UNIVERSITY OF CALIFORNIA
RIVERSIDE

The SpiceC Parallel Programming System

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Min Feng

September 2012

The Dissertation of Min Feng is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

This thesis would not have been possible without all people who have supported and inspired me during my doctoral study.

I own my deepest gratitude to my advisor, Dr. Rajiv Gupta, for his continuous support of my research and study. I always feel motivated by his perpetual enthusiasm in research and unshakable belief in my potential. He guided and inspired me to find and pursue my own ideas, which made research life enjoyable and rewarding for me. I could not have imagined having a better advisor for my doctoral study.

Dr. Laxmi N. Bhuyan and Dr. Iulian Neamtiu deserve a thank as my thesis committee members. I am grateful to them for their help in this dissertation. I would also like to thank them for their guidance and help in our joint work.

I would like to express my gratitude to all my lab buddies: Chen Tian, Dennis Jeffrey, Vijay Nagarajan, Changhui Lin, Yan Wang, Kishore K. Pusukuri, Li Tan, and Sai Charan for helping me in many ways during these years. In particular, I would like to thank Chen Tian for his help to get my research started.

Finally, I would like to thank my family, particularly my wife Yi Hu, and my parents Huiyuan Feng and Huixue Cai. Their unconditional support help me overcome all kinds of difficulties I encoutered during my doctoral study.

To my wife and parents for all the support.

ABSTRACT OF THE DISSERTATION

The SpiceC Parallel Programming System

by

Min Feng

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2012
Dr. Rajiv Gupta, Chairperson

As parallel systems become ubiquitous, exploiting parallelism becomes crucial for improving application performance. However, the complexities of developing parallel software are major challenges. Shared memory parallel programming models, such as OpenMP and Thread Building Blocks (TBBs), offer a single view of the memory thereby making parallel programming easier. However, they support limited forms of parallelism. Distributed memory programming models, such as the Message Passing Interface (MPI), support more parallelism types; however, their low level interfaces require great deal of programming effort.

This dissertation presents the SpiceC system that simplifies the task of parallel programming while supporting different forms of parallelism and parallel computing platforms. SpiceC provides easy to use directives to express different forms of parallelism, including DOALL, DOACROSS, and pipelining parallelism. SpiceC is based upon an intuitive computation model in which each thread performs its computation in isolation from other threads using its *private space* and communicates with other threads via the *shared* *space.* Since, all data transfers between shared and private spaces are explicit, SpiceC naturally supports both shared and distributed memory platforms with ease.

SpiceC is designed to handle the complexities of real world applications. The effectiveness of SpiceC is demonstrated both in terms of delivered performance and the ease of parallelization for applications with the following characteristics. Applications that cannot be statically parallelized due to presence of dependences, often contain large amounts of input dependent and dynamic data level parallelism. SpiceC supports *speculative parallelization* for exploiting dynamic parallelism with minimal programming effort. Applications that operate on large data sets often make extensive use of *pointer-based dynamic data structures.* SpiceC provides support for partitioning dynamic data structures across threads and then distributing the computation among the threads in a partition sensitive fashion. Finally, due to large input sizes, many applications repeatedly perform I/O operations that are interspersed with the computation. While traditional approach is to execute loops contain I/O operations serially, SpiceC introduces support for *parallelizing computations in the presence of I/O operations.*

Finally, this dissertation demonstrates that SpiceC can handle the challenges posed by the memory architectures of modern parallel computing platforms. The memory architecture impacts the manner in which data transfers between private and shared spaces are implemented. SpiceC does not place the the burden of data transfers on the programmer. Therefore portability of SpiceC to different platforms is achieved by simply modifying the handling of data transfers by the SpiceC compiler and runtime. First, it is shown how SpiceC can be targeted to shared memory architectures both with and without hardware support for cache coherence. Next it is shown how accelerators such as GPUs present in heterogeneous systems are exploited by SpiceC. Finally, the ability of SpiceC to exploit the scalability of a distributed-memory system, consisting of a cluster of multicore machines, is demonstrated.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallel systems are currently evolving towards massive parallelism and heterogeneity for performance and power efficiency. This trend has created exciting opportunities for achieving increased performance with multithreaded software. Meanwhile, the increasing complexity and variety of parallel systems has made it difficult to efficiently realize the benefits of their computing power via parallelism, as it has also raised many new challenges in software development.

Two types of parallel programming models have been proposed to facilitate the development of parallel applications for modern parallel systems: shared memory programming models; and distributed memory programming models. Existing systems based upon these models have their drawbacks that greatly limit their ability to deal with the complexity of developing multithreaded code for real world applications.

Existing shared memory programming models such as OpenMP [36] and Threading Building Blocks (TBB) [108] have been widely used because they offer a single view of the memory. Developers do not have to be aware of the location of data, thereby making parallel programming easier. However, these parallel programming models have limited applicability. First, they can only be applied to certain types of applications as they only support limited forms of parallelism and are unable to handle parallelism in the presence of popular programming features such as pointer-based linked dynamic data structures and I/O operations. Second, these programming models do not support distributed memory systems since they require a shared physical memory. *Although the shared memory programming models are easy to use, they cannot be adopted in many real applications due to the lack of support for various application types and platforms.*

Distributed memory programming models such as MPI [52], on the other hand, are more versatile. They do not require any physical shared memory and can be used to exploit more forms of parallelism. They are often used in large-scale applications that demand better scalability. However, the distributed memory programming models are hard to use since their interfaces tend to be low level. Developers need to spend significant effort to explicitly program distribution of workload among threads as well as communication and synchronization of threads. Moreover, to achieve efficient implementation of parallel applications, developers must understand the underlying hardware and undertake the error-prone task of programming optimizations [60]. *Even though the distributed memory programming models are more versatile, the programmability issue often prevents them from being used.*

This dissertation overcomes the above drawbacks by developing a parallel programming system that combines the programmability of the shared memory programming models and the versatility and portability of the distributed memory programming models. While developing such a system is challenging, its benefit in facilitating parallel programming for wide range of applications and platforms is clear. Developers can exploit parallelism while focusing on high-level algorithmic issues. They do not need to understand the underlying hardware to run their programs on different platforms.

1.1 Dissertation Overview

This dissertation presents the SpiceC parallel programming system that enables development of parallel software for various types of applications and platforms with ease. Figure 1.1 provides an overview of SpiceC. The key issues addressed by this system are briefly described next.

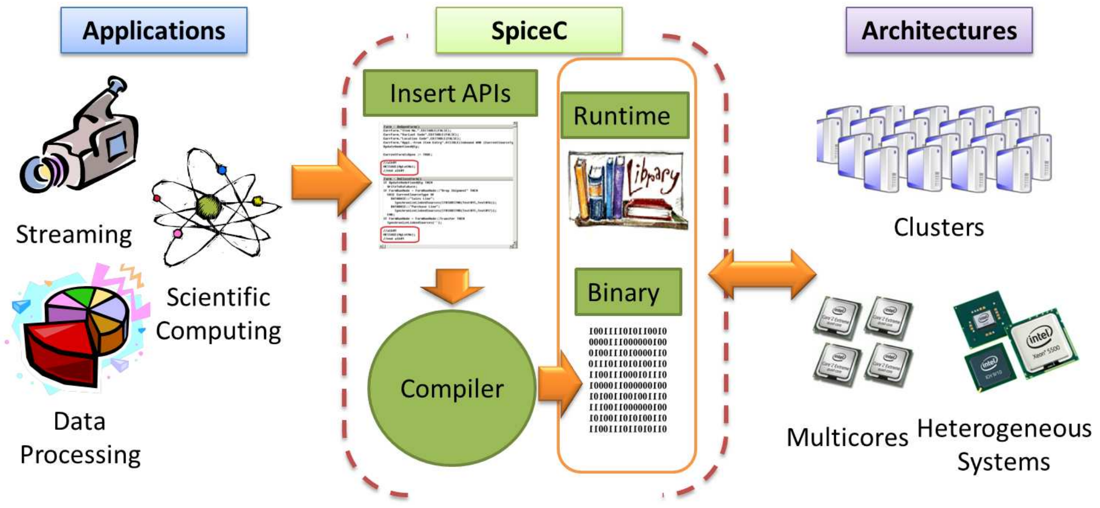


Figure 1.1: An overview of SpiceC.

1.1.1 SpiceC System for Multicores

As commercial processors transition from single-core to multicore, parallel programming becomes a challenge on commercial processors. To meet this challenge, programming systems should address the programmability, versatility, and portability issues of parallel programming on multicore processors.

Programmability. The complexity of parallel programming hinders its use by developers. To make programmers more productive, a parallel programming system should use high-level programming style. For example, instead of having the developers write new parallel programs, they could simply add parallelization primitives into the text of an existing sequential program. Parallelism can be exploited by the compiler based on the added primitives.

Versatility. To efficiently parallelize applications, a parallel programming system should support various forms of parallelism as different computations contain different forms of parallelism to exploit. Since the parallelism in many real applications cannot be guaranteed at compile time, speculative parallelization should be supported for dynamically exploiting the parallelism.

Performance portability. A parallel programming system should be able to adapt to different types of multiprocessors. As multiprocessors evolve towards manycores, the scalability of traditional cache coherence protocols, such as snoopy- or directory-based protocols, becomes a problem. Recently, Intel Labs has created their new Tera-scale multiprocessor – "Single-chip Cloud Computer (SCC)" which is a 48-core processor with no cache coherence support. Therefore, in the long term, a parallel programming model should support not only multicore processors with cache coherence but also manycore processors without cache coherence.

The core SpiceC system presented in this dissertation is designed to address the above parallel programming issues for multicore processors. It has a high-level programming model, which is easy to use and designed to exploit multiple forms of parallelism. The compiler translates SpiceC code into explicitly parallel code, exploiting parallelism based on the high-level programming primitives. Its computation model makes it portable across multiprocessors that may or may not support cache coherence.

1.1.2 Support for Dynamic Linked Data Structures

Recently, parallelization of computations in the presence of dynamic data structures has shown promise. Efficient parallel execution in the presence of large dynamic data structures requires exploiting data locality of and speculative parallelism.

Data locality. Data partitioning-based computation distribution [9] has been proposed to exploit data locality on distributed systems. The overall approach involves partitioning the data, assigning the partitions to threads, and finally assigning computations to threads such that the thread that owns the data required by a computation performs the computation. Programming the above strategy is not a trivial task. Developers need to write code for: partitioning and assigning the data to threads; and distributing computation among threads based on the partitioning. They also program required synchronizations between computations for different partitions.

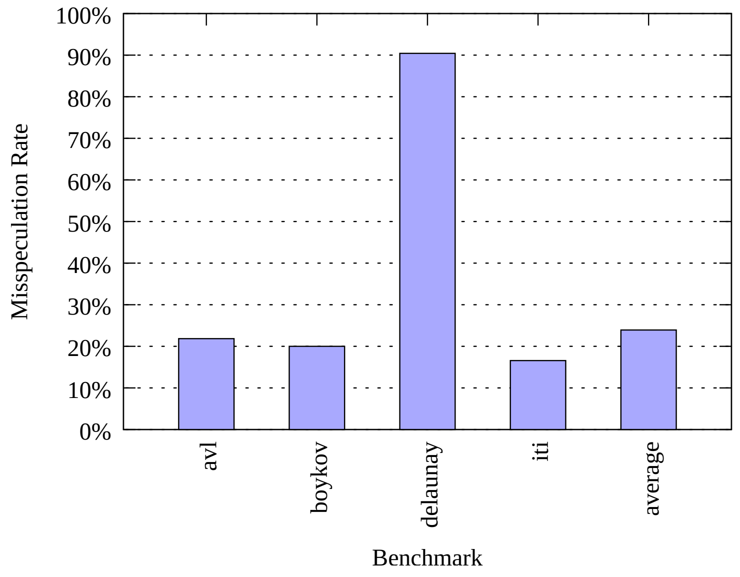


Figure 1.2: Misspeculation rates of four applications that use dynamic data structures.

Speculative parallelism. Mostly each thread works on its assigned partition; however, occasionally multiple threads need to simultaneously access the same data elements of a dynamic data structure. In such situations, the threads are designed to speculatively operate on same elements. However, if the parallel accesses are found to violate data dependences present in sequential version, then misspeculation is detected and the application must be designed to recover from them. It has been observed that the misspeculation rates of applications that use dynamic data structures can be very high. Figure 1.2 shows the misspeculation rates of four applications on a 8-core machine. Since these misspeculation rates are very high (exceeding 15% for all applications), very high recovery overhead will be incurred. To efficiently exploit speculative parallelism, it is necessary to optimize speculative execution to reduce the misspeculation overhead.

This dissertation presents the SpiceC support for dynamic data structures, which addresses both of the above issues.

1.1.3 Support for Parallelism in the Presence of I/O Operations

Existing parallel programming models only target loops that contain pure computations, i.e., they are free of I/O operations. Since applications that work on large data sets contain many computatation loops which also contain I/O operations, they fail to yield much speedup when these loops are executed serially. Therefore, it is highly desirable to support parallel programming models which allow parallel execution of *hybrid loops*, i.e., loops with both computation and I/O operations. Figure 1.3 shows the percentage of total execution time taken by the hybrid loops in eight applications on a 24-core machine. We can see that a significant portion of the total execution time is taken by the hybrid loops in all these applications. Therefore, to further improve the performance of these applications, developers must exploit the parallelism in hybrid loops.

