocation overheads, the compiler tries to parallelize from the outermost loops to their inner loops. If the performance estimator considers a loop parallelization strategy to be profitable, the compiler stops parallelizing its inner loops that may lead better performance improvement.

For example, Figure 6.6 shows the outermost hot loop in `052.alvinn`. The hot loop calculates inputs and outputs using weights that are updated in `update_weights()` at the end of each iteration, so the loop has loop-carried memory flow dependences on the weights that affect most of the loop instructions. The ASAP compiler parallelizes the outermost loop with Spec-DSWP scheme respecting the loop-carried flow dependences, and generates a parallel loop with a large sequential stage and a small parallel stage. Figure 6.7 illustrates the pipeline strategy. The dark gray box means a SCC with loop-carried dependences that cannot be executed in parallel, so most of the instructions in the loop are allocated to the sequential stage. There are only small number of instructions in the white

```
for (epoch = 0; epoch < NUM_EPOCHS; epoch++) {
  error = 0.0;

  for (pattern = 0; pattern < NUM_PATTERNS; pattern++) {
    input_hidden(next_input[pattern], hidden_act);
    hidden_output(hidden_act, output_act);
    update_stats(next_output[pattern], output_act, &error);
    output_hidden(next_output[pattern], output_act, hidden_act);
    hidden_input(next_input[pattern]);
  }

  update_weights();
  printf("EPOCH NUMBER %d: ERROR = %.5f\n",
    epoch+1, error / (NUM_PATTERNS *NOU));
}
```

Figure 6.6: The outermost hot loop in `052.alvinn`

boxes that will be executed in parallel. Since the performance estimator considers the parallel codes to improve the performance though it is very slight improvement, the ASAP compiler stops parallelizing its inner loops.

Unlike the automatic parallelization, the manual parallelization parallelizes the inner loop of the hot loop in `052.alvinn`. The outermost loop has an inner loop that takes most of the execution time, and each iteration in the inner loop is independent each other because all the inputs and outputs are indexed by `pattern`. The manual parallelization parallelizes the inner loop in Spec-DOALL scheme, avoiding the loop-carried dependences in the outer loop. Compared to the outer loop parallelization that allows only a small number of instructions to be executed in parallel, the inner loop parallelization allows all the iterations in the loop to be independently executed, leading better performance improvement.

To compare the automatic parallelization to the manual one, the ASAP compiler is forced to skip the outer loop parallelization. The compiler correctly finds and parallelizes the inner loop with DOALL scheme. Here, since the compiler does not support reduction, the parallelized program is slightly modified to support the reduction operations in

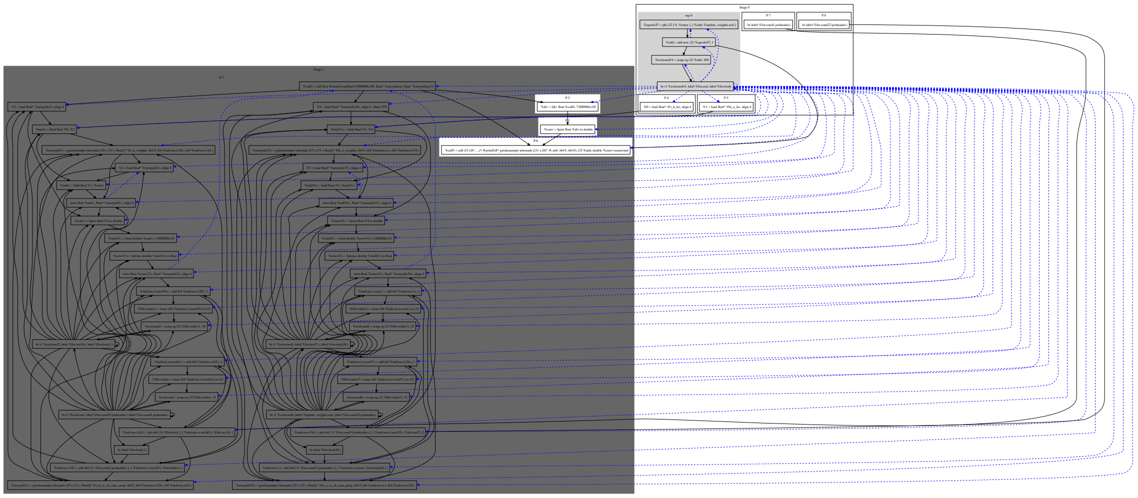


Figure 6.7: Pipeline strategy for `052.alvinn[A]`. All the boxes and lines have the same meaning to Figure 6.5. The dark gray box means a SCC that has at least one loop-carried dependence, so cannot be executed in parallel.

the loop. Figure 6.8 shows performance speedups of the manually and automatically parallelized codes. Although the ASAP compiler improves the performance of the program, the achieved performance speedup is still limited because of repeated high invocation overheads and reduced coverage from the excluded reduction instructions. Unlike the automatic parallelization, the manual parallelization invokes the parallel codes only once at the outer loop, delivers the updated weights from the master process to the worker at each outer loop iteration, and reduces the invocation overheads of the inner loop parallelization on the clusters. With light invocation overheads on shared memory system, Johnson et al. achieve the scalable performance improvement on `052.alvinn` up to 12 times on 24 cores with a fully automatic parallelization technique [38]

6.2.3 User annotation

Since the compiler cannot change the semantics of a program, the ASAP compiler parallelizes the sequential codes conservatively respecting the original algorithm. However, programmers can understand the semantic of the algorithm, and changes the codes during the manual parallelization. This difference allows the programmers to parallelize programs

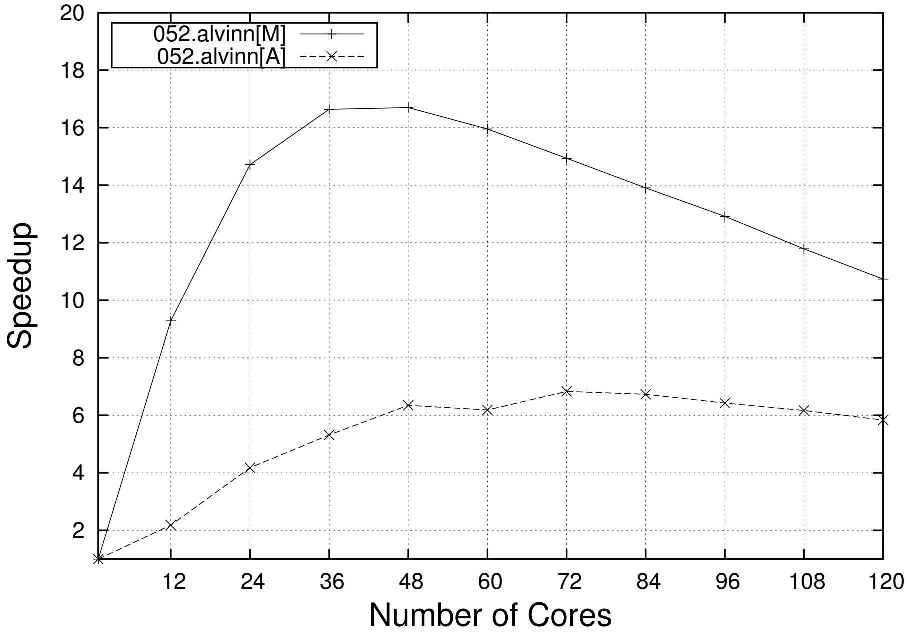


Figure 6.8: Performance comparison between automatic parallelization and manual parallelization for `052.alvinn` after modification

```
while(--argc > 0) {
  errors |= crc32file(*++argv, &crc, &charcnt);
  printf("%08X %7d %s\n", crc, charcnt, *argv);
}
```

Figure 6.9: The outermost hot loop in `crc32`

in more flexible ways than the compiler, so the manually parallelized codes can achieve better performance speedups than the automatically parallelized ones.

For example, `164.gzip` dynamically decides block sizes based on the compression results to maximize compression ratio, so it is impossible to compress later blocks in advance. Since the dynamic decision limits parallelism opportunities, programmers change the codes to partition the block in advance, enduring slightly reduced compression ratio, and parallelize the modified program. However, the ASAP compiler cannot change the original semantics, so fails to parallelize the program.