The presence of I/O operations raises the following challenges for parallel programming. First, a hybrid loop has cross-iteration dependences caused by the I/O operations as the file read/write operations from different iterations move the same file pointer. Usually when the computation within a loop does not involve cross-iteration dependences, maximum parallelism can be exploited via DOALL parallelization where all loop iterations can be executed in parallel. However, in a hybrid loop, even when the computation does not involve cross-iteration dependences, DOALL parallelization is not possible due to the cross-iteration dependences caused by the I/O operations. Fully exploiting the parallelism in the loop requires a way to break the cross-iteration dependences due to I/O operations.

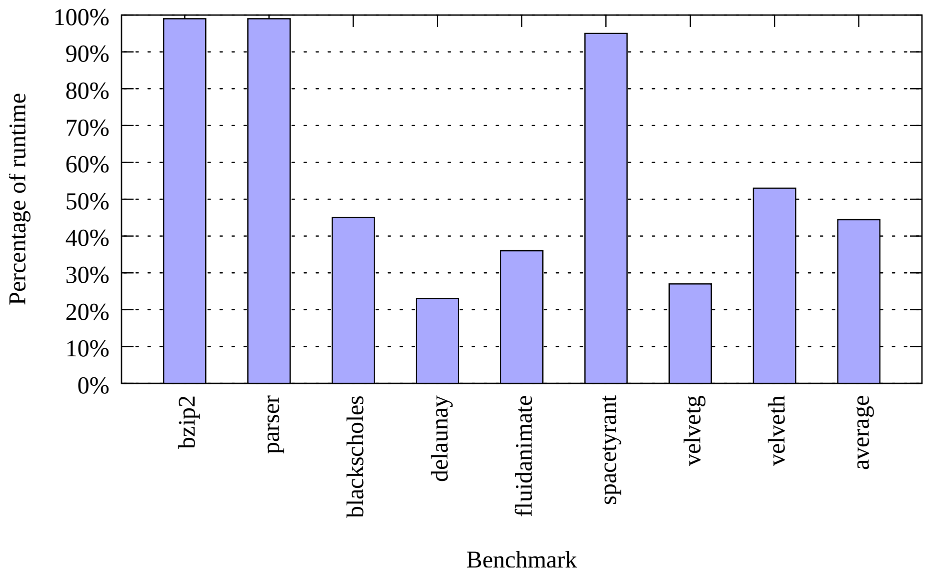


Figure 1.3: The percentage of total execution time taken by the hybrid loops in eight applications.

Second, cross-iteration dependences may exist in the computation part of a hybrid loop. It is impossible to perform DOALL parallelization of such loops even without the dependences introduced by I/O operations. Speculative parallelization has proven to be an effective approach to handling cross-iteration dependences in the computation part when these dependences manifest themselves infrequently. Previous works on speculative parallelization [126, 43] assume that the loops do not contain I/O operations. To apply speculative parallelization to the hybrid loops, the speculative execution of the I/O operations must be enabled.

Finally, after a hybrid loop is parallelized, multiple parallel threads may access the I/O bus at the same time. The contention on the I/O bus is increased with the increased parallelism in the hybrid loop. The increased I/O contention may lead to performance degradation. Therefore, techniques for reducing I/O bus contention must be developed to effectively parallelize hybrid loops.

This dissertation presents the SpiceC support for hybrid loops, which meets the above challenges.

1.1.4 SpiceC on Heterogeneous Multicores with GPUs

Graphics processing units (GPUs) provide an inexpensive, highly parallel system to perform computations that are traditionally handled by CPUs. SpiceC can be easily extended to support GPUs since its computational model can naturally handle the host/device memory hierarchy in heterogeneous systems with GPUs.

A few high-level programming models [76, 10, 14] have been proposed to explore loop-level parallelism using GPUs. To run a loop using the GPU, the programmer or compiler must ensure that there is no cross-iteration dependence. This prevents GPUs from executing loops that may encounter cross-iteration dependences at runtime.

The cross-iteration dependences in many loops rarely manifest themselves at runtime. Although these loops cannot be parallelized statically by any existing programming model, they are good candidates to be executed on GPUs when no cross-iteration dependence happens at runtime. Figure 1.4 shows the loop speedups of three applications achieved on a GPU by assuming the absence of cross-iteration dependence at runtime. These speedups are quite substantial, exceeding a factor of 10x in all cases. Thus, finding a method for running such loops on GPUs is higher desirable.

Speculative parallelization techniques based upon thread level speculation [126, 43], although applicable to CPUs, cannot be directly used on GPUs. This is due to the

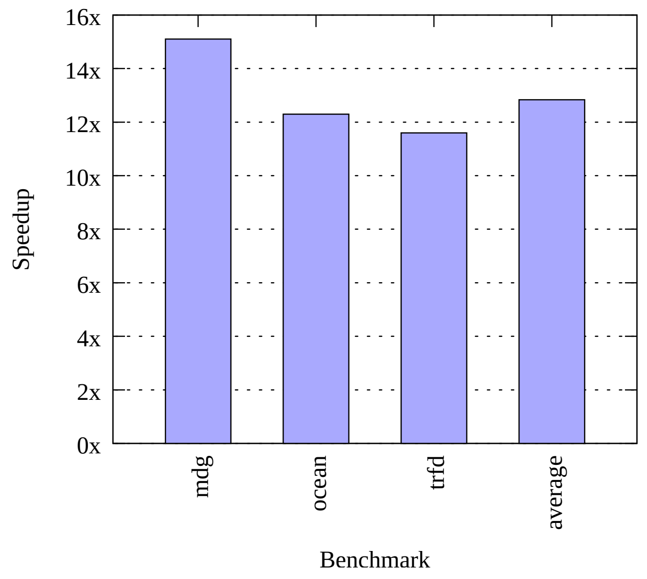


Figure 1.4: Potential loop speedups of three applications achieved on a GPU.

architectural differences between CPUs and GPUs. First, they require the results of each speculatively executed iteration to be stored in a logically separate space. GPUs do not have enough memory space to create a separate space for every thread due to thousands of threads. Second, these techniques commit the results of each iteration in a sequential order to preserve the original sequential semantics of the loop. Once misspeculation is detected in an iteration, these techniques require the iteration to be re-executed before committing the results of subsequent iterations. This requires complicated synchronization between threads, which cannot be efficiently implemented on GPUs due to very high GPU synchronization overhead [32].

This dissertation presents an extension to the programming model and runtime support of SpiceC, that enables speculative parallelism on GPUs.

1.1.5 SpiceC on Distributed Memory Clusters

Modern applications from important domains (e.g., data mining, machine learning, and computational geometry) have high processing demands and a *very large memory footprint*. To meet the demands of such applications, it is natural to consider the use of clusters of multicore machines. Clusters can not only scale to a large number of processing cores, they also provide a large amount of memory. They are especially attractive for modern applications because, with the advances in network technology, a cluster can provide an application with a faster storage system than a disk based storage system.

SpiceC can be extended for clusters since its computation model does not necessarily require shared memory. This dissertation addresses the following issues to port SpiceC to a cluster of multicore machines.

Data distribution. To support distributed memory clusters, the SpiceC runtime should be able to spread shared data across different machines. When a parallel thread accesses a shared data item, the runtime must locate the shared data item in the cluster. In case that dynamic linked data structures are used, the runtime should support reference to shared data via pointers.

Tolerating network latency. Network communication on clusters can be time-consuming. Figure 1.5 shows the percentage of the execution time taken by network communication for four applications running on a cluster of five multicore machines. The applications were executed sequentially using the memory from the five machines. In all four applications, network communication takes a significant portion of the execution time. Therefore, to efficiently run programs on clusters, the the network latency must be hidden by being kept

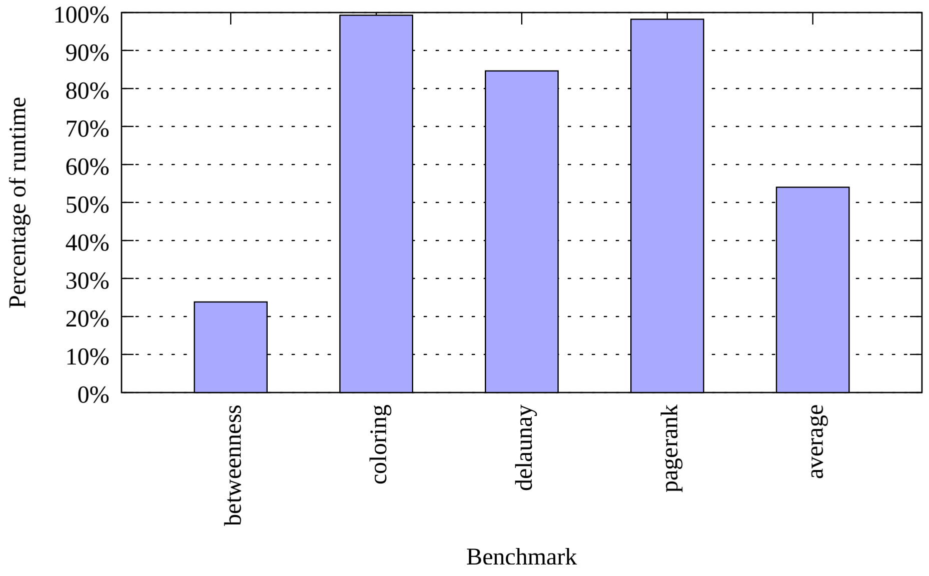


Figure 1.5: Percentage of the execution time taken by network communication.

off the critical path of the program execution.

1.2 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 describes the core features of the SpiceC system and its implementation. Chapter 3 discusses the compiler support for efficiently allowing speculative execution in SpiceC. Chapters 4 and 5 present the SpiceC support for parallelization in the presence of dynamic linked data structures and I/O operations. Chapters 6 and 7 describe how SpiceC is extended to take advantage of parallel systems other than multicore processors, i.e., systems with GPUs and distributed memory cluster of multicores. Related work is given in Chapter 8. Chapter 9 summarizes the contributions of this dissertation and identifies directions for future work.

Chapter 2

SpiceC on Multicore Systems

Shared memory parallel systems such as multicore processors have been widely used because they offer a single view of the memory thereby making parallel programming easier. Developers do not have to be aware of the location of data. A few shared memory parallel programming models, e.g., OpenMP [36] and Threading Building Blocks (TBB) [108], have been proposed to facilitate the development of parallel applications on shared memory multiprocessors. However, the above models have limitations in the forms of parallelism and the platforms they support.

To overcome the limitations of above models this dissertation introduces SpiceC for shared-memory multicore systems that is aimed at meeting the following demands:

I. Different forms of parallelism. A shared memory programming model must support various forms of parallelisms for successfully handling a wide range of applications. DOALL parallelism can be easily expressed using existing shared memory parallel programming models. However, more forms of parallelism (such as DOACROSS and pipelining) are needed and even support for speculative parallelism is required to enable parallelization in the presence of cross-iteration dependences.

II. Different forms of multicore platforms. A shared memory programming model should support not only multicore processors with cache coherence but also manycore processors without cache coherence. Existing shared memory programming model only works for processors with cache coherence. However, as multiprocessors transition from multicores to manycores, the scalability of traditional cache coherence protocols becomes a problem. Intel Labs has created their new Tera-scale multiprocessor – "Single-chip Cloud Computer (SCC)" which is a 48-core processor without support for cache coherence. Other similar manycore processors may be created in the future.

This chapter presents a shared memory programming model, called SpiceC (*S*calable *p*arallelism via *i*mplicit *c*opying and *e*xplicit *C*ommit), which is the core of the SpiceC system. It is designed to satisfy the aforementioned demands. The chapter also describes the implementation of the SpiceC programming model, which consists of a source-to-source compiler and a runtime library.

2.1 The Design of SpiceC

2.1.1 The SpiceC Computation Model

The *SpiceC* computation model represents a simple view of a parallel computation that serves as the basis for the programmer interface as well as corresponding implementation. In this model a parallel computation consists of multiple threads where each thread maintains a *private space* for data. In addition, a *shared space* is also maintained that holds shared data and provides a means for threads to communicate data to each other. A thread cannot access the private space belonging to other threads.

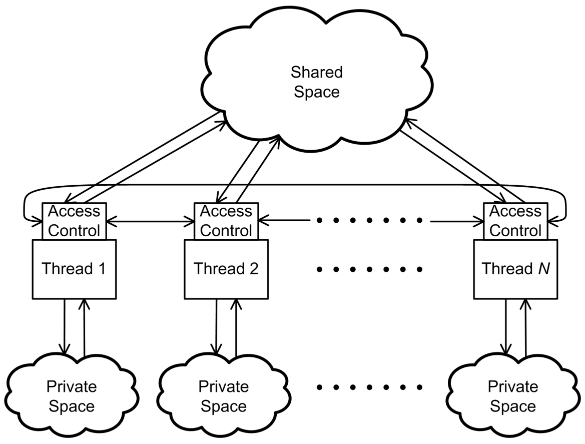


Figure 2.1: SpiceC Computation Model.

Each thread carries out its computation as follows: using *copying* operations it transfers data to its private space from shared space; it *operates* on the data in private space and produce results that are also written to the private space; and finally the thread affects the transfer of results to shared space using a `commit` operation thus allowing all threads to see the computed results. Thus, in this model, a thread performs its computation in *isolation* from other threads. Associated with each thread is an *access control* module which manages all accesses a thread makes to shared space. The access control modules of different threads perform actions on the shared space in coordination with each other.

A variable in shared space may be accessed by a thread by simply referring to it. A thread's first access to a shared space variable results in an image of that variable being created in its private space and this image is initialized by *copying* the value of the variable from shared space to private space. After copying, all subsequent accesses to the variable, including both read and write accesses, refer to the variable's image in the private space. In other words, the thread can manipulate the value of the variable in isolation, free of contention with other threads. Most importantly, the above semantics is enforced automatically by the implementation of SpiceC model – the programmer simply refers to the variable by its name without having to distinguish between first or subsequent accesses. Therefore the above copying is referred to as *implicit copying* as it is automatically performed without help from the programmer. There will be times when the private space of a thread will need to be cleared and so updated values of shared data can be accessed from shared space. The clearing of private space, like its setup through copying, is also *implicit*. The private space of a thread is cleared at program points that correspond to boundaries of *code regions* (e.g., iteration boundaries of parallel loops, boundaries separating pipeline stages, synchronization points).

When a thread has finished a block of computation and wishes to expose its results to other threads, it performs an *explicit commit* operation. This results in the transfer of variable's values from private space to shared space. The programmer is provided with an Application Programming Interface (API) using which he/she can specify points in the program where the values of variable's need to be committed to shared space. SpiceC supports two types of `commit` operations. The first type of `commit` operation simply copies all copied variables to the shared space after acquiring the write permission to the shared space. The second type of `commit` operation checks for *atomicity violation* before committing, that is, it checks to see if any of the copied variables was updated by another thread after being copied into the private space of the committing thread. An *atomic commit* enables programmers to express speculative computations with little effort.

2.1.2 SpiceC Design Choices

This section briefly describes how the design of SpiceC addresses the following issues: programmability (i.e., reducing the burden of parallel programming), versatility (i.e., providing means to program different types of parallelism), and performance portability (i.e., programs deliver performance on different platforms without having to rewrite them).

I. Programmability. The complexity of parallel programming hinders developers from using it. To make programming easier, SpiceC uses programming style similar to OpenMP [36]. Developers do not write new parallel programs but simply add *SpiceC directives* into the text of a sequential program. Parallelism is exploited by the compiler based upon the directives. SpiceC model can be used to cause selected portions of the code to be executed in parallel while the remaining code remains serial.

To ensure that the programmer can easily reason about a program, a form of *sequential consistency* is employed as follows. In SpiceC, the *copy* and *commit* operations naturally divide code in parallel regions into chunks. These chunks can be executed with atomicity and in isolation as specified by developers. Sequential consistency is then enforced by the software at the coarse grain level of chunks. For performance, the compiler can freely reorder instructions within a chunk without the risk of violating coarse grain sequential consistency.

SpiceC removes the need to explicitly specify communication between threads. The work of *copying* and *commit* is done by the compiler and the implementation of supporting API functions. Developers only need to insert *commit* calls in their codes. This greatly reduces developer burden of programming explicit communication in a message-passing model such as Message Passing Interface (MPI). For example, in a message-passing model, developers need to write the following three functions to transfer a linked list from a thread to another: the first converting the linked list to a binary string, the second transferring the string to another thread, and the last for reconstructing the linked list from the string. In SpiceC, after a thread commits the linked list, other threads can access it as any other shared space variable. Developers do not need to write any special code to handle the linked list.

II. Versatility. To efficiently parallelize applications, SpiceC supports multiple forms of parallelism: DOALL, DOACROSS, pipelining, and speculative parallelism. It is difficult to use existing parallel programming model to implement some forms of parallelism while ensuring correctness and delivering high performance. For example, speculation must be expressed using low level primitives in existing languages [101]. Speculative computation, misspeculation check, and misspeculation recovery must be programmed by the user which can be a non-trivial programming exercise. In contrast, SpiceC supports high level primitives and developer only needs to specify the parallelism and the coarse-grained dependences in the program. Compilers and runtime systems take care of the tedious part of parallel code generation.

SpiceC provides other benefits to parallel program developers. First, since a thread only updates variables in its private space before committing, it is easy to reverse the computation by simply discarding the content in the private space. In addition, the *commit* operation provided by SpiceC can check atomicity violation at runtime before committing. This makes it easy for developers to write programs with speculative parallelism. Secondly, parallel programming is error-prone. In SpiceC, an error in one thread does not affect other threads until it is committed. Therefore, it is easy to realize *fault tolerance* in SpiceC model. For example, developers can capture all fatal exceptions in thread T and reset T when exceptions happen. This enables other threads to continue operating even in the presence of errors in T.

III. Performance Portability. SpiceC can be implemented on any multicore or manycore system that has a logically shared memory with or without cache coherence. The portability of SpiceC makes it attractive alternative for developers who want their programs deployed on multiple platforms. Moreover, since current cache coherence protocols may not scale to manycore systems [11, 22], SpiceC makes it possible to scale shared-memory programs (e.g., OpenMP programs) to manycore systems.

2.1.3 Programming Interface

SpiceC aims to provide a portable standard for writing parallel programs while hiding the complexity of parallel programming from developers. This section presents directives supported by SpiceC.

SpiceC compiler directives enable the programmer to mark and parallelize a section

Directive	**Description**
#pragma SpiceC parallel [parallelism]	specify a parallel region
#pragma SpiceC region [name] [order]	specify a subregion
#pragma SpiceC commit [check_type] [order]	perform commit operation
#pragma SpiceC barrier [for_variable (variables)]	clear variables from private space

Table 2.1: Compiler directives implemented by SpiceC

of code that is meant to be run in parallel just as is the case in OpenMP [36]. A compiler directive in C/C++ is called a pragma (pragmatic information). Compiler directives specific to SpiceC model in C/C++ start with `#pragma SpiceC`. Table 2.1 shows the compiler directives supported by SpiceC.

`#pragma SpiceC parallel` is used to specify the parallel region of the program. The parallel region must be a loop, written in `for`, `while`, or `do ... while` form. Developers must specify in what form of parallelism the parallel region is to be executed. SpiceC supports three types of parallelism: DOALL, DOACROSS, and pipelining.

`#pragma SpiceC region` is used to specify a subregion in the parallel region. Developers can assign a name to the subregion and specify the execution order of different instances of the subregion or different subregions. By default, the whole loop body of the parallel region is one subregion.

`#pragma SpiceC commit` is used to call the `commit` operation, which commits all variables that are updated in the subregion to the shared space. Developers can specify whether to perform *atomicity* check before the commit operation. If the atomicity check fails, the subregion is re-executed. Developers can also specify the execution order of the `commit` operation which can be used to enforce the sequential semantics the parallel region. By default, a commit operation is placed at each exit of a subregion if no explicit commit is specified.

`#pragma SpiceC barrier` is used to insert a barrier in the parallel region. In the SpiceC model, each thread clears its private space when it encounters a barrier. Developers can optionally specify which variables to clear to enable avoidance of unnecessary copying overhead.

The following sections illustrate the use of directives in marking code regions and expressing different forms of parallelism. A few examples written in SpiceC are presented to show how different forms of parallelism can be expressed with little effort by the programmer.

2.1.4 Data Transfers in SpiceC

Figure 2.2 shows an example of explicit parallel code written in SpiceC that illustrates how shared space may be used by threads to communicate values to each other. The parallel code is executed by two threads. It is divided into two phases. In the first phase the two threads calculate values of variables a and b respectively. The programmer does not specify that a and b are to be copied into the private spaces – they are automatically copied upon their first uses. At the end of the first phase, a and b need to be committed into the shared space so that their new values are accessible to other threads. The programmer need not identify a and b in the commit operation – this is handled automatically which makes it easy to use. In other words, developer only specifies "when" to commit but need not specify "what" to commit. At the barrier, the two threads clear variable a and b in

```
#pragma SpiceC parallel {
  int tid = spicec_get_thread_id();
  if (tid == 0)
    a = calculate_a();
  else
    b = calculate_b();
  #pragma SpiceC commit
  #pragma SpiceC barrier for_variable(a,b)
  if (tid == 0)
    c = calculate_c(a,b);
  else
    d = calculate_d(a,b);
}
```

Figure 2.2: Communicating parallel threads written in SpiceC.

their private spaces and wait for each other. When both threads reach the synchronization point, they continue their computation and copy the value of a and b from the shared space again when they are used. Developers may or may not specify which variables to clear at the barrier. If variables are not specified at the barrier, the entire private space is cleared.

2.1.5 Non-speculative Parallelism

```
#pragma SpiceC parallel doall {
  for(i=0;i<nodes;i++)
    nodes[i].cluster = closest_cluster(nodes[i], clusters);
  #pragma SpiceC commit
} //implicit barrier


#pragma SpiceC parallel doall {
  for(i=0;i<clusters;i++)
    clusters[i].mean = calculate_mean(clusters[i], nodes);
  #pragma SpiceC commit
}
```

Figure 2.3: K-means clustering algorithm written in SpiceC.

I. DOALL Parallelism in SpiceC. Figure 2.3 shows a K-means clustering program written in SpiceC. This program looks very similar to an OpenMP program. Without the directives, the program can still be compiled as a sequential program. The K-means clustering algorithm has two steps: the first step assigns each node to the cluster with the closest mean and the second step calculates the new means to be the centroid of the nodes for each cluster. Each step is parallelized using DOALL parallelism. `Commit` operations are inserted at the end of each parallel region. By using DOALL, workloads are divided and assigned among the threads at the beginning of the parallel region. No execution order is specified in this example. Therefore each thread executes fully in parallel with other threads. All threads join at the end of the parallel region, which acts as a implicit barrier.

II. DOACROSS Parallelism in SpiceC. Figure 2.4 shows a typical stream decoding program written in SpiceC model. The program repeatedly does the three steps: read a stream from the input buffer, decode the stream, and write the stream back to the output buffer. There are two cross-iteration dependences in the loop. In each iteration, the `read_stream` function reads a stream from the input buffer and moves the input buffer pointer to the next stream. Therefore, the `read_stream` in the current iteration depends on itself in the previous iteration. Similarly, `write_stream` also has cross-iteration dependence on itself. To parallelize this program, the loop is annotated with DOACROSS parallelism. By using DOACROSS, consecutive iterations are assigned to different threads to allow them to be executed in parallel. The loop is divided into three subregions with each step forming one subregion. Subregion R1/R3 in the current iteration is specified to only be executed after R1/R3 in the previous iteration finishes. The keyword "ITER" used in the SpiceC pragmas refers to the current iteration and "ITER-1" refers to the preceding iteration. No execution

```
#pragma SpiceC parallel doacross {
  for(i=0;i<n;i++) {
    #pragma SpiceC region R1 order_after(ITER-1, R1)
    {
      job = read_stream(input_buffer);
      #pragma SpiceC commit
    }
    #pragma SpiceC region R2 no_order
    {
      decode(job);
      #pragma SpiceC commit
    }
    #pragma SpiceC region R3 order_after(ITER-1, R3)
    {
      write_stream(output_buffer, job);
      #pragma SpiceC commit
    }
  }
}
```

Figure 2.4: A typical stream decoder written in SpiceC.

order is specified for R2 since it does not have cross-iteration dependence.

III. Pipelined Parallelism in SpiceC. Figure 2.5 shows the same stream decoder as shown in Figure 2.4. But in this example, it is parallelized using pipelined parallelism. The loop body is still divided into three subregions. Each subregion is a stage of the pipeline. By using pipelining, each subregion is executed in one thread, similar to Decoupled Software Pipelining [106]. In this program, the subregion R2 needs data from R1 and R3 needs data from R2. Therefore, R2 is specified to be executed after R1 and R3 after R2. In this case, it is not necessary to specify the cross-iteration execution order (e.g., R1 in the current iteration should be executed after R1 in the previous iteration) since it is ensured by the very nature of pipelining. Unlike the model of producer and consumer, SpiceC model commits data to the same place every time. Therefore, subsequent `commit` may overwrite the value from previous `commit`. To avoid this problem, the scalar variable `job` is expanded into an array and make each iteration compute using different memory locations.

2.1.6 Speculative Parallelism

SpiceC also supports speculative parallelization for exploiting parallelism between code regions that occasionally involve dependences although mostly they can be safely executed in parallel. With the SpiceC programming model, speculation can be easily applied to all forms of parallelism described in the preceding section. Users just need to add an `atomicity_check` clause if the regions are being speculatively executed in parallel.