Figure 6.9 shows another example in `crc32`. The function, `crc32file`, reads an input file, calculates a CRC value, and checks an error. Due to the file access that may

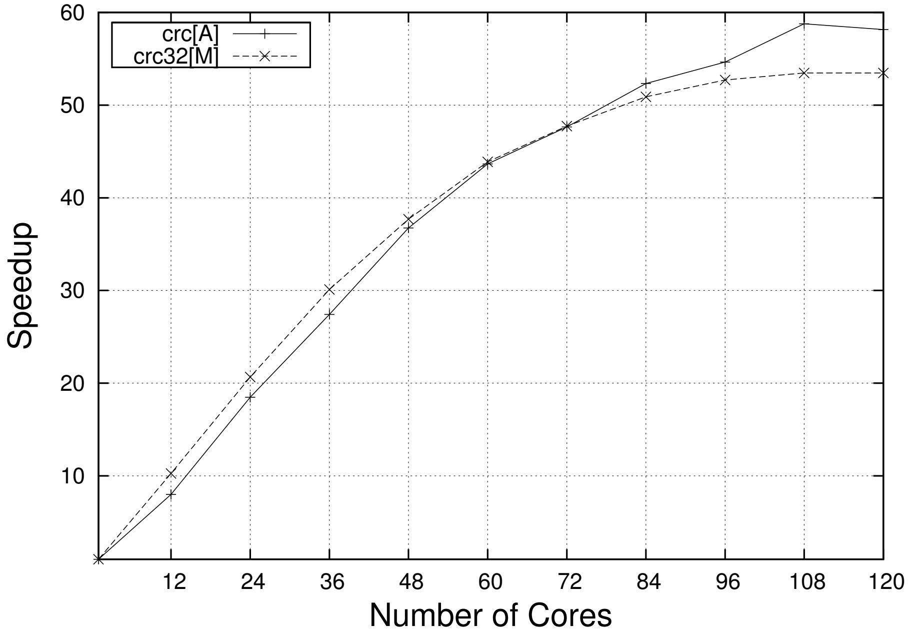


Figure 6.10: Performance comparison between automatic parallelization with annotations and manual parallelization for `crc32`

cause side effects, the ASAP compiler conservatively considers all the iterations to depend on the previous iterations, so it cannot parallelize the loop. However, programmers know what are input files, assume all the input files are not related, and freely parallelize the loop. For the same reason, the ASAP compiler cannot parallelize `456.hmmer` that has loop-carried dependences on a random number generator. However, programmers can allow the program to reorder the random number generation, and parallelize the program removing the loop-carried dependences.

To parallelize the programs, the compiler needs permission to modify the algorithm. One way to give the permission to the compiler is using user annotations. Bridges et al. proposes two user annotations such as Y-branch and commutativity [17]. Y-branch annotation forces a program to branch a target address if the annotation is executed more than a pre-defined threshold. For example, the annotation forces the loop in `164.gzip` to jump to the loop header breaking the input blocks, so the program can compress the next block independently. Commutativity annotation [17, 62] allows reordering of program execution,

```
int main(int argc,char **argv) {
  ...
  if (setjmp(cntxt.c_jmpbuf) == 0)
    for (i = 1; i < argc; i++)
      if (!xlload(argv[i],TRUE,FALSE))
        xlfail("can't load file");
  ...
}

void xljump(CONTEXT *cptr,int type,NODE *val)
{
  ...
  longjmp(xlcontext->c_jmpbuf,type);
  ...
}
```

Figure 6.11: `setjmp` and `longjmp` in `130.li`

so it breaks loop-carried dependence on an instruction or function call. For example, the annotation allows the random number generation reordered.

To evaluate the effect of the annotations, this dissertation implements a function level commutativity annotation that breaks loop-carried dependences on a function call, and applies the annotation to `crc32file` in `crc32`. The annotation makes the ASAP compiler parallelize `crc32` with two-stage Spec-DSWP scheme; the first parallel stage concurrently executes `crc32file` assuming all the input files are independent, and the second sequential stage updates `errors`. The parallelization plan is the same to the manually parallelized codes. Figure 6.10 shows the performance speedups of the automatically and manually parallelized codes. The automatically parallelized one performs very close to the manual version. This result shows that the annotation can effectively increase the parallelism opportunities, and again that the ASAP compiler can achieve scalable performance similar to manual parallelization if the compiler makes the right parallelization plan.

6.2.4 setjmp and longjmp

The ASAP compiler largely relies on the profiling results when it parallelizes a program, from finding hot loops to speculatively removing dependences. Although the compiler can parallelize a loop without profiling results, its applicability is limited to simple structure that can be easily analyzed, so the profiling results are crucial for the compiler to parallelize complex programs. However, profilers such as the loop aware memory profiler, the loop profiler and the speculative privatization profiler fail to generate profiling results for some programs due to `setjmp` and `longjmp`. For example, Figure 6.11 shows a hot loop and parts of benchmark `130.li`. The program calls `setjmp` before executing the hot loop, and `longjmp` in the hot loop. The profilers track program behavior building a function call stack and a loop nest stack. If the program pointer jumps to outside of a function or a loop, the profilers lose their tracking information, thus failing to generate correct profiling results.

Annotation can solve the problem. If all the `longjmp`s in a loop do not jump to outside of an iteration, the jumps do not affect execution of other iterations, so all the iterations can be independently executed. Assuming annotations on the boundary of `longjmp`, the manual parallelization parallelizes the hot loop in `130.li`, and achieves scalable performance improvement. Speculation on the boundary of `longjump` can solve the problem in the same way, but with additional validation algorithm that checks the boundary of the jump.

6.2.5 Interprocedural partitioning

The ASAP compiler may parallelize a loop that include a function call. If the target function read or write memory spaces, the compiler needs to generate a program dependence graph (PDG) including the instructions in the function. There are two ways to manage the dependence graph.

First, the compiler generates a PDG including all the instructions in target functions. If an instruction `inst_A` in a function `func_A` writes a memory space, and another instruc-

```
while (True) {
  blockNo++;
  initialiseCRC ();
  loadAndRLEsource ( stream );
  if (last == -1) break;

  blockCRC = getFinalCRC ();
  combinedCRC = (combinedCRC << 1) | (combinedCRC >> 31);
  combinedCRC ˆ= blockCRC;
  doReversibleTransformation ();
  bsPutUChar ( 0x31 ); bsPutUChar ( 0x41 );
  bsPutUChar ( 0x59 ); bsPutUChar ( 0x26 );
  bsPutUChar ( 0x53 ); bsPutUChar ( 0x59 );
  bsPutUInt32 ( blockCRC );

  if (blockRandomised) {
    bsW(1,1); nBlocksRandomised++;
  } else
    bsW(1,0);
  moveToFrontCodeAndSend ();
}

void moveToFrontCodeAndSend ( void ) {
  bsPutIntVS ( 24, origPtr );
  generateMTFValues();
  sendMTFValues();
}
```

Figure 6.12: Hot loop in `256.bzip2`

tion inst_B in a function func_B reads the same memory, the compiler directly inserts a flow dependence from inst_A to inst_B. This way can illustrate the all the dependence information in detail. However, if a program becomes complex, the compiler may suffer from memory explosion because the number of edges quadratically increases to the number of instructions.

The other way is making a function call in the loop represent all the instructions in the target function. For the previous example, the compiler inserts a flow dependence from a call instruction call_A to the other call instruction call_B instead of the two instructions in the functions. Since the method does not increase the number of instructions in a PDG, the compiler can avoid the memory explosion. However, since the dependence information is abstracted, the method loses some parallelism opportunities.

To avoid memory explosion, the ASAP compiler adopts the second method, but it suffers from limited applicability due to the abstracted PDG. For example, Figure 6.12 shows a hot loop in 256.bzip2. Each iteration in the loop reads a block from input files, compresses the block, and writes the results to an output file. Though reading and writing input and output files cannot be parallelized, since each iteration can independently compress each block, the manual parallelization partitions the loop into three stages; a sequential read stage, a parallel compression stage, and a sequential write stage. Here, the manual parallelization splits moveToFrontCodeAndSend, and allocates generateMTFValues to the parallel compression stage and bsPutIntVS and sendMTFValues to the sequential write stage. However, the ASAP compiler considers the call to moveToFrontCodeAndSend as a single instruction, so allocates the call instruction to a sequential stage. Since a larger function has more chance to include an instruction with loop-carried dependences, the abstracted PDG makes the ASAP compiler difficult to parallelize complex programs. Here, the abstracted PDG does not harm Spec-DOALL scheme because all the instructions in the Spec-DOALL loop do not have any loop-carried dependence, and the abstracted PDG does not lose any information.

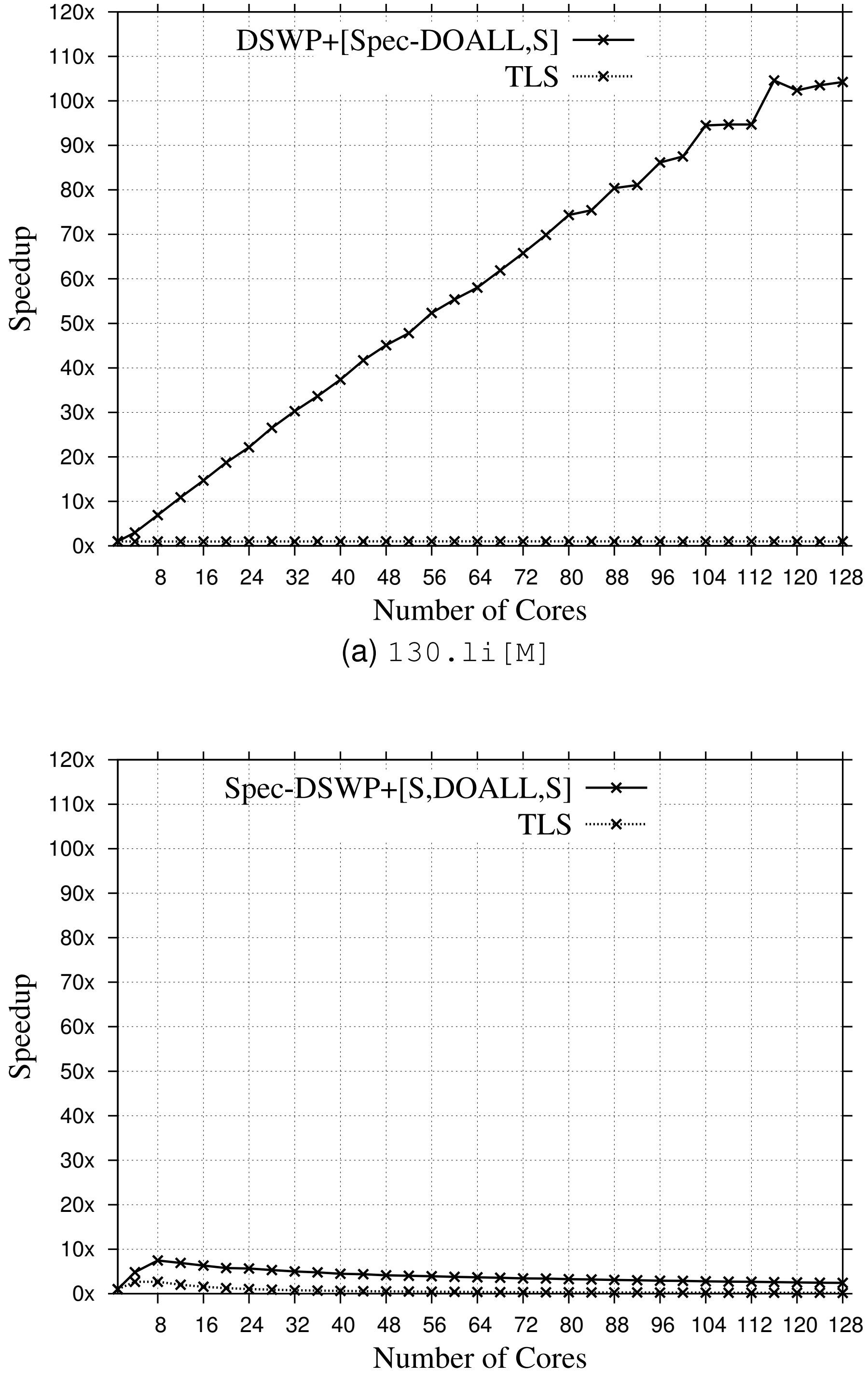
6.2.6 Global variable localization

Some programs such as `164.gzip` and `256.bzip2` have a lot of global variables that are used in multiple functions. Although the global variables may make programmers easily write a program, they make the ASAP compiler generate inefficient parallel codes even with the right parallelization strategy. For example, the hot loop in Figure 6.12 updates a global variable `last` that points the end of a read block. Although `last` is not accessed after the hot loop, since it is a global variable which lifetime scope is the whole program, the compiler conservatively delivers every update of `last` to the master process as a live-out variable. Since the hot loop updates a lot of global variables that are not accessed after the hot loop, the automatically parallelized codes waste communication bandwidth to deliver unnecessary data, thus harming the performance. Unlike the ASAP compiler, the programmers only deliver live-out values that are really used in later execution, so the manually parallelized codes can achieve performance speedup.

6.3 Comparison of DSWP and TLS

As mentioned in Section 1.1, DSWP has higher latency tolerance than TLS by keeping the communication pattern unidirectional. This section evaluates and compares the performance of TLS and DSWP on a 128-core cluster (32 nodes × 4 cores/node). Each node is a Dell PowerEdge 1950 with two dual-core processors (Intel Xeon 5160 @ 3.00GHz) and 16GB of RAM. The same loops in each benchmark are manually parallelized and optimized with DSWP and TLS schemes.

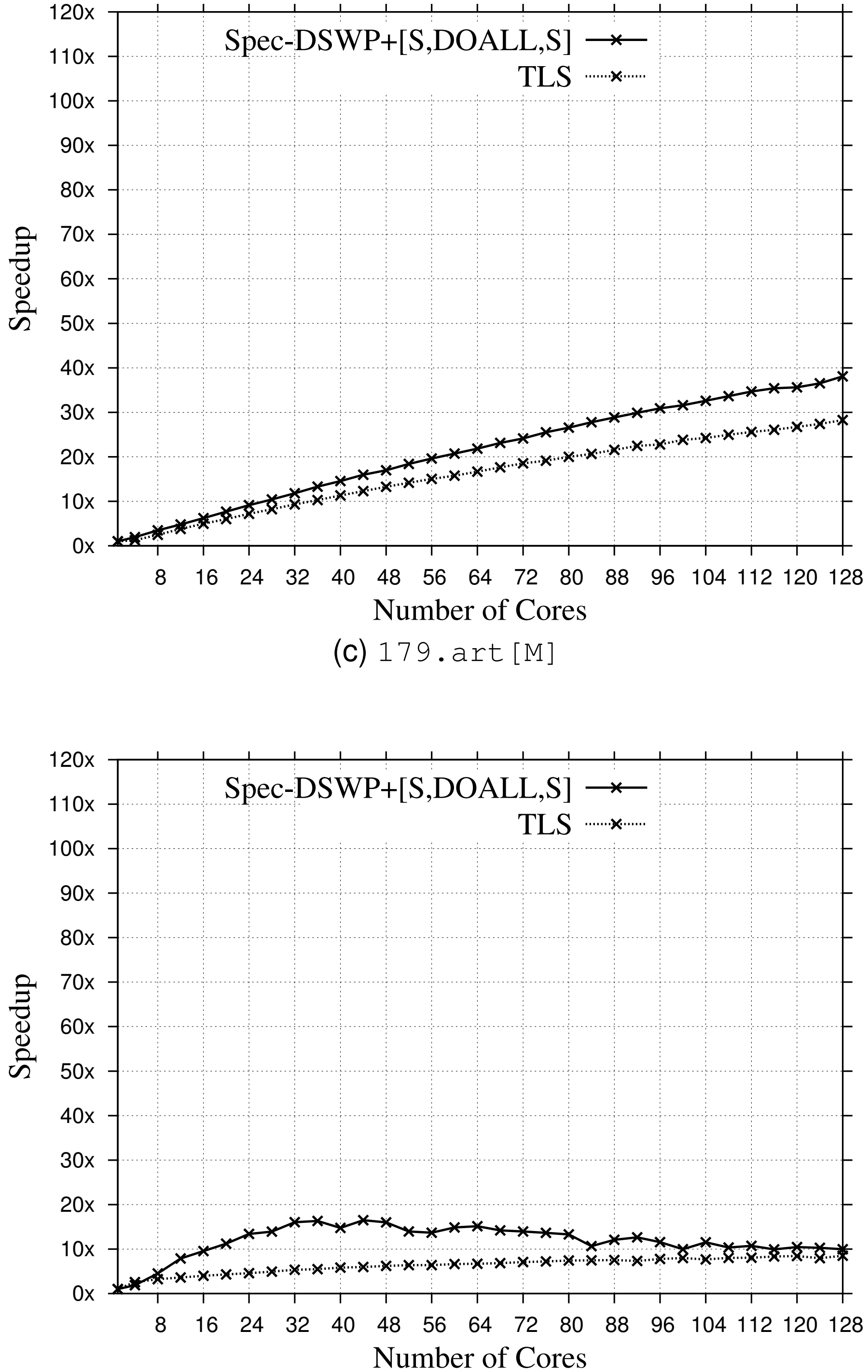
Figure 6.13 shows that DSWP generally scales better than TLS as analyzed in Section 1.1. The *DSWP+[...]* notation describes how the programs are parallelized in the Spec-DSWP scheme. Within square brackets, parallelization techniques applied to each stage are specified. For example, Spec-DSWP+[S, DOALL, S] means three stage pipeline parallelism. Here, *S* indicates a stage that is sequentially executed, whereas *Spec-* indicates



(a) `130.li[M]`

(b) `164.gzip[M]`

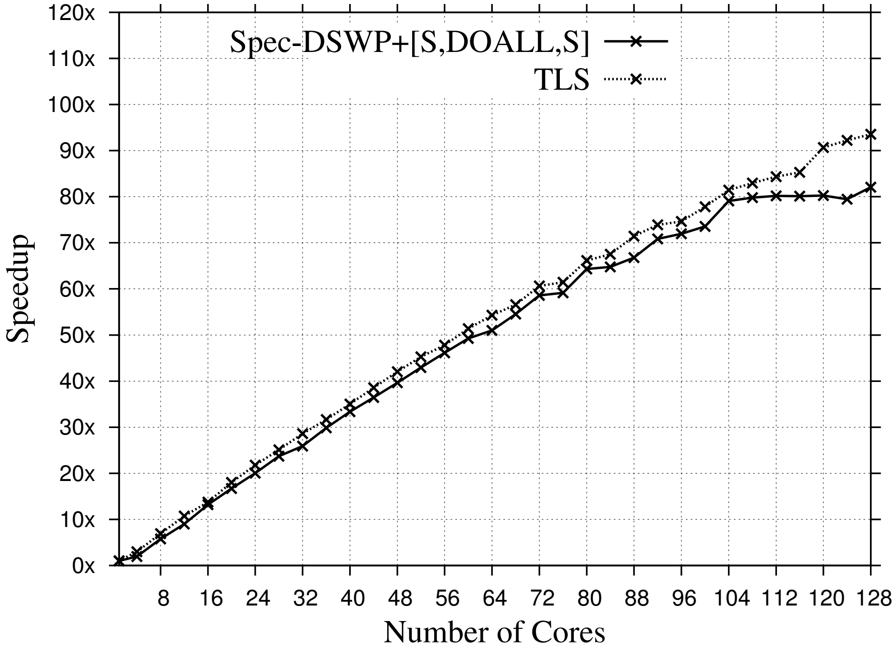
Figure 6.13: Performance speedups of Spec-DSWP and TLS using the ASAP runtime