Figure 2.6 shows an example of using speculation in context of a DOACROSS loop. The code in the example is the kernel of benchmark telecomm-CRC32 in MiBench

```
#pragma SpiceC parallel pipelining {
  for(i=0;i<n;i++) {
    #pragma SpiceC region R1 {
      job[i] = read_stream(input_buffer);
      #pragma SpiceC commit
    }
    #pragma SpiceC region R2 order_after(ITER, R1) {
      decode(job[i]);
      #pragma SpiceC commit
    }
    #pragma SpiceC region R3 order_after(ITER, R2) {
      write_stream(output_buffer, job[i]);
      #pragma SpiceC commit
    }
  }
}
```

Figure 2.5: Stream decoder parallelized using pipelining parallelism.

```
error=0;
i=0;
#pragma SpiceC parallel doacross {
  while(i<n) {
    #pragma SpiceC region R1 order_after(ITER-1, R1) {
      string = read_file(file[i++]);
      #pragma SpiceC commit
    }
    #pragma SpiceC region R2 no_order {
      ret = calculate_crc(string);
      if (ret != 0)
        error |= ret;
      #pragma SpiceC commit \
                    atomicity_check order_after(ITER-1, R2)
    }
  }
}
```

Figure 2.6: Kernel of benchmark telecomm-CRC32.

suite. The program calculates 32-bit CRC for each file in the file list and put the results into the `error` variable. The update of `error` depends on the value of `error` from previous iterations. Therefore, the subregion R2 has a cross-iteration dependence on itself. However, since the function `calculate_crc` usually returns 0 that represents success in real runs, the cross-iteration dependence is rarely manifested. The program can be parallelized with *speculating* on the absence of the dependence. The program is parallelized using DOACROSS parallelism. The loop body is divided into two subregions. The subregion R1 is executed sequentially since there is a cross-iteration dependence due to variable i. The subregion R2 is executed in parallel but the commit operation in R2 is performed sequentially to ensure the sequential semantics of the loop. Before the commit operation, an atomicity check is performed to detect if the cross-iteration dependence occurred. If the dependence is detected, then the whole subregion R2 is re-executed. Otherwise, R2 can commit its results safely.

2.2 The Implementation of SpiceC

This section describes the implementation of SpiceC on shared-memory systems with or without cache coherence. The core components of the SpiceC prototype implementation consist of: a source-to-source compiler and a user-level runtime library, as shown in Figure 2.7. The compiler translates high-level SpiceC code into explicitly parallel C/C++ code while the runtime provides support for access control, synchronization, and loop scheduling. The remainder of this section first presents the implementation of the SpiceC runtime library amd then it describes the SpiceC compiler.

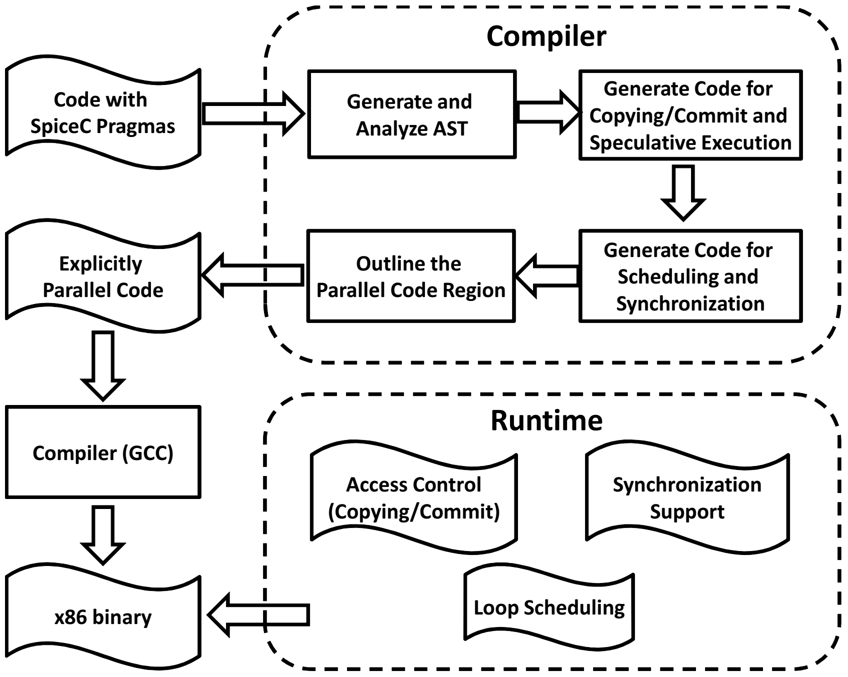

Compiler
Code with SpiceC Pragmas
Generate and Analyze AST
Generate Code for Copying/Commit and Speculative Execution
Generate Code for Scheduling and Synchronization
Outline the Parallel Code Region
Explicitly Parallel Code
Compiler (GCC)
x86 binary
Runtime
Access Control (Copying/Commit)
Synchronization Support
Loop Scheduling

Figure 2.7: Overview of SpiceC implementation.

2.2.1 The Runtime

I. Access control. In the implementation of SpiceC, all accesses to the shared space are managed by the *access control* runtime system. The access control runtime system makes sure that all variables are copied and committed properly in the parallel region and the copy and commit operations strictly follow the user-specified execution order. The access control is implemented based on an STM library — Transactional Lock II (TL2) [41]. The *copy* operations are implemented using STM read/write barriers. To make sure that every variable is copied and committed properly, the TL2 library is modified to maintains metadata for each variable. Each variable has *metadata* in both shared and private spaces. In the shared space, the version of each variable is recorded for atomicity check. In a private space, metadata includes its shared space address, its type/size, whether they have been copied into the private space, and its version when it is copied.

II. Relaxed copy/commit model. The programs written in SpiceC model are portable across CMP machines with or without cache coherence. However, in some cases, `copy` and `commit` operations are unnecessary and may incur higher overhead compared to directly accessing the shared space. To improve the performance of SpiceC in these cases, SpiceC relaxes the copy/commit model by allowing the threads to directly access the shared space.

On systems with cache coherence, since the shared space is cachable and fast to access, SpiceC allows directly accessing the shared space if copy and commit operations are unnecessary. The copy/commit model is relaxed in the following cases. (1) If a subregion uses non-check commit without any order specification, SpiceC allows the subregion to directly operate on the shared space since the subregion has no dependence with any other part of program that is executed simultaneously. (2) Read-only variables do not need to be committed even in the subregions where copy and commit operation are required. Therefore, a variable is only copied when it is to be written. When a variable is to be read, SpiceC only records its version number for atomicity check. In this way, the copy/commit overhead for read-only variables is reduced.

On systems without cache coherence, shared space is usually non-cachable to avoid consistency problem. Directly accessing the shared space is much slower than accessing the private spaces which is cachable. Therefore, SpiceC makes copy of every variable that is to be accessed in the private space to make subsequent accesses to the same variable faster.

The copy/commit model on systems without cache coherence is not relaxed since it is better for performance.

III. Synchronization. In the SpiceC programming model, developers can specify the execution order for subregions and commits. Depending on the system configuration, the SpiceC runtime uses different synchronization mechanisms to ensure the specified execution order.

On systems with cache coherence, the busy-waiting algorithm is used to ensure the execution order. A flag is set after each subregion is finished. Before a subregion or commit is executed, the SpiceC runtime checks the flags of the subregions which it is waiting for. The execution continues if the flags are set. The busy-waiting algorithm achieves low wake-up latency and hence yield good performance [88].

Systems without cache coherence usually provide hardware support for message-passing interface (MPI) to enable fast communication between cores. Therefore SpiceC uses MPI to ensure the execution order. For every iteration or subregion, the SpiceC runtime knows the thread that will execute it. Since the execution order is explicitly specified in the code, a thread can know which thread is waiting for it at runtime. Therefore, after a thread finishes a subregion, it can send signal to the waiting thread.

IV. Loop scheduling. Loop iterations are scheduled according to the specified parallelism. For loops with DOALL parallelism, the workloads are distributed among the threads at the beginning of the loop. Since subsequent iterations often display temporal locality, they are assigned to the same thread for better cache performance. For example, for a DOALL loop with n iterations, thread i will execute the loop from iteration $n/m * i$ to $n/m * (i + 1) - 1$, where m is the total number of threads. For loops with DOACROSS parallelism, each thread executes one iteration at a time. After committing an iteration, a thread fetches another iteration by executing the loop index. The loop index is always executed in sequential order due to the existence of cross-iteration dependences. It is not allowed to depend on any part of loop body. If the loop index does depend on loop body, developers need to move it into the loop body and specify the execution order or check type for it. For example, a DOACROSS loop is executed by m threads. Thread i only executes iteration $m * j + i$, where $j \in \{0, 1, 2, \ldots\}$. Thread i begins to execute iteration $m * j + i$ only after it finishes iteration $m * (j - 1) + i$ and thread $i - 1$ finishes the index part of iteration $m * j + i - 1$. For loops with pipelining parallelism, each thread is dedicated to execute one subregion, as in [106]. For example, a pipelining loop contains m subregions. Thread i executes the i-th subregion for every iteration.

2.2.2 The Compiler

The SpiceC compiler analyzes the code annotated with SpiceC directives and generates explicitly parallel code. The code translation consists of four steps, as shown in Figure 2.7. Three out of the four steps are described in this section. One of the key contributions of this dissertation is that the SpiceC compiler automatically generates code for speculative execution by inserting copy/commit operations into the parallel code. This part is elaborated separately in Chapter 2 due to its complicated nature.

I. Generating and analyzing the AST. Before code translation, the SpiceC code is analyzed to determine what translation to perform. The code analysis is performed on the abstract syntax tree (AST) of the SpiceC code, which is generated using the ROSE compiler infrastructure [104]. During the AST generation, the SpiceC directives are converted to the attributes of corresponding AST node (i.e., code constructs).

Two passes of analysis are performed on the AST. In the first top-down pass, parallelism attributes are passed down from the outer loops to inner regions. Translation of code regions requires parallelism information to generate correct synchronization code. In the second bottom-up pass, inner regions' attributes (e.g., ordering information and atomicity checks) are passed to the outer loops. The information is needed for generating initialization code. For example, if atomicity checks are specified inside inner regions, the initialization of STM library must be inserted before the loop.

II. Generating code for synchronization and scheduling. After the AST analysis, the SpiceC compiler generates code for synchronization and scheduling. *Synchronization code* controls signal sending and waiting between regions. It is inserted at the beginning and/or end of regions to enforce the execution order between region instances. Synchronization data structures are created to hold signals. They are defined according to the form of parallelism. *Scheduling code* controls how the loop iterations are scheduled. Calls to the scheduling runtime library (as described in Section 2.2.1 are inserted before all parallel loops. The parallelism type decides which scheduling policy is used.

III. Outlining parallel code region. At the end of the code translation, the parallel loops are wrapped into functions, which can then be called for scheduling. Outlining is implementation using ROSE [104]. Outlining needs to be performed at last since the parameters of the wrapper functions are defined according to the data dependences. All above code translation may change the data dependences and then invalidate outlining.

2.3 Evaluation

This section evaluates the performance of SpiceC on systems with and without cache coherence support. Two implementations of SpiceC were developed: (1) the first is designed for shared-memory systems with cache coherence using relaxed copy/commit model and busy-waiting synchronization; and (2) the second is designed for shared-memory systems without cache coherence using strict copy/commit model and message passing synchronization. The experiments are conducted on a 24-core DELL PowerEdge R905 machine. Table 2.2 lists the machine details. The machine runs Ubuntu Server v10.04. Both implementations run on the machine. Since the machine has hardware support for cache coherence, accessing the shared space on it is faster than that on a system without cache coherence. Therefore, the performance reported for the strict copy/commit model is higher than the actual performance on a system without cache coherence. The machine is used for evaluating the strict model because a manycore machine with message passing hardware instead of cache coherence is not available.

Processors	4×6-core 64-bit AMD Opteron 8431 Processor (2.4GHz)
L1 cache	Private, 128KB for each core
L2 cache	Private, 512KB for each core
L3 cache	Shared among 6 cores, 6144KB
Memory	32GB RAM

Table 2.2: Dell PowerEdge R905 machine details.

2.3.1 Benchmarks

Benchmark	LOC	Parallelism	Speculation?	% of Runtime	# of Dir.
streamcluster	300	DOALL	No	88%	21
fluidanimate	206	DOALL	No	96%	16
CRC32	15	DOACROSS	No	100%	7
		DOACROSS	Yes	100%	5
256.bzip2	127	DOACROSS	Yes	99%	17
		pipelining	No	99%	7–19
197.parser	1183	DOACROSS	Yes	99%	15
ferret	84	pipelining	No	99%	13–25
mpeg2enc	91	pipelining	Yes	56%	13-21

Table 2.3: Benchmark details. From left to right: benchmark name, lines of code in the parallelized loops, type of parallelism exploited, whether speculation is used, percentage of total execution time taken by the parallelized loops, and number of directives used for parallelization.

SpiceC was applied to benchmarks selected from PARSEC, SPEC CPU2000, MediaBench, and MiBench suites. Two different forms of parallelism were applied to benchmark `CRC32` and `bzip`. For some of the benchmarks, their loops were unrolled to increase the workload of each iteration. Table 2.3 shows the details of the benchmarks.

I. Benchmarks using DOALL. `Streamcluster` was developed to solve online clustering problem. It is from the PARSEC suite and thereby has a manually parallelized version. Its serial version was parallelized with the SpiceC directives. There are 9 loops that can be parallelized using DOALL. Among them, three loops need to compute sums of large arrays of values. For the three loops, simple modifications was done to realize parallel sum reduction.

`Fluidanimate` was designed to simulate an incompressible fluid for interactive animation purposes. It is also from the PARSEC suite. Its serial version requires manually partitioning workloads for parallelization. SpiceC was applied to its pthread version with all pthread calls replaced by the SpiceC directives. In the pthread version, there are a lot of lock operations for synchronization. In the SpiceC version, all these lock operations are removed, which makes the program easier to understand.

II. Benchmarks using DOACROSS. The `CRC32` program computes the 32-bit CRC used as the frame check sequence in Advanced Data Communication Control Procedures (ADCCP). Its kernel is shown in Figure 2.6. In each iteration, it first reads data from a file, then calculates CRC based on the data, and finally summarizes the error code into variable "error". In the experiments, `CRC32` was parallelized using two forms of parallelism. One way to parallelizing it is applying non-speculative DOACROSS. The loop body was divided into three subregions — input, computation and output. Input and output subregions need to be executed in order since there are cross-iteration dependences in them. Computation subregion can executed in parallel. The other way of parallelizing it is using speculation as shown in Figure 2.6.

`Bzip2` is a tool used for lossless, block-sorting data compression and decompression. In the experiments, its compression part was parallelized. There are many superfluous cross-iteration dependences on global variables in bzip2. To remove these dependences, many global variables were made local to each iteration and replicate buffers for each iteration. `Bzip2` was parallelized using both DOACROSS and pipelining. For the DOACROSS version, the loop body was divided into three subregions – input, compression, and output. Cross-iteration dependences in input and output subregions are enforced by specifying execution order for them. Cross-iteration dependences in compression subregions rarely manifest at runtime. Therefore, the compression subregion was executed speculatively. For the pipelining version, the loop body was carefully divided into stages. Several functions were inlined in the loop and some parts of loop were moved around to make the stages balanced.

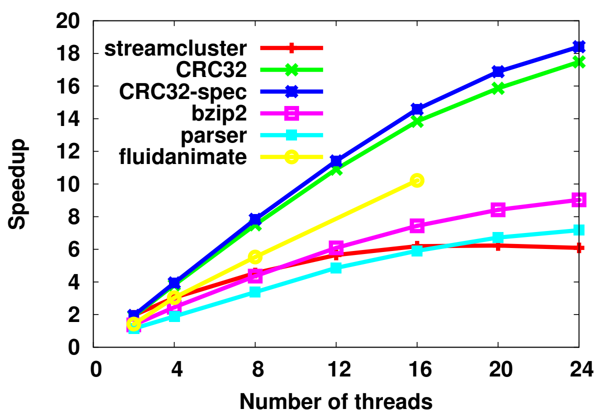
`Parser` is a syntactic English parser based on link grammar. It was parallelized using speculative DOACROSS. The problem with parser is that dependences may occurs for many variables, including control variables which is a set of global variables and dictionary structures which is a set of linked dynamic data structures. This causes the commit operations to become bottlenecks for the parallelized program.

III. Benchmarks for pipelining. `Ferret` is a tool used for content-based similarity search of feature-rich data. In the experiments, its serial version was parallelized using two different forms of pipeline: a 6-stage pipeline and a more fine-grained 14-stage pipeline. To realize the pipelining, several functions were inlined but no statement was moved.

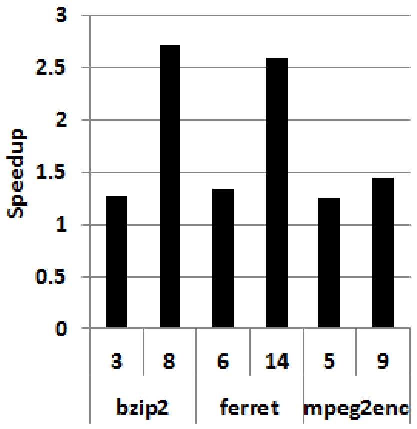
`Mpeg2enc` is a MPEG video encoder. A loop in function "dist1" was parallelized using a 5-stage pipeline and a 9-stage pipeline. Loop exits are required to be speculated to achieve parallelization.

2.3.2 Performance of the Relaxed Model

Figure 2.8(a) shows the speedups of benchmarks with DOALL and DOACROSS parallelisms using the relaxed model. As we can see, the performance of most benchmarks scales well over the number of threads, except for `streamcluster`. The performance of `streamcluster` goes down when the number of threads exceeds 20. This is because only the inner loops of `streamcluster` were parallelized. As the number of threads increases,



(a) DOALL & DOACROSS



(b) Pipelining

Figure 2.8: Speedup with the relaxed model.

the overhead of creating and joining threads gets higher while the workload for each thread shrinks. Only results of 2, 4, 8, and 16 threads are shown for `fluidanimate` because it can only take power of 2 as thread number. `CRC32` achieves the highest speedup among all the benchmarks. This is because most part of the program can be executed in parallel. Its speculative version performs better than the non-speculative version since the misspeculation rate is only around 0.1%. `Parser` and `bzip2` achieves moderate speedup. Their misspeculation rates are 0.92% and 0.5% respectively. Most performance loss is due to atomicity checks.

Figure 2.8(b) shows the speedups of benchmarks with pipelining parallelism. The number under each bar indicates the number of threads used. The figure only shows the results of certain numbers of threads since it is hard to breakdown the program into arbitrary number of stages. Their speedups are between 1.2x and 2.8x, much less than the speedups achieved by DOALL and DOACROSS. For `bzip2` and `ferret`, their poor performance is mainly due to the unbalanced workloads. For `mpeg2enc`, a significant performance loss is due to its high misspeculation rate which is around 16%.

Figure 2.9 shows the average parallelization overhead on each thread. The number under each bar indicates how many threads are used when calculating the overhead. The overhead includes time spent on synchronization, copy/commit, atomicity check, and misspeculation recovery. As we can see, parallelization overhead increases with the number of threads. `streamcluster` and `fluidanimate` has higher overhead since they were parallelized on more fine-grained level. The overhead of `mpeg2enc` is mainly due to high misspeculation rate. `Parser` has moderate overhead, which is caused by the atomicity check for dynamic data structures. A significant overhead of `bzip2` using pipelining is synchronization due to unbalanced workloads. The rest benchmarks have relatively low parallelization overhead.

2.3.3 Performance of the Strict Model

Figure 2.10(a) shows the speedups for benchmarks with DOALL and DOACROSS parallelism using the strict model. Similar to results in the previous section, the performance of most benchmarks scales well with the number of threads, except for `streamcluster`. We can see that `CRC32` and `bzip2` are not affected much by using strict copy/commit model. `CRC32` still achieves highest speedup since it has very few variables that require copying. `Bzip2` needs to copy only a few more variables using the strict model. Therefore, its performance does not degrade much. The performance of `Streamcluster` and `fluidanimate` are greatly degraded by use of the strict model. The major part of performance loss is due

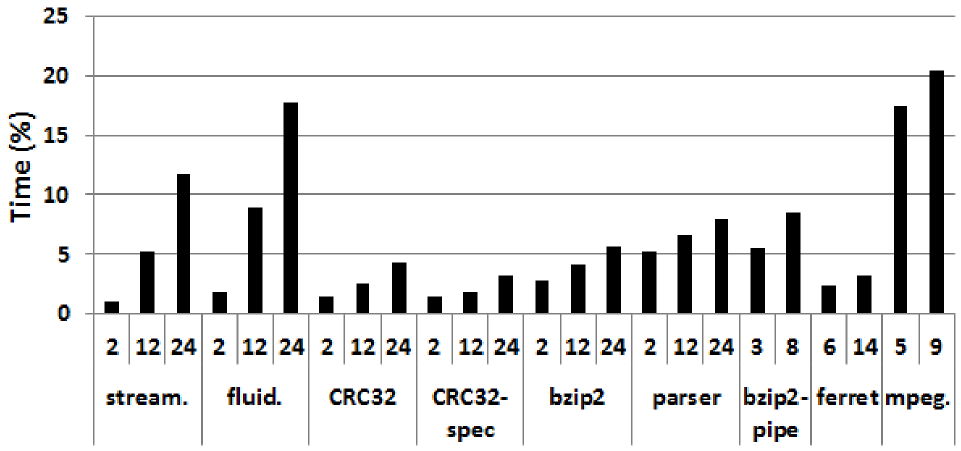
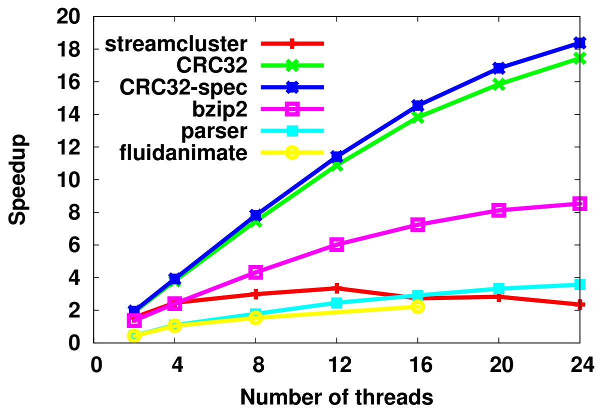
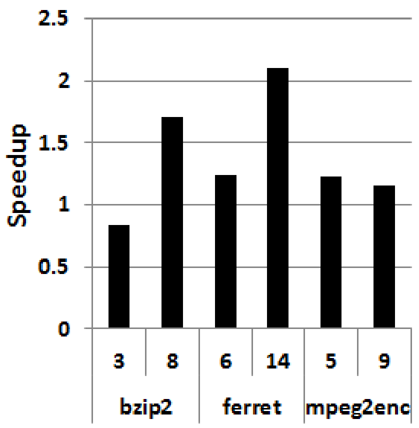


Figure 2.9: Parallelization overhead with the relaxed model.



(a) DOALL & DOACROSS



(b) Pipelining

Figure 2.10: Speedup with the strict model.

to the large amount of data that needs to be copied. `Streamcluster` and `fluidanimate` require 98MB and 163MB respectively memory space at runtime. Using the strict model, they need to dynamically copy/commit most of the data in each parallel region. Since the parallelization is done at fine-grained level, the copy and commit overhead is significant compared to the computation workload assigned to each thread. Therefore, `streamcluster` and fluidanimate show loss of performance using the strict model. `Parser` also takes a moderate performance hit using the strict model. This is because many dynamic data structures need to be copied even though they are read-only. Figure 2.10(b) shows the speedups of the benchmarks with pipelining parallelism. `Bzip2` loses performance because some buffers need to be copied between subregions. `Mpeg2enc` is parallelized at very fine-grained level. The extra copy and commit overhead significantly drags down its performance. `Ferret` is only slightly affected by using the strict model. Its workload is well partitioned so that there is not much data requiring copy and commit.

Figure 2.9 shows the average parallelization overhead on each thread when using the strict model. The number under each bar indicates how many threads are used when calculating the overhead. As we can see, parallelization overhead increases in the strict model. `Streamcluster` and `fluidanimate` incur up to 30% overhead due to extra copy and commit operations. `Parser`'s overhead using 24 threads goes from 8% up to 18% which significantly degrades its performance. `Mpeg2enc`'s overhead with the 9-stage pipeline increases by 5%, which is significant compared to each stage's workload.

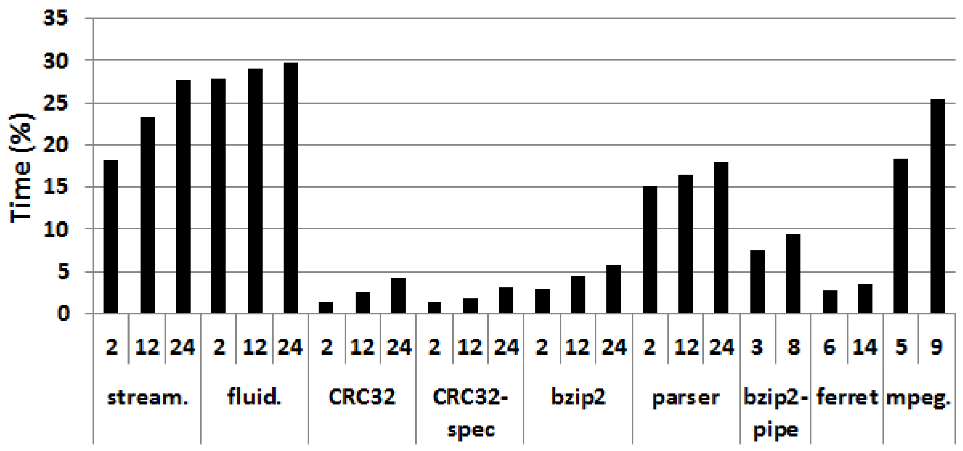


Figure 2.11: Time overhead with the strict model.

2.4 Summary

This chapter has presented the core SpiceC system. It has a shared memory parallel programming model that is designed to exploit various forms of parallelism and can be ported across shared memory multiprocessors that may or may not support cache coherence. The SpiceC programming model provides a set of compiler directives, which can be easily used to parallelize sequential programs. The implementation of SpiceC consists of a source-to-source compiler and a runtime library. The compiler translates the high-level SpiceC code into explicitly parallel code while the runtime provides support for access control, synchronization, and loop scheduling. The SpiceC system executes programs using different runtime schemes to achieve optimal performance on systems with and without cache coherence support.

Chapter 3

Support for Speculative Execution

Many real-world applications that cannot be statically parallelized due to the presence of dependences have been observed to contain high levels of parallelism. This is due to the fact that the dependences present arise very infrequently at runtime. To exploit this form of dynamic parallelism, speculative parallelization [107, 126, 43] has been proposed. Speculative parallelization executes the code in parallel, optimistically assuming that the dependences which inhibit parallel execution will not arise. However, the runtime looks out for the manifestation of such dependences, and if they do arise, the code is reexecuted preserving the original sequential semantics of the program.

While exploiting dynamic parallelism is highly desirable, programming speculatively parallelized code in an unmanaged language, such as C/C++, is a demanding task for programmers. In this chapter it is shown how the SpiceC compiler addresses this problem by automatically generating speculatively executed code. Table 3.1 shows the programming burden of writing speculatively executed code using previous software transactional memory (STM) [118] implementations. All of these implementations require significant amount of programming effort, including inserting read/write barriers for each shared read/write, annotating functions called directly and indirectly in a transaction (i.e., a speculatively executed code region), and manually handling the precompiled functions and system calls. In the SpiceC system, all these tasks are automatically done by the compiler, which greatly alleviates the programming burden.