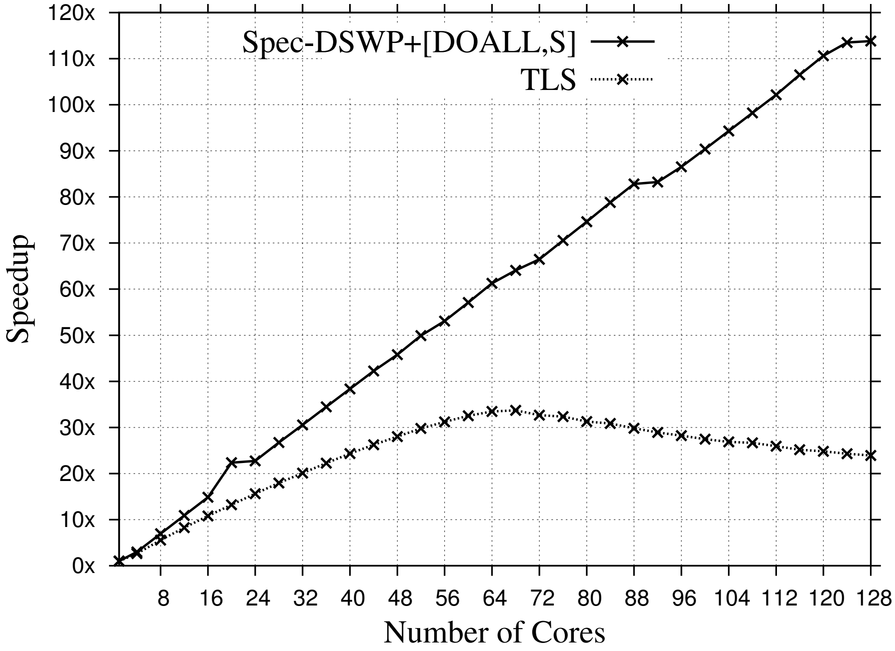
(c) `179.art[M]`

(d) `197.parser[M]`

Figure 6.13: Performance speedups of Spec-DSWP and TLS using the ASAP runtime

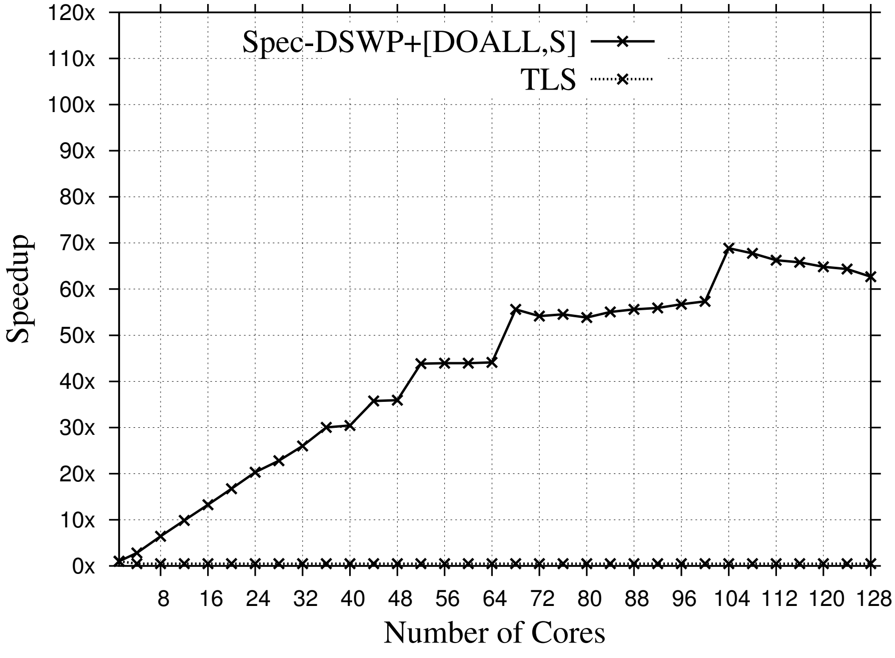


(e) 256.bzip2[M]

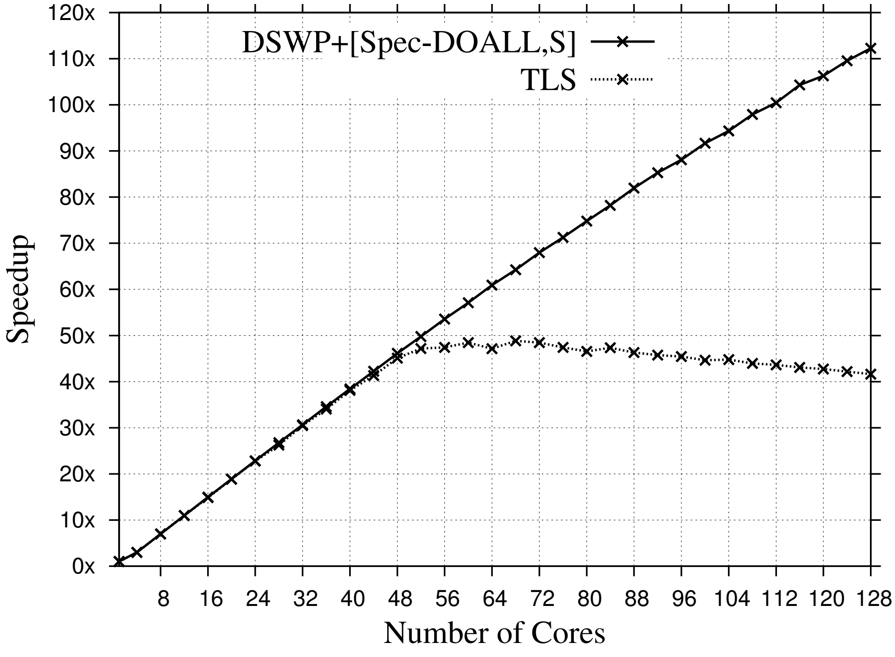


(f) 456.hmmer[M]

Figure 6.13: Performance speedups of Spec-DSWP and TLS using the ASAP runtime

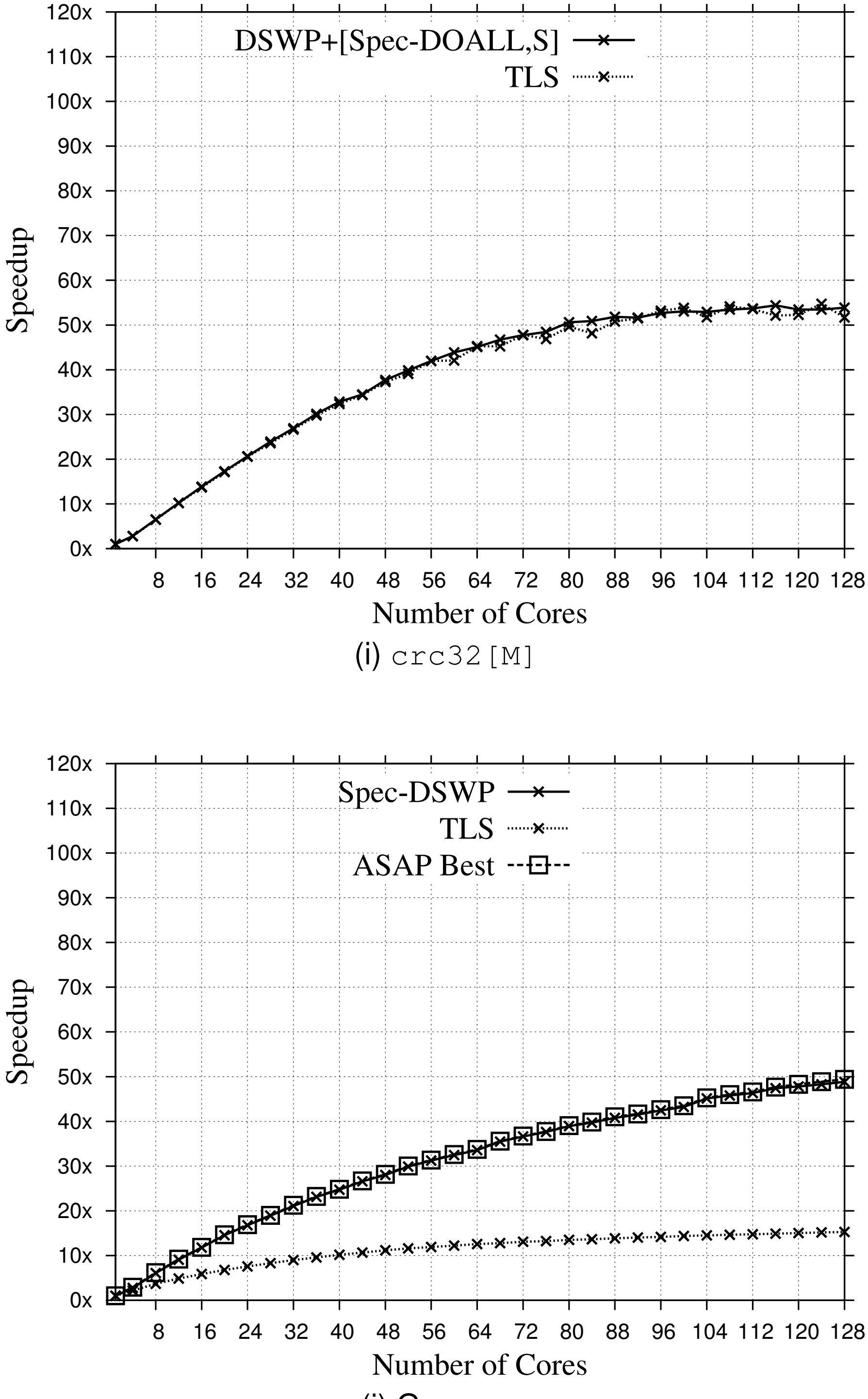


(g) 464.h264ref[M]



(h) blackscholes[M]

Figure 6.13: Performance speedups of Spec-DSWP and TLS using the ASAP runtime



(i) `crc32[M]`

(j) Geomean

Figure 6.13: Performance speedups of Spec-DSWP and TLS using the ASAP runtime

speculation between stages. *Spec-* in DSWP+[S, Spec-DOALL, S] means that speculation is applied only to the second stage.