	STM libs	Intel STM	SpiceC
Instrumenting shared accesses	Manual	Auto	Auto
Annotating speculative functions	Manual	Manual	Auto
Dealing with precompiled library functions	Manual	Manual	Auto
Dealing with system calls	Manual	Manual	Auto

Table 3.1: A comparison of programming burden for writing speculatively executed code. *Manual* — tasks done manually by the programmer; *Auto* — tasks done automatically by the compiler.

With the SpiceC programming model, programmers only need to mark the code regions that need to be executed speculatively. The compiler generates code for speculative execution by automatically inserting STM constructs (which implement the copy/commit operations as mentioned in Section 2.2.1). The compiler also provides support for precompiled library functions (e.g., C standard library) and system calls (e.g., I/O operations and system calls) to enable their parallel execution within transactions.

This chapter presents the design of the SpiceC compiler that transforms annotated C/C++ code into speculatively-executable code based on the high-level SpiceC directives. A series of important optimizations are introduced for reducing the overhead of speculative execution, including: placement of read/write barriers only for accesses that actually can cause a data race; elimination of redundant read/write barriers by caching shared variables; and eliminating unnecessary search in the write buffer.

3.1 Code Generation for Speculative Execution

The key part of the SpiceC compiler is automatically inserting the STM constructs into the C/C++ code to achieve the speculative execution as well as the copy/commit scheme in the strict model. Since the copy/commit scheme is part of the speculative execution, this chapter mainly focus on the code generation for speculative execution.

Automatically inserting STM read/write barriers (i.e., the copy operations) into the C/C++ code is very challenging. First, for C/C++ programs, the compiler must generate code statically—it is impossible to use JIT compilation as managed languages do. JIT compilers can perform operations such as creating an atomic clone for a function on the fly, dynamically suppressing redundant/dead barriers, and transact-ifying library functions by inlining them, which static compilation cannot do. Second, due to the unsafe use of pointers in C/C++ programs, the compiler is forced to use word-based STMs while the compilers for managed languages can use object-based STMs. Finally, without type safety, shared reads/writes should be checked conservatively, which further makes reducing STM overhead quite challenging.

The SpiceC compiler translates programs annotated with the SpiceC directives into source code instrumented with calls to the STM library — TL2 [41]. Figure 3.1 shows the process of code generation, where the grey blocks indicate the work done by the compiler.

The user code is written in C/C++ annotated with the SpiceC directives. The compiler first takes the user code as input and translates the functions and calls based on the call graph. It then instruments code with low-level STM API constructs and uses static data race analysis to find shared data accesses, as described in detail in Section 3.1.2. Finally, the compiler eliminates redundant read and write checks to reduce the overhead, which will be described in Section 3.3. The generated code is C/C++ code with calls to low-level STM functions, and can be compiled with regular compilers, such as GCC, to generate an executable binary.

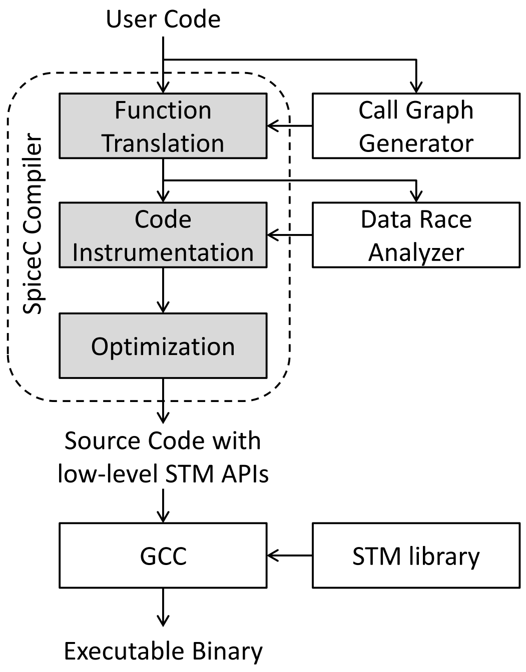


Figure 3.1: Overview of code generation.

3.1.1 Function Translation

This first step in code translation is to translate all functions defined in the user code according to their call sites and their usage via function pointers. To do this, the compiler first statically analyzes the user code to generate the static call graph. Based on the call graph, the compiler classifies the functions into the following four types.

1. *Atomic functions* are functions called only inside transactions (i.e., code subregions specified with atomicity checks). In other words, their call sites are either in transactions or in other atomic functions. These functions need to be executed atomically. Therefore, the compiler must instrument the shared reads/writes in them with low-level STM API constructs.

2. *Non-atomic functions* are functions that are never called in any transaction, i.e., these functions are called either outside transactions or from non-atomic functions. The compiler does not need to do any special code translation for these functions unless the strict copy/commit model is used.

3. *Double-duty functions* are functions called both inside and outside transactions. The compiler creates atomic clones for such functions and instruments the clones with low-level STM API constructs. All calls to these functions that appear in transactions are replaced with calls to their atomic clones.

4. *Dynamically-called functions* are functions called through function pointers. Since these functions may be called both inside and outside transactions, it can only be decided at runtime whether to call the original function or its atomic clone. To solve

this problem, for each dynamically-called function, the compiler creates an atomic clone and places a conditional call to the atomic clone at the beginning of the original function. Figure 3.2 gives an example: the original function checks a thread-local variable, `inside_transaction` at the beginning. The variable is set to be true when the current thread enters a transaction. If the variable is true, the original function then calls its atomic clone; otherwise, the original statements inside the function are executed.

```
int original_func()
{
   if ( inside_transaction == true )
      return atomic_func();
   // original statements here
}
```

Figure 3.2: Code translation for a dynamically-called function.

3.1.2 Code Instrumentation

After function translation, the compiler instruments the transaction code, atomic functions, and atomic clones of functions with low-level STM API constructs. Table 3.2 presents the low-level STM API calls used by the SpiceC compiler. This API is designed for word-based STM libraries, such as TL2 [41] and TinySTM [46]. SpiceC uses word-based STM libraries rather than object-based libraries due to the lack of type safety and the presence of unsafe pointer arithmetic in C/C++ [134]. Although the compiler is designed to generate code for STM libraries, it can also be used for hardware and hybrid TM libraries as long as their implementations are compatible with the API listed in Table 3.2.

API function	Description
txDesc* stmGetDesc()	Get a transaction descriptor.
void stmBegin(txDesc*)	Start a transaction
void stmEnd(txDesc*)	Validate/commit a transaction
void stmAbort(txDesc*)	Explicitly abort a transaction
Type stmRead⟨Type⟩(txDesc*, Type*)	STM read barrier
void stmWrite⟨Type⟩(txDesc*, Type*, Type)	STM write barrier
void stmReadBytes(txDesc*, void*, void*, size_t))	STM read barrier
void stmWriteBytes(txDesc*, void*, void*, size_t))	STM write barrier
void stmLogStack(txDesc*)	Log local variables on the stack
void stmLogBytes(txDesc*, void *, size_t)	Log a specific memory location
(void*) stmMalloc(txDesc*, void*)	STM malloc
void stmFree(txDesc*, void*)	STM free
Type stmDirectRead⟨Type⟩(txDesc*, Type*)	STM read barrier without searching the write buffer

Table 3.2: The low-level STM API used in translated code.

All low-level STM APIs require a transaction descriptor as an input. The transaction descriptor is obtained by calling the function `stmGetDesc`. Each thread has a unique transaction descriptor, which is created in thread-local storage when function `stmGetDesc` is called for the first time in the thread. Calls to function `stmGetDesc` are inserted before every transaction. To eliminate redundant accesses to thread-local storage within a transaction, a local variable is used to hold the transaction descriptor and passed to every called atomic function through arguments.

The compiler inserts the `stmBegin` and `stmEnd` API calls at the boundaries of transactions to start and commit them. The `stmAbort` call is inserted where explicit transaction abort is specified. The above three inserted functions dynamically decide if the transaction is nested in another transaction by checking a thread-local variable `inside_transaction`. If so, they start/end the transaction as an inner transaction; otherwise, the transaction is treated as an outermost transaction. In previous works [134, 93], transactions are statically classified as outermost transactions and inner transactions and different API constructs are inserted to start/commit them. Dynamic checks are used because a function containing transactions may be called from both inside and outside of other transactions. Therefore, the transactions in the function may be either outermost or inner transactions, depending on the function's call site.

Barrier functions perform the required STM operations to ensure consistency and detect conflicts for shared memory accesses in transactions. Read/write barriers are implemented for each basic data type in C/C++. For user-defined data constructs, `stmReadBytes` and `stmWriteBytes` are used as read/write barriers. The SpiceC compiler instruments atomic code with read/write barriers as follows:

1. *Find Shared Variables.* It is very important for the compiler to only place read/write barriers at necessary places since they are usually time-consuming at runtime. Instrumenting all accesses to global/heap variables with read/write barriers usually introduces unnecessary barriers since some of global/heap variables may be read-only in transactions, or may not be shared across transactions. To avoid placing unnecessary read/write barriers, the compiler uses static data race analysis [102] to find potentially shared variables, i.e., variables that two transactions may access without synchronization and one of the accesses is a write.

2. *Normalize Operators.* The compiler then normalizes C/C++ operators on these variables. In C/C++, a variable reference may be both a read and a write at the same time. Since the compiler works on C/C++ source code, it cannot directly insert read/write barriers for such variable references. For example, the reference to `a` in `a++` is both a read and a write. To enable barrier insertion, the compiler converts `a++` to `a=a+1`, where the first reference to `a` is a write and the second is a read.

3. *Insert Barriers.* Finally, the compiler inserts read/write barriers for accesses to the potentially shared variables found in step 1. For example, after instrumentation, `a=a+1` will be `stmWriteInt(tx, &a, stmReadInt(tx, a)+1)`, where `tx` is the transaction descriptor.

Although it is not necessarily to detect conflicts for writes to live-in private variables (including local variables on the stack, thread-local variables, and global variables that are not shared), the original values in these variables need to be logged to allow rollback if needed. The compiler inserts `stmLogStack` before each transaction to save the local state. Function `stmLogStack` uses the *ebp* and *esp* registers to locate and save the local variables on the stack. The compiler also inserts `stmLogBytes` before every write to thread-local variables and global variables that are not shared. The function saves the value at an address if the address has not been logged in the transaction.

Functions `stmMalloc` and `stmFree` are the STM versions of `malloc` and `free`. The compiler replaces `malloc` and `free` in transactions with the STM versions. The read barrier `stmDirectRead` is used for an optimization discussed in Section 3.3.

3.2 Support for Library Functions

In previous works [134, 93, 12], two types of functions cannot be transact-ified: precompiled library functions and functions that cannot be rolled back (e.g., system calls and I/O operations). Calls to these types of functions within transactions can be detected by either the compiler or the runtime system. If a transaction is identified to contain such function calls, it is executed in serial mode, i.e., other transactions cannot run concurrently with it. This makes it impossible to speculatively parallelize code that contains calls to precompiled or irreversible functions. This section introduces two SpiceC directives to avoid serialization of transactions when such functions are called in transactions.

3.2.1 Precompiled Library Functions

Construct `#pragma SpiceC precompiled` is designed to annotate precompiled library functions so that the compiler can transact-ify them. The syntax is given in Figure 3.3.

```
#pragma SpiceC precompiled [read(...)] [write(...)]
// function declaration here
```

Figure 3.3: Syntax of the `precompiled` construct.

The directive is introduced immediately preceding a function declaration, and tells the compiler what memory locations the function reads and writes. A memory location can be a variable or a (pointer,size) pair. After a precompiled function is annotated with the construct, it can be used as a regular function in transactions. In a transaction where the function is called, the compiler creates local copies of the shared memory locations that the function accesses. The function then works on the local copies instead of directly accessing the shared memory locations. In this way, the need to transact-ify the code inside the function is eliminated since the function does not touch shared memory locations. The compiler only needs to add read/write barriers when copying data between shared memory locations and local copies. The transactions with annotated library functions can thus be executed in parallel with other transactions.

```
#pragma SpiceC precompiled read(x)

float sin (float x);

...

#pragma SpiceC precompiled \

    read(dst, src, num, (*src, num)) \

    write((*dst, num))

char * memcpy(void *dst, void *src, size_t num);
```

Figure 3.4: Using the `precompiled` construct to annotate function declarations.

The code given in Figure 3.4 provides an example with the `precompiled` construct used to annotate two library functions. The first function is a mathematical function provided by the C numerics library. Argument `x` is placed in the `read` clause since it is read in the function. The `write` clause is omitted since no variable is written by the function. The second function is a memory copy function provided by C string library. It copies the values of `num` bytes from the memory location pointed to by `src` to the one pointed to by `dst`. Since the function reads the `num`-byte memory block pointed to by `src`, `(*src, num)` is placed in the `read` clause. Similarly, `(*dst, num)` is put in the `write` clause.

In certain cases it is impossible to know the memory locations accessed by a function until it is called. In such cases, programmers can use the `precompiled` construct to annotate the call site of the function. In Figure 3.5, the function copies a string pointed to by `src` into the array pointed to by `dst`. The string ends in a null character. It is impossible to know the memory size accessed by the function at its declaration, since the string length is not fixed. However, programmers can conservatively annotate the function at its call site since the maximum length of the string (i.e., the size of the allocated memory block) is known at that time.