DSWP scales to a higher core count than TLS especially in `179.art`, `456.hmmer`, and `blackscholes`. It is because the cyclic dependence of TLS puts inter-process communication latency on the critical path of program execution, which becomes the bottleneck as the number of processes increases.

In `130.li` and `464.h264ref`, additional synchronization points limit the performance speedups of TLS. For example, each iteration in the hot loop of `130.li` executes output operations as `printf` that should be synchronized to achieve correct output. While DSWP allocates the output operations in a separate sequential stage, TLS inserts synchronization between processes, so the performance speedup is limited.

Unlike other benchmarks, TLS shows better performance than DSWP for `256.bzip2`, because DSWP sends the whole input file to other processes in later stages while TLS sends only the file descriptor of the input file to others. In this case, communication bandwidth is the main factor that affects the performance, so TLS shows slightly better performance than DSWP.

In summary, for the eight manually parallelized benchmarks, while TLS shows 16.46× geomean speedup, DSWP shows 47.96× geomean speedup with better scalability.

6.4 Recovery Overhead

Figure 6.14 shows how different misspeculation rates affect the performance speedup of `blackscholes[A]` on different numbers of cores. The input files are modified to cause misspeculation with varying rates from 0.01% to 0.64%. Higher misspeculation rate and more cores generally lead to greater performance penalties. The misspeculation overhead is more sensitive to the misspeculation rate than the number of cores because additional misspeculation causes a new recovery operation while synchronization overhead from ad-

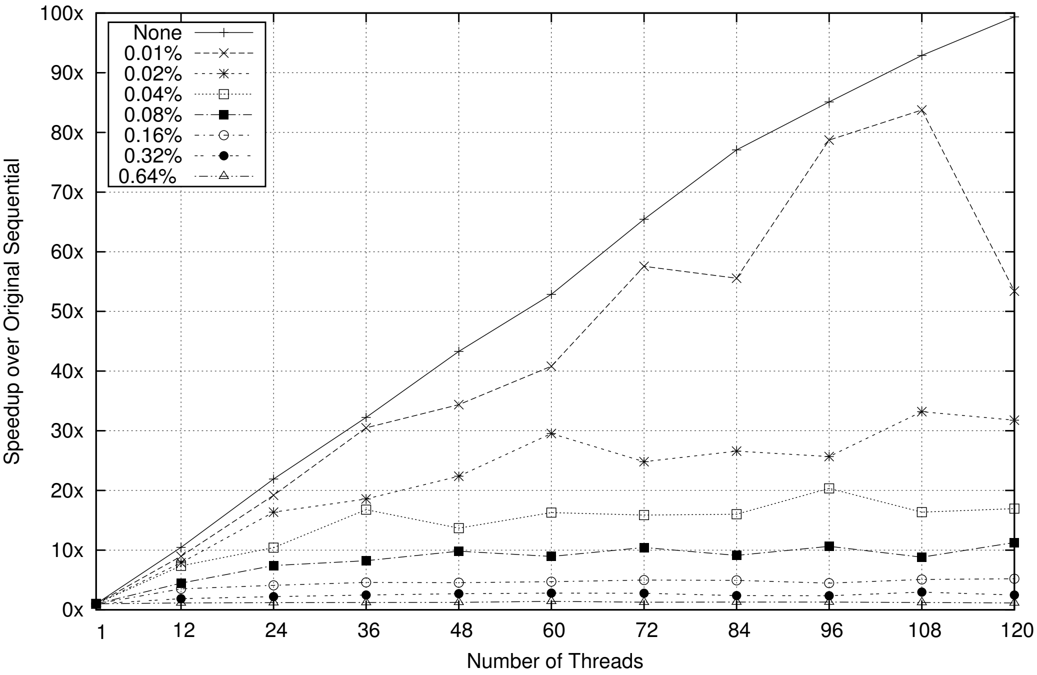


Figure 6.14: Speedup of `blackscholes[A]` with varying misspeculation rates

ditional cores is overlapped with the existing one. Due to the high recovery overhead, the ASAP compiler should speculate dependences only with high confidence to achieve good parallel performance.

6.5 Communication Overhead

Communication overhead is one of the critical features that limit performance improvement of parallel programs on clusters. Although acyclic parallel programs are tolerant of communication latency, their performance speedups are limited to communication bandwidth. Figure 6.15 presents bandwidth requirements for each parallel program. Bandwidth is computed by dividing the total data transferred via the runtime system by the total execution time. To show how the bandwidth requirement increases as more cores are used, data is presented for three consecutive core counts starting from the number of pipeline stages in the parallelization.

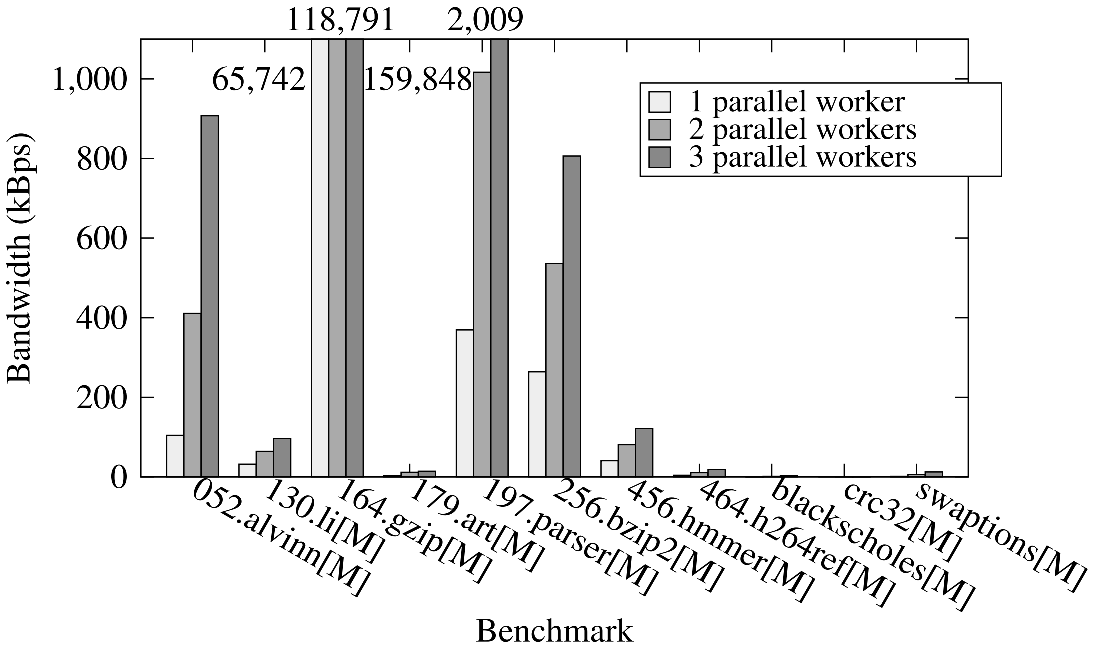


Figure 6.15: Bandwidth requirement for each application

The figure shows that the required communication bandwidth increases as more cores are used. It is because the execution times become shorter and shorter as the number of cores increases while the total amount of data communication remains or increases. It explains the plateauing of the speedups of `052.alvinn[M]` and `197.parser[M]`. In both programs, as the number of processes increases, the application bandwidth increases much faster when compared to the other programs. At a large number of processes, the bandwidth requirements limit the speedup.

`164.gzip[M]` has very high bandwidth requirements that grow as the number of processes is increased, explaining its limited scalability. Interestingly, the amount of data transferred by `256.bzip2[M]` is similar to `164.gzip[M]`; however, the amount of computation in `256.bzip2[M]` is much more resulting in longer execution time and much lower bandwidth. This explains the vast difference in their performance improvements.

Although DSWP is communication latency tolerant, it is sensitive to the overhead of the operations required to send a datum [67]. Since a single invocation of a send or receive function can take as many as 2,295 instructions in OpenMPI [18], the ASAP runtime system coalesces multiple data transfer requests, thereby amortizing the costs. Communication using the queues in the runtime system can sustain a bandwidth of 480.7 MBps, whereas communication using `MPI_Send`, `MPI_Bsend`, or `MPI_Isend` directly provides 13.1, 12.7 and 8.1 MBps of bandwidth respectively.

6.6 Tiling Optimization

Tiling [43, 55, 69] is a well-known optimization to improve kernel applications with regular memory access such as matrix multiplication. Combined with the optimization, the ASAP system can achieve synergistic effects in performance speedup. Among the programs listed in Table 6.1, `2mm`, `3mm` and `gemm` are applicable of the tiling optimization. This thesis manually applies tiling optimization to these programs, and the ASAP system automatically parallelizes the tiled programs. Figure 6.16 compares performance speedups of tiled and non-tiled, sequential and parallel programs. The best performance speedup results are chosen if parallel programs are not scalable up to 120 cores. The tiling optimization improves the performance of the sequential execution, and the ASAP system achieves additional performance speedups on the tiled program. Since the ASAP system parallelizes the outer-most loops of the program, all the parallel threads execute tiled sequential codes, so the synergistic performance improvement can be achieved.

6.7 Energy Consumption

Energy efficiency becomes a major technology issue in computer systems not only for mobile devices and embedded systems, but also for servers and data centers. At first glance, the runtime system might seem energy inefficient because parallel execution requires additional resources such as cores and memory. This section analyzes the energy consumption of the runtime system on the shared memory system.

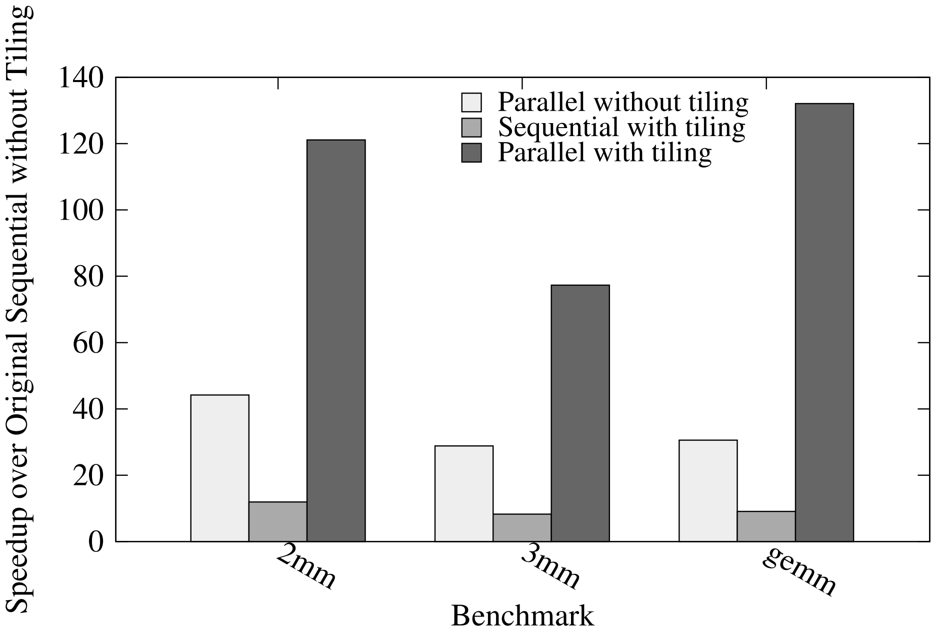
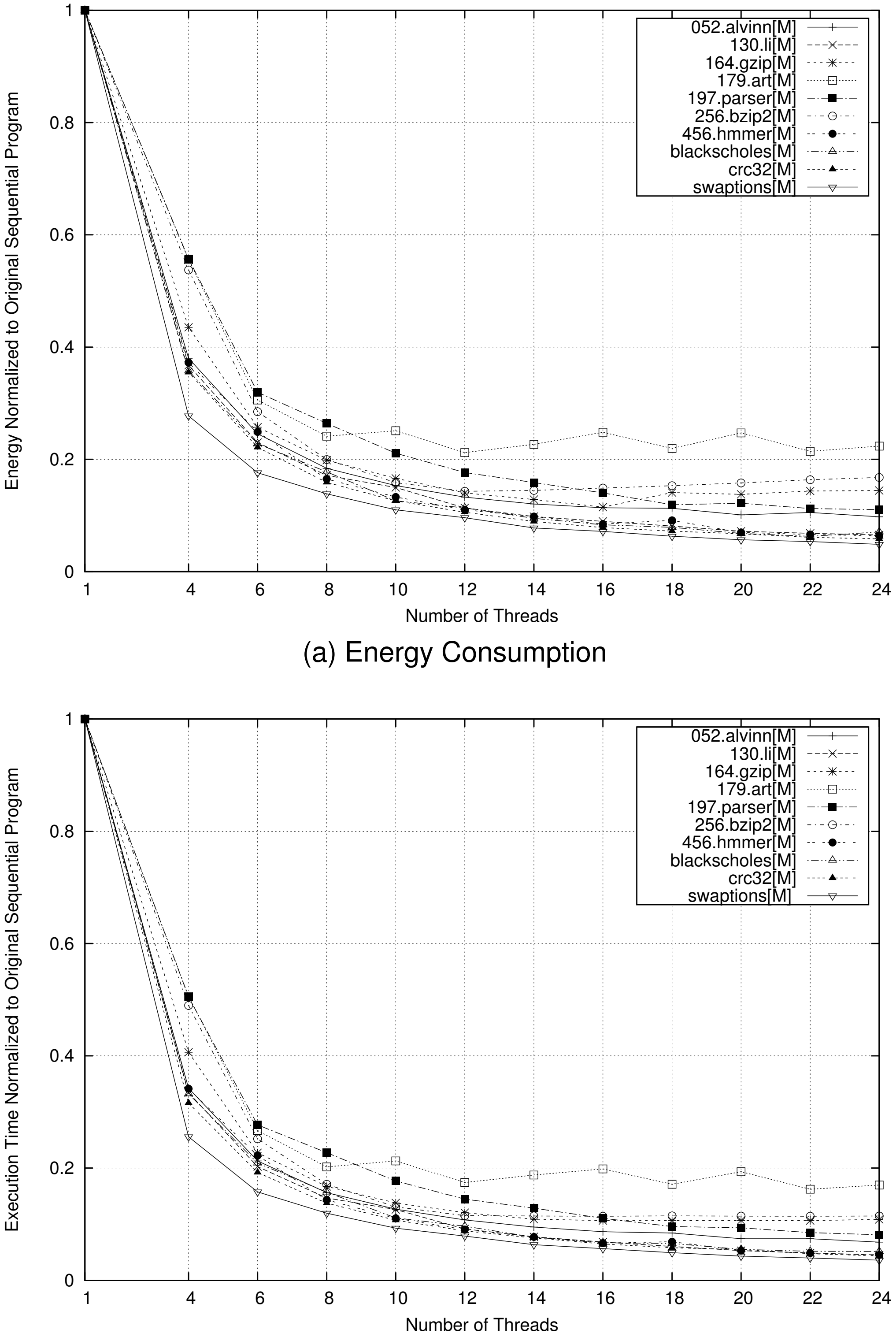


Figure 6.16: Performance comparison between tiled and non-tiled, sequential and parallel programs

Since there is no power meter in the cluster machines, a shared memory machine is used for the energy performance evaluation. The machine has four Intel 6-core Xeon X7460 processor running at 2.66 GHz with 24GB of memory. It runs 64-bit Ubuntu 9.10. Full system power is measured at the maximum sampling rate (13 samples per minute) supported by the power distribution unit (AP7892 [8]). The energy consumption in Figure 6.17 is normalized to the original sequential program.