Calls to an annotated library function need to be translated and instrumented with low-level STM API constructs for the purpose of conflict detection and potential rollback. Instrumenting calls to a precompiled function proceeds as follows:

1. *Log Values of Thread-local Variables.* For every thread-local variable in the write clause, the compiler uses the log functions to log their values as they may be needed in case of a rollback.

2. *Create Local Copies for Shared Variables.* For every shared scalar variable in the read and write clauses, the compiler creates a local variable on the stack. The compiler initializes the local copies of the variables in the read clause by using `stmRead`⟨`Type`⟩. For other shared variables (such as arrays, objects, and dynamic data structures) in the read and write clauses, the compiler uses `malloc` to allocate space for their local

```
char src[256], dst[256];

#pragma SpiceC region R1

{

        ... // statements here

        #pragma SpiceC precompiled \

            read(dst, (*src, 256), src) \

            write((*dst, 256))

        strcpy(dst, src);

        #pragma SpiceC commit atomicity_check

}
```

Figure 3.5: Using the `precompiled` construct to annotate function call sites.

copies and assigns the starting addresses to the corresponding pointers. Function `stmReadBytes` is called to copy data for these variables. The compiler replaces the function arguments with their local copies.

3. *Update Values of Shared Variables.* After the function is completed, the shared variables in the write clauses need to be updated with the values in their local copies. Therefore, the compiler inserts calls to `stmWrite`⟨`Type`⟩ and `stmWriteBytes` after the function call for shared variables in the write clause. Finally, the compiler frees all temporarily allocated variables.

Figure 3.6 the translated call to precompiled function `memcpy`, whose annotation was described before.

```
int l_num = stmReadInt(tx, &num);
void *l_src = (void*)malloc(tx, 256);
void *l_dst = (void*)malloc(tx, 256);
stmReadBytes(tx, (void*)l_src, (void*)src, 256);
memcpy(l_dst, l_src, l_num);
stmWriteBytes(tx, (void*)dst, (void*)l_dst, 256);
free(l_src); free(l_dst);
```

Figure 3.6: The translated call to precompiled function `memcpy`.

3.2.2 Irreversible Functions

In some transactions that call irreversible functions, other statements may not be data-dependent on the irreversible functions. For example, function `printf` only prints text on the screen but does not produce any data. Another example is function `system`, which invokes the shell to execute a system command. Other statements in the transaction may not depend on the system command. Therefore, in these transactions, the execution of such functions can be safely suspended during speculative execution. The input of these functions are buffered during the speculative execution and the functions are executed when the transactions have completed successfully.

Construct `#pragma SpiceC suspend` is designed to annotate the functions to be suspended during the speculative execution. Its syntax is similar to that of the `precompiled` construct except that there is no `write` clause; it annotates function declarations, as shown in Figure 3.7. The `putchar` function is an output function from the standard C library—it prints the character `c` to the current position in the standard output. Since the function does not use any pointer as argument, programmers can annotate it when it is declared.

```
#pragma SpiceC suspend read(c)
int putchar (int c);
```

Figure 3.7: Using the `suspend` construct to annotate function declarations.

Similar to the `precompiled` construct, the `suspend` construct can also be used at function call sites as shown in Figure 3.8. In the example, function `puts` writes the string

```
char str[256];

#pragma SpiceC region R1

{

  ...  // statements here

  #pragma SpiceC suspend read(str, (*str, 256))

  puts(str);

  #pragma SpiceC commit atomicity_check

}
```

Figure 3.8: Using the `suspend` construct to annotate function call sites.

pointed to by `str` to standard output. As the string ends in a null character, it is impossible to know the string length at the function declaration. Therefore, the function needs to be annotated at its call site, where the maximum length of the string is known.

Calls to an annotated irreversible function need to be translated and instrumented to enable suspending with transactions. The compiler instruments calls to irreversible functions as follows:

1. *Record Arguments.* For every variable in the read clause, the compiler pushes its value in the thread-local queue `args`. For shared variables, `stmRead`$\langle$`Type`$\rangle$ or `stmReadBytes` is called inside the push function to ensure consistency. These values will be used to invoke the irreversible function outside the transaction.

2. *Record Function.* The compiler replaces the function call with a statement that saves

the function identifier (generated from the function name for each annotated function) in the thread-local queue `funcs`. The function identifier will be used to invoke the irreversible function outside the transaction.

3. *Call Function Outside.* The compiler generates a wrapper function `resume_suspended_funcs` that goes through these queues and calls the irreversible functions they contain. The wrapper function will be called from `stmEnd` when the transaction is successfully committed.

The code given in Figure 3.9 shows the translated call to irreversible function `putchar`, whose annotation was described before.

A global lock is used to prevent the wrapper functions called in different transactions from interleaving. A thread needs to acquire the lock before performing a wrapper function during the commit phase.

The `precompiled` and `suspend` constructs can be used to annotate most C/C++ standard library functions. However, there is one case where these constructs cannot be used—the standard template library (STL). This is because some STL functions operate on linked data structures, hence it is difficult to determine the memory locations that are accessed prior to executing them.

3.3 Optimizations

Section 3.1.2 has presented the use of static data race analysis for reducing the number of read/write barriers inserted in the code. This section presents three other compile

```
stmBegin(tx);
... // statements here
args.sharedpush(tx, &c, sizeof(char));
funcs.push( F_PUTCHAR );
stmEnd(tx);
...
void resume_suspended_funcs() {
   while ( !funcs.empty() )
      switch ( funcs.pop() ) {
         case F_PUTCHAR:
            putchar( *((char*)args.pop()) );
            break;
         ...
      }
}
```

Figure 3.9: Translated call to irreversible function `putchar`.

time optimizations to reduce the time overhead incurred by STM.

3.3.1 Eliminating Redundant Barriers

During code instrumentation, redundant read/write barriers may be introduced in the code, which can significantly increase the STM overhead. Figure 3.10 shows two arithmetic statements and their intermediate code with read/write barriers.

```
Original code  Intermediate code
b = a+1;       stmWriteInt(tx, &b, \
                      stmReadInt(tx, &a)+1);
b = a*b;       stmWriteInt(tx, &b, \
                      stmReadInt(tx, &a)* \
                      stmReadInt(tx, &b));
```

Figure 3.10: An example of intermediate code with read/write barriers.

In the intermediate code, `read barrier` is called three times and `write barrier` is called twice. Actually, only two barriers are required in this code, one read barrier for `a` and one write barrier for `b` (all other barriers are redundant).

To eliminate redundant barriers in an expression, the compiler first creates temporary variables to hold the values loaded/stored by read/write barriers and uses the temporary variables in the expression. Figure 3.11 shows the previous intermediate code with temporary variables inserted for read/write barriers.

With temporary variables, read/write barriers are separated from the original statements. A read barrier can be eliminated if it is pre-dominated by read or write barriers to the same variable within the same transaction. A write barrier can be eliminated if it is post-dominated by write barriers to the same variable within the same transaction.

```
l_a = stmReadInt(tx, &a);
l_b = l_a+1;
stmWriteInt(tx, &b, l_b);
l_a = stmReadInt(tx, &a);
l_b = stmReadInt(tx, &b);
l_b = l_a*l_b;
stmWriteInt(tx, &b, l_b);
```

Figure 3.11: Intermediate code with temporary variables.

Figure 3.12 shows the final code generated for the previous example after elimination of redundant barriers.

```
l_a = stmReadInt(tx, &a);
l_b = l_a+1;
l_b = l_a*l_b;
stmWriteInt(tx, &b, l_b);
```

Figure 3.12: Final code after redundancy elimination.

3.3.2 Reducing Searches in Write Buffers

Since the overhead of write buffering-based STMs comes largely from searching write buffers [114], eliminating unnecessary data searches in write buffers can greatly improve performance. To eliminate unnecessary searches in write buffers, the SpiceC compiler checks the control flow for each read barrier. If a read barrier is not preceded by any write barrier to the same variable within the same transaction, it does not need to search the transaction's write-set since the variable cannot be in the write-set. The compiler replaces such read barriers with calls to `stmDirectRead⟨Type⟩`, which is a read barrier that reads a value in a regular memory location without searching the write-set.

3.3.3 Synchronization Between Transactions

Speculative parallelization often requires transactions to be committed in a specific order, to preserve the sequential semantics of the original program. One way to enforce a commit order between transactions is to place synchronization code prior to the end of the transaction. However, this has two major drawbacks. First, performing synchronization in transactions introduces extra overhead since synchronization code usually does not need to be executed speculatively. Second, performing synchronization in transactions may cause transactions to abort, due to inconsistent states of shared data structures used for synchronization. Therefore, it increases transaction abort rates.

For example, in Figure 3.13, busy-waiting is used to synchronize the transaction commit. In the example, the transaction will not commit until a signal is received. Let us assume that when the transaction enters the busy-waiting loop, variable `signal`'s value is 0. The value of `signal` can be changed to 1 after the transaction has started, e.g., by another transaction. This leads to an inconsistent memory state since the transaction eventually sees two values ('0' and '1') for variable `signal`. Most STM implementations,

```
#pragma SpiceC region R1
{
        ... // statements here
        while (signal == 0);
        #pragma SpiceC commit atomicity_check
}
```

Figure 3.13: Busy waiting inside a transaction.

such as Transactional Lock II (TL2) [41], will abort the transaction in this case since inconsistent memory state may trigger a fatal exception (e.g., segmentation fault) or cause the transaction to enter an endless loop.

SpiceC introduces a directive, `#pragma SpiceC beforevalidate`, which specifies code that is executed non-speculatively *immediately before* the atomicity check is performed. The `beforevalidate` construct is designed for synchronization between transaction commits to keep the sequential semantics of the original program.

For example, the preceding synchronization code can be written as shown in Figure 3.14. In the example, the busy-waiting loop is executed non-speculatively before transaction validation and commit. Therefore, it will not cause extra overhead or increase transaction abort rate.

It should be noted that this construct is different from *commit handlers* proposed in previous works [85, 12]. The construct is used for synchronization to preserve the sequential semantics of the original program. Therefore, the statement block specified by

```
#pragma SpiceC region R1
{
        ... // statements here
        #pragma SpiceC commit atomicity_check
}
#pragma SpiceC beforevalidate
{
        while (signal == 0);
}
```

Figure 3.14: Busy waiting using the `beforevalidate` construct.

`beforevalidate` is executed before transaction validation. In contrast, a *commit handler* allows the programmer to specify code that runs once a transaction is known to have completed successfully, hence a *commit handler* is executed after transaction validation.

3.4 Safety

This section presents the solutions to two safety issues during compilation.

3.4.1 Aliasing of Shared Variables

Temporary variables introduced by redundancy elimination (as described in Section 3.3) may cause aliasing issues. For example, the snippets given in Figure 3.15 show original code that manipulates two pointers (left) and hypothetical code that would be generated if aliasing was not handled safely (right). The hypothetical code eliminates the read barrier for the third statement `a=*p` since it is pre-dominated by a write barrier to the same location. However, in this example, if `p` and `q` point to the same memory location, the value of `a` in the hypothetical code will be wrong since `a` always gets its value from `t1`. To solve this problem, a temporary variable that holds data pointed to by a pointer is killed when another pointer is dereferenced.

```
Original code  Hypothetical (unsafe) code
               t1 = f();
*p = f();      stmWriteInt(tx, p,
*q = g();      t1);
a = *p;        t2 = g();
               stmWriteInt(tx, q,
               t2);
               a = t1;
```

Figure 3.15: An example of aliasing.

3.4.2 Exception Handling

Exceptions in transactions require careful consideration and treatment since exception objects may contain data from speculative states. Uncaught exceptions that propagate out of a transaction may pollute the non-speculative state with data from the speculative state and thus causes inconsistent memory state. The SpiceC compiler inserts code to catch all exceptions in transactions. When an exception is caught, the function `stmEnd` is called to validate the transaction. If the transaction passes validation, the transaction is committed and the exception is transferred to the handler outside the transaction; otherwise, the transaction is rolled back and the exception is discarded.

3.5 Evaluation

This section evaluates the SpiceC compiler for generating speculatively executed code. The experiments were conducted on an 8-core DELL PowerEdge T605 machine. Table 3.3 lists the details of the machine. The machine ran *CentOS v5.5*.

Ten applications were used to evaluate the SpiceC compiler: eight STAMP benchmarks and two real applications—Velvet and ITI. Table 3.4 lists their description. This section first presents the experimental results for the STAMP benchmarks. It then introduces the real applications, explains how they were parallelized using our approach, and presents the results of performance evaluation.

3.5.1 STAMP Benchmarks

STAMP is a benchmark suite designed for STM research. It consists of eight parallel benchmarks [91]. All STAMP benchmarks are originally instrumented with low-level STM constructs. To evaluate the SpiceC compiler, the low-level STM constructs in the STAMP benchmarks were replaced with the SpiceC directives. Each benchmark has two input data sets. In the experiments, the larger input data set is used since it is more suitable for experiments on real machines. Two of the benchmarks, `Kmeans` and `Vacation`,

Processors	2×4-core AMD Opteron processors (2.0GHz)
L1 cache	Private, 64KB for each core
L2 cache	Private, 512KB for each core
L3 cache	Shared among 4 cores, 2048KB
Memory	8GB RAM

Table 3.3: Machine details.