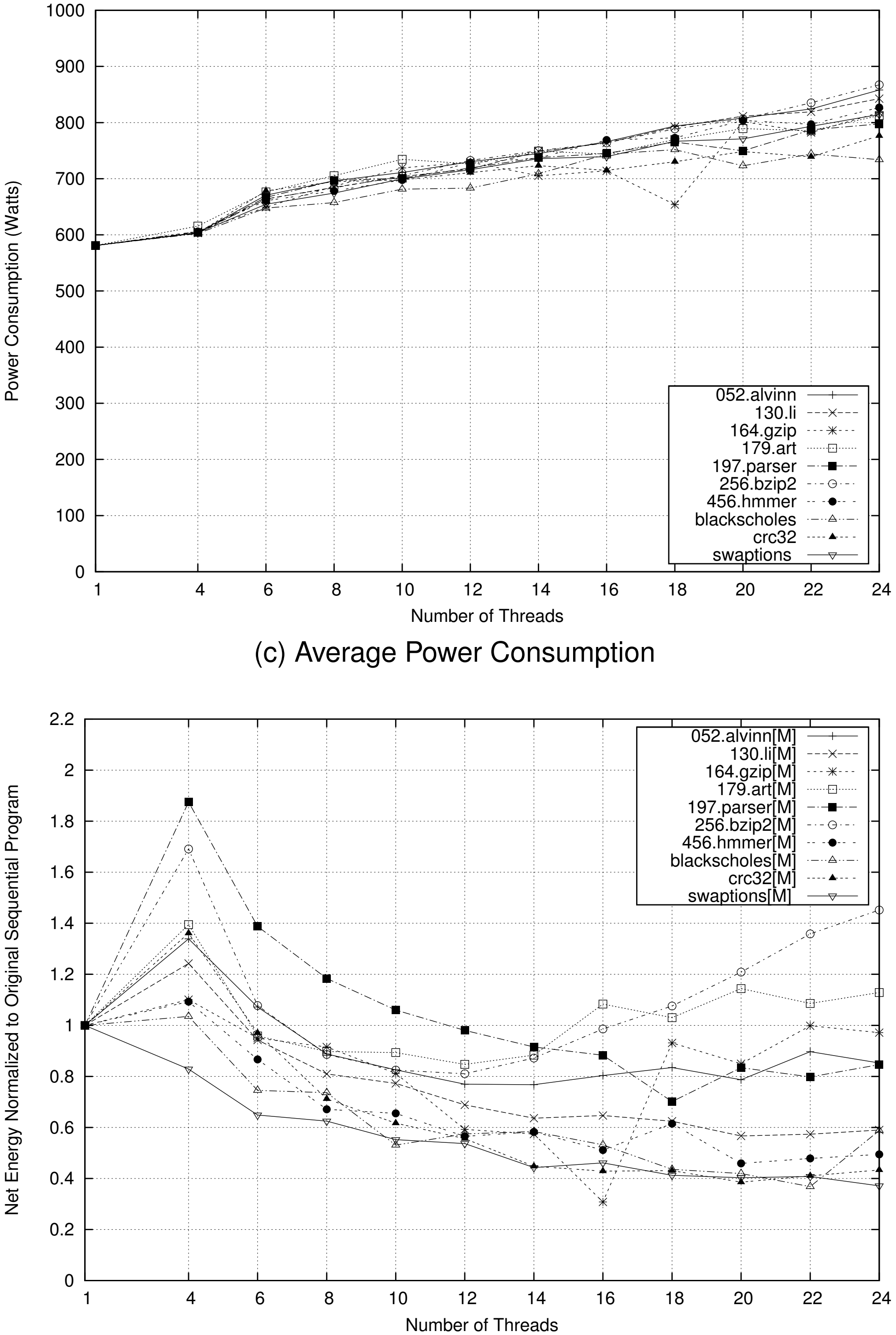
Figure 6.17(a) shows how much energy the runtime system consumes when it executes each benchmark. Unlike the expectation, the runtime system consumes less energy when it uses more processes. Although the runtime system causes additional power consumption for parallel processes, it reduces execution time enough to save energy. To analyze the energy consumption patterns in detail, this thesis measured the execution time and average power consumption. The energy consumption follows a very similar trend line with execu-



(a) Energy Consumption

(b) Execution Time

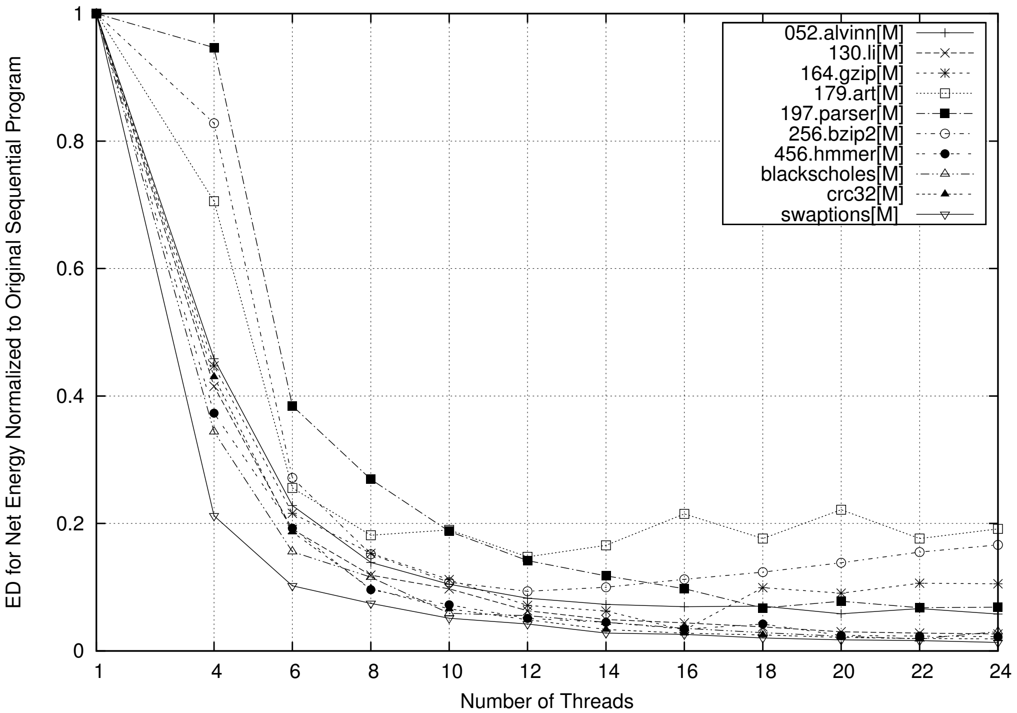
Figure 6.17: Energy consumption of speculative parallel execution



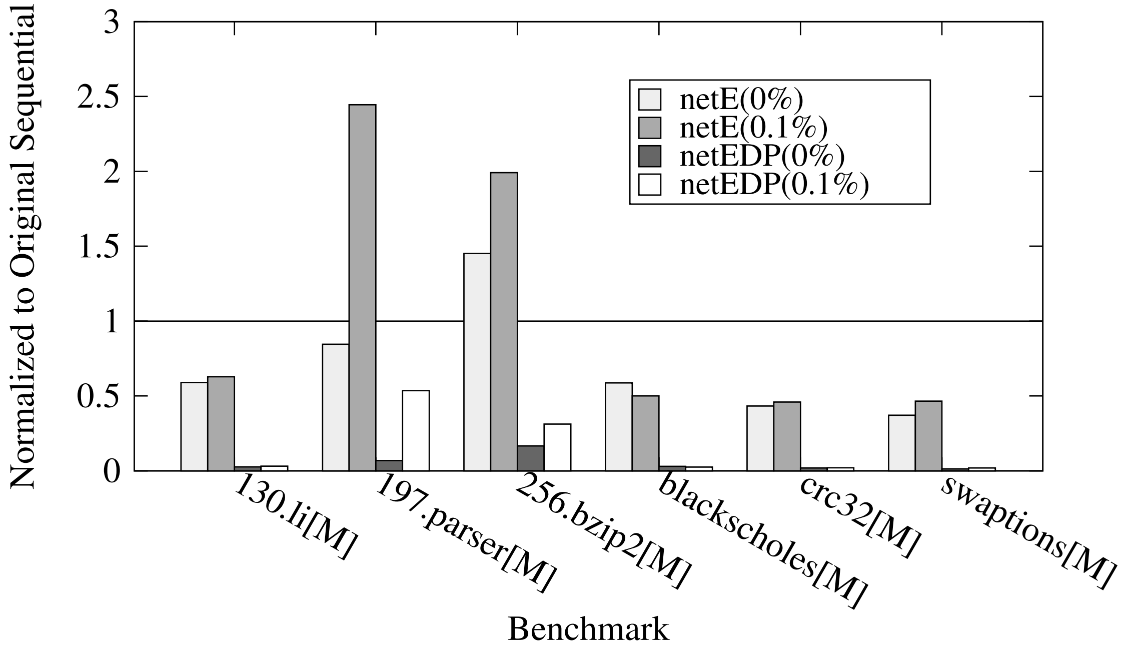
(c) Average Power Consumption

(d) Net Energy Consumption (netE)

Figure 6.17: Energy consumption of speculative parallel execution



(e) Net Energy Delay Product (netEDP)



(f) netE and netEDP with 0.1% Misspeculation

Figure 6.17: Energy consumption of speculative parallel execution

tion time in Figure 6.17(b), which decreases as the number of threads increases. The reason can be found in Figure 6.17(c), average power consumption. Servers consume power even in idle time. Figure 6.17(c) shows the idle power takes a large portion of overall power consumption, so increase in power consumption due to additional thread allocation is relatively modest. Hence, execution time plays a dominant role in calculating energy consumption, which allows the runtime system to save total energy consumption.

If a server is fully energy proportional assuming there is no static power consumption, the energy consumption patterns become like Figure 6.17(d). In the ideal machine, reduced execution time does not guarantee saving energy any more because there is no idle power. However, even in the ideal machine, the runtime system does not consume energy more than 1.9×, while it can save energy up to 70%. The system consumes energy a lot with four threads because only two worker processes contribute the performance speedup. The other two processes are the validator and the master that manage speculation without contributing performance speedup. On the ideal machine, there is an energy optimal point for each benchmark. By manipulating the number of threads, energy can be saved. Beyond the energy optimal point, one trades energy consumption for lower delay.

For a system that has the ability to trade energy for performance, Gonzalez and Horowitz propose a metric, Energy-Delay Product (EDP), which aims to balance energy consumption and execution time (delay) [30]. Figure 6.17(e) shows that the runtime system improves EDP for all benchmarks with higher thread counts. Note that net energy is used for EDP calculation.

Even with misspeculation, the runtime system can save net energy. Figure 6.17(f) shows net energy and EDP on 24 cores with 0% and 0.1% misspeculation rates. Although the energy consumption increases if misspeculation occurs, the runtime system still saves net energy, and shows better EDP numbers (<1) than the original sequential execution. `197.parser[M]` and `256.bzip2[M]` increases net energy larger than the other benchmarks because they have larger SEQ overheads during recovery.

Chapter 7

Conclusion

Parallelization is a crucial key to increase performance of programs on parallel platforms such as clusters. Automatic parallelization is an attractive solution that alleviates the programmer's efforts on manual parallelization. This dissertation proposes the ASAP system that automatically parallelizes sequential programs for commodity clusters and achieves scalable performance improvement.

The ASAP system is the first fully automatic speculative acyclic parallelization system for commodity clusters. The ASAP system consists of the ASAP compiler and the ASAP runtime system. With synergistic combination of automatic parallelization, speculation and acyclic parallelization, the ASAP compiler transforms sequential programs to scalable parallel programs. The ASAP runtime system executes the speculative parallel programs on commodity clusters without any hardware support.

The ASAP compiler is the first compiler that implements speculative decoupled software pipelining (Spec-DSWP) for clusters. The compiler transforms the loops to parallel codes inserting communication primitives for flow dependences across parallel contexts. The acyclic communication pattern of Spec-DSWP scheme makes the parallel codes tolerant of high communication latency on clusters. In addition, the compiler adopts different speculation techniques such as control flow speculation and object lifetime speculation that do not require communication for validation. The compiler also batches and promotes explicit communication, and reduces the communication overheads. The optimization allows the parallel program to achieve scalable performance improvement.

The ASAP runtime system is the first transactional memory system that supports multi-threaded transactions (MTXs) without any hardware supports. According to Spec-DSWP scheme, the ASAP compiler splits a loop iteration (a transaction) into multiple pipeline stages (sub-transactions) across multiple threads, making MTXs. The ASAP runtime system implements the MTX concept, and executes the Spec-DSWP codes on commodity clusters. To overcome high communication costs of clusters, the runtime system optimizes communication in MTX operations.

With the synergistic combination with the ASAP compiler and the ASAP runtime system, the ASAP system achieves a geomean speedup of 8.91× with 109× maximum performance speedup on a 120-core cluster. In addition, this dissertation manually parallelizes 11 sequential C programs with the ASAP system, and achieves a geomean speedup of 39.52× with 110.6× maximum performance speedup. Comparing the ASAP system with the manual parallelization, this dissertation demonstrates a path for the ASAP system to achieve scalable performance improvement for general-purpose applications as future work.

Bibliography

[1] The OpenMP API specification. http://www.openmp.org.

[2] K. Agrawal, J. T. Fineman, and J. Sukha. Nested parallelism in transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[3] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, 1987.

[4] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[5] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, 1993.

[6] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[7] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1991.

[8] APC metered rack PDU user's guide. http://www.apc.com.

[9] J. a. Barreto, A. Dragojević, P. Ferreira, R. Guerraoui, and M. Kapalka. Leveraging parallel nesting in transactional memory. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[10] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, 2002.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[12] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The next generation in parallelizing compilers. In *Proceedings of the workshop on Languages and Compilers for Parallel Computing*, 1994.

[13] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008.

[14] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, 2008.

[15] P. Boulet, A. Darte, G.-A. Silber, and F. Vivien. Loop parallelization algorithms: from parallelism extraction to code generation. *Journal of Parallel Computing*, 24(3-4):421–444, 1998.

[16] P. Boulet and M. Dion. Code generation in Bouclettes. In *Proceedings of the Fifth Euromicro Workshop of Parallel and Distributed Processing*, 1997.

[17] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[18] D. Buntinas, G. Mercier, and W. Gropp. Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the nemesis communication subsystem. *Parallel Computing, North-Holland*, 33(9):634–644, 2007.

[19] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks. HELIX: automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012.

[20] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, Sept. 2008.

[21] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2005.

[22] J.-F. Collard. Code generation in automatic parallelizers. In *Proceedings of the IFIP WG10.3 Working Conference on Applications in Parallel and Distributed Computing*, 1994.

[23] J.-F. Collard, T. Risset, and P. Feautrier. Construction of DO loops from systems of affine constraints. *Parallel Processing Letters*, 05(03):421–436, 1995.

[24] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. In *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, 2009.

[25] A. Darte and F. Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, 1996.

[26] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared-Memory Programming*. John Wiley and Sons, 2005.

[27] P. Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, 1992.

[28] P. Feautrier. Some efficient solutions to the affine scheduling problem: II. multi-dimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[29] M. I. Frank. *SUDS: Automatic Parallelization for Raw Processors*. PhD thesis, MIT, 2003.

[30] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *Solid-State Circuits, IEEE Journal of*, 31(9):1277 –1284, 1996.

[31] V. Gramoli, R. Guerraoui, and V. Trigonakis. TM2C: a software transactional memory for many-cores. In *Proceedings of the 7th ACM european conference on Computer Systems*, 2012.

[32] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, 1998.

[33] M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20(3):195–208, 2007.

[34] J. P. Hoeflinger. Extending OpenMP to clusters. *White Paper Intel Corporation*, 2006.

[35] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, 2010.

[36] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, and D. I. August. Decoupled software pipelining creates parallelization opportunities. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, 2010.

[37] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: an overview of the pips project. In *Proceedings of the 5th international conference on Supercomputing*, 1991.

[38] N. P. Johnson, H. Kim, P. Prabhu, A. Zaks, and D. I. August. Speculative separation for privatization and reductions. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012.

[39] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, 1995.

[40] H. Kim, N. P. Johnson, J. W. Lee, S. A. Mahlke, and D. I. August. Automatic speculative doall for clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012.

[41] H. Kim, A. Raman, F. Liu, J. W. Lee, and D. I. August. Scalable speculative parallelization on commodity clusters. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[42] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. DiSTM: A software transactional memory framework for clusters. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, 2008.

[43] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, 1991.

[44] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, 1974.

[45] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, 2004.

[46] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming*, 22:183–205, 1994.

[47] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th international conference on Supercomputing*, 1999.

[48] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997.

[49] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: a TLS compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2006.

[50] K. Y. Luigi, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. pages 10–11, 1998.

[51] K. Manassiev, M. Mihailescu, and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2006.

[52] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, 2009.

[53] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: log-based transactional memory. In *Proceedings of The Twelfth International Symposium on High-Performance Computer Architecture*, 2006.

[54] The message passing interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi/.

[55] J. J. Navarro, T. Juan, and T. Lang. Mob forms: a class of multilevel block algorithms for dense linear algebra operations. In *Proceedings of the 8th international conference on Supercomputing*, 1994.

[56] C. E. Oancea and A. Mycroft. Software thread-level speculation: an optimistic library implementation. In *Proceedings of the 1st international workshop on Multicore software engineering*, 2008.

[57] J. Oplinger, D. Heine, S. wei Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, Stanford University, Computer Systems Laboratory, February 1997.

[58] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[59] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, 2005.

[60] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 4th edition, 2008.

[61] L.-N. Pouchet. PolyBench: the Polyhedral Benchmark suite.
http://www-roc.inria.fr/ pouchet/software/polybench/download.

[62] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. Commutative set: a language extension for implicit parallel programming. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011.

[63] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992.

[64] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[65] H. E. Ramadan and E. Witchel. The Xfork in the road to coordinated sibling transactions. In *The Fourth ACM SIGPLAN Workshop on Transactional Computing*, 2009.

[66] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, 2010.

[67] R. Rangan, N. Vachharajani, A. Stoler, G. Ottoni, D. I. August, and G. Z. N. Cai. Support for high-frequency streaming in cmps. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[68] L. Rauchwerger and D. Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, 1995.

[69] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, 1999.

[70] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. *International Journal of Parallel Programming*, 31(4):251–283, 2003.

[71] M. M. Saad and B. Ravindran. HyFlow: a high performance distributed software transactional memory framework. In *Proceedings of the 20th international symposium on High performance distributed computing*, 2011.

[72] M. M. Saad and B. Ravindran. Snake: control flow distributed software transactional memory. In *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems*, 2011.

[73] Standard Performance Evaluation Corporation.
http://www.spec.org.

[74] Stanford Compiler Group. SUIF: A parallelizing and optimizing research compiler. Technical Report CSL-TR-94-620, Stanford University, Computer Systems Laboratory, May 1994.

[75] J. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, 1998.

[76] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005.

[77] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th annual international symposium on Computer architecture*, 2000.

[78] The Liberty Research Group, 2002.

[79] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[80] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, 2008.

[81] N. Vachharajani. *Intelligent Speculation for Pipelined Multithreading*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, November 2008.

[82] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[83] H. Volos, A. Welc, A.-R. Adl-Tabatabai, T. Shpeisman, X. Tian, and R. Narayanaswamy. NePalTM: design and implementation of nested parallelism for transactional memory systems. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2009.

[84] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, 1991.

[85] R. M. Yoo and H.-H. S. Lee. Helper transactions: Enabling thread-level speculation via a transactional memory system. In *PESPMA '08: Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures*, 2008.

[86] B. Zhang and B. Ravindran. Brief announcement: Relay: A cache-coherence protocol for distributed transactional memory. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, 2009.

[87] B. Zhang and B. Ravindran. Location-aware cache-coherence protocols for distributed transactional contention management in metric-space networks. In *Proceed-*

ings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems, 2009.

[88] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for speculative parallelization of partially-parallel loops in dsm multiprocessors. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, 1999.

[89] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *Proceedings of IEEE 14th International Symposium on High Performance Computer Architecture*, 2008.

[90] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, 2002.