Program	Source	Programming effort					
		STM API		Intel STM	*SpiceC compiler*		
		barrier	other		boundary	precomp.	irrev.
`Bayes`	STAMP	49	127	75	15	0	0
`Labyrinth`	STAMP	36	62	31	3	0	0
`Genome`	STAMP	60	62	31	5	0	0
`Intruder`	STAMP	75	190	95	3	0	0
`Kmeans`	STAMP	8	9	3	3	0	0
`Ssca2`	STAMP	24	31	10	10	0	0
`Vacation`	STAMP	63	296	151	3	0	0
`Yada`	STAMP	99	225	115	6	0	0
`Velvet`	real	–	–	–	2	0	9
`ITI`	real	–	–	–	1	2	30

Table 3.4: Benchmark summary and programming effort (number of constructs) when using the low-level STM API, the Intel STM compiler, and the SpiceC compiler. Programming effort using the low-level STM API is split into constructs used for read/write barriers and others, while programming effort using the SpiceC compiler is split into constructs used for transaction boundaries, precompiled functions, and irreversible functions.

have inputs with low contention and inputs with high contention. Both were used in the experiments to show the performance on different contention levels.

Programming Effort

To compare the programming effort with prior approaches, Table 3.4 shows the number of programming constructs inserted into the benchmarks, using the low-level STM API, the Intel STM compiler, and the SpiceC compiler. The reported numbers include the number of programming constructs inserted in the library functions called in these benchmarks. With the low-level STM API, programmers must define transaction boundaries, insert read/write barriers, and add STM-related arguments to function declarations. When using the Intel STM compiler, programmers must define transaction boundaries and annotate the functions called within transactions. The SpiceC compiler only requires programmers to define transaction boundaries (i.e., the `region` directives).

Compared to the low-level STM API, the SpiceC compiler requires on average 97% fewer programming constructs to be inserted into each benchmark. For each benchmark in the PARSEC benchmark suite, the number of accesses to global/heap variables within transactions was counted. On average each transaction contains 144 accesses to global/heap variables, out of which 81 are shared accesses. To achieve better performance with the low-level STM API, programmers need not only place read/write barriers, but also put effort in examining all the accesses to global/heap variables to see whether they are shared accesses. The SpiceC compiler liberates programmers from the burden of identifying shared accesses and placing read/write barriers. Compared to the Intel STM compiler, the SpiceC compiler requires on average 91% fewer programming constructs to be inserted into each benchmark. With the SpiceC compiler, programmers do not need to annotate any function called within transactions. *Overall, the SpiceC compiler only requires 3–15 programming constructs for each benchmark.*

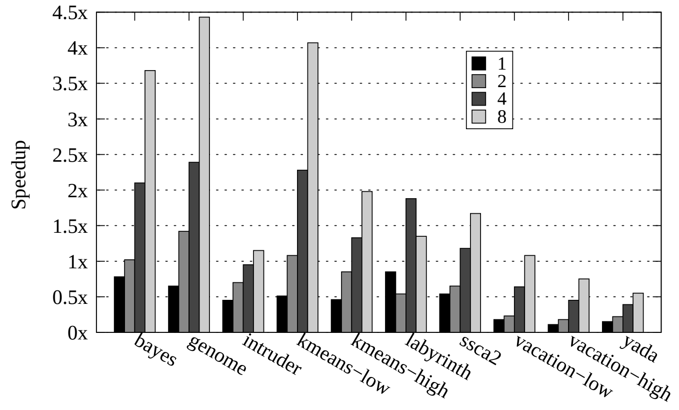


Figure 3.16: Speedups over sequential versions of the same programs achieved by the SpiceC compiler using 1, 2, 4, and 8 threads.

Performance

Figure 3.16 shows the speedups achieved for each STAMP benchmark using different numbers of threads. For each benchmark, the baseline is the single-threaded original version of the program. Numbers greater than 1 reflect better performance than the single-threaded original versions. The numbers are measured using all optimizations. With 8 threads, the SpiceC compiler achieves speedups for all STAMP benchmarks except `vacation-high` and `yada`. For `vacation-high` and `yada`, the large number of read/write barriers imposes a significant performance penalty. *On average, the SpiceC compiler achieves* 1.65x *speedup for the STAMP benchmarks with 8 threads.*

Optimization Benefits

The benefits of various optimizations are evaluated. The optimizations include using static data race analysis (DRA) to avoid placing barriers for accesses that do not cause data races, caching the values loaded/stored by shared reads/writes to eliminate redundant barriers (ERB), and eliminating unnecessary searches (EUS) in write buffers for data that are definitely not present in write buffers.

Table 3.5 compares the number of read/write barriers inserted with different optimizations. In the baseline, read/write barriers were inserted for every access to global/heap data, which are potentially shared by threads. As we can see, in the baseline, each benchmark has on average 144 read/write barriers. DRA reduces the number of barriers by 25.3% on average for these benchmarks. The number of barriers inserted in `yada` is reduced significantly since `yada` operates on numerous locally-allocated objects. ERB further reduces the

Program	Baseline	DRA	DRA+ERB
Bayes	172	87	71
Labyrinth	107	67	67
Genome	40	40	37
Intruder	183	152	148
Kmeans	7	7	7
Ssca2	24	24	24
Vacation	196	180	156
Yada	424	304	289
Average	*144*	*108*	*100*

Table 3.5: Number of read/write barriers inserted with various optimizations.

number of barriers by 7.2%. EUS does not reduce the number of barriers since it only eliminates the unnecessary searches in write buffers. Since `kmeans` and `ssca2` have only small transactions that do not call any functions, their baseline does not contain any barriers to eliminate. *Overall, the optimizations eliminate* 30.7% *of the barriers from the baseline.*

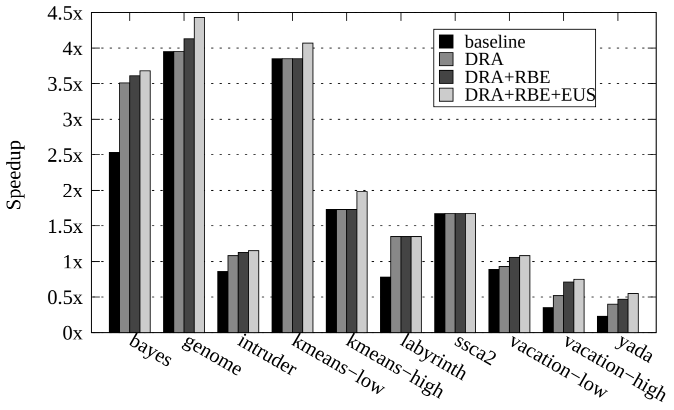


Figure 3.17: Performance with different optimizations.

Figure 3.17 shows the impact of various optimizations on program performance. The numbers were measured using 8 threads. In the baseline, read/write barriers are inserted for every access to global/heap data. DRA improves the performance by 23.3% on average for these benchmarks. The performance of `bayes` and `labyrinth` is improved most since a lot of barriers in them are eliminated by DRA. ERB and EUS further improve the performance by 7.4% and 5.4% for these benchmarks. EUS significantly improves performance for `genome` and `kmeans` since they have shared read-only data. *Overall, the optimizations improve the performance by* 32.8% *on average for these benchmarks.*

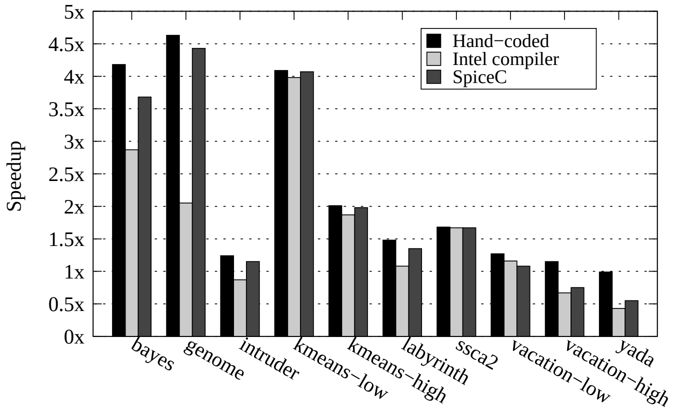


Figure 3.18: Performance comparison of the hand-coded transactional code, the Intel STM compiler, and the SpiceC compiler.

Comparison

Figure 3.18 compares the performance of hand-coded transactional code using the low-level STM API, the Intel STM compiler, and the SpiceC compiler. The STAMP benchmark suite provides hand-coded transactional code via low-level STM constructs. To measure the performance of the Intel STM compiler, their programming constructs were applied to the STAMP benchmarks. The numbers were measured using 8 threads. Among the three implementations, the hand-coded transactional code achieves the best performance. This is expected since a lot of effort has been spent on manually optimizing the hand-coded transactional code provided by STAMP. For example, there are no redundant barriers in the hand-coded transactional code. The performance of the SpiceC compiler is quite close to that of the hand-coded transactional code. *Compared to the hand-coded transactional code, the programs generated by the SpiceC compiler are slower by* 14.3%*. Compared to the Intel STM compiler, the SpiceC compiler improves the performance by* 20.8%*.*

3.5.2 Real Applications

This section evaluates the SpiceC compiler using two real applications, `Velvet` and `ITI`. These applications are used to show that the SpiceC compiler has low programmer burden and can improve performance for loops with precompiled and irreversible functions. The low programming burden is evident in the last two rows of Table 3.4: users only need to add just 11 constructs for Velvet and 33 for ITI. Note that, since low-level STM APIs cannot be used to transact-ify irreversible functions, they cannot be used for these two real applications (see '–' entries in Table 3.4). Similarly, the Intel STM compiler cannot be used to parallelize these applications because the compiler serializes the execution of transactions that contain precompiled and irreversible functions.

Velvet Genomic Assembler

`Velvet` [143] is a widely-used genomic assembler designed for short read sequencing technologies. Due to its popularity, developers have put a lot of effort into parallelizing it.

In the latest version of `velvet` (version 1.1), fine-grained locks (i.e., one lock for each shared object) are used to parallelize two loops in function `fillUpGraph`. The two loops account for approximately 50% of the execution time. To apply fine-grained locks, 257 lock-related statements were added by the Velvet developers. Apart from the extra code, the programmer must ensure that there is no deadlock and no livelock. Lock contention needs to be managed to maximize program performance. For example, programmers must determine whether threads should spin or block when acquiring a lock.

The SpiceC compiler was used to speculatively parallelize the two loops. Each iteration of the two loops is treated as a transaction. Numerous output operations (e.g., `fprintf`) and system calls (e.g., `sysconf`) are called inside the two loops. Since these functions are irreversible, 9 programming constructs were inserted to suspend and perform the functions outside the transactions. Out of the 9 inserted constructs, 3 were used to annotate system calls and 6 were used to annotate output operations. In total, 11 programming constructs were inserted in `velvet`, including 2 for annotating the transaction boundaries.

Figure 3.19 compares the performance of the SpiceC compiler and fine-grained locks (FGL). The baseline is the sequential version of `velvet`. The numbers were measured using nucleotide sequence *SRR027005* [2] as input. With only 11 programming constructs added, the SpiceC compiler achieves 1.48x speedup using 8 threads. Compared to code with fine-grained locks, the code generated by the SpiceC compiler is slower by only 11.4%. Considering the significantly lower programming effort, the SpiceC compiler provides an easy way to parallelize real applications.

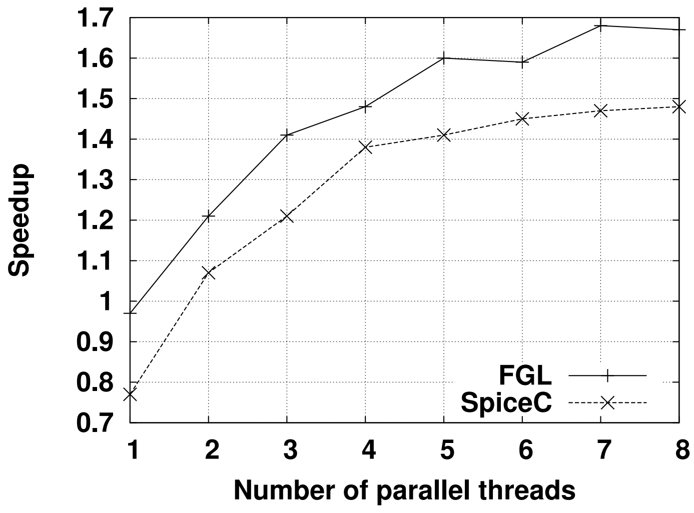


Figure 3.19: Speedups of Velvet over its sequential version.

Incremental Tree Inducer

Incremental tree inducer (ITI) [131], also called Direct Metric Tree Induction (DMTI), is a widely-used decision tree constructor; it incrementally constructs decision trees from labeled examples. The application has not been parallelized before.

The SpiceC compiler was used to speculatively parallelize the main loop, each iteration of which reads a labeled example and updates the decision tree. The loop body is annotated as a transaction. Both precompiled and irreversible library functions are used inside the loop. Two C string functions, `strlen` and `strcmp`, are called inside the loop. Since the string length is known, 2 `precompiled` constructs were used to annotate their declarations. C standard output function, `printf`, is also called inside the loop. Since different calls to `printf` use different formats of arguments, 30 `suspend` constructs were inserted to annotate its call sites. In total, 33 programming constructs were added, including

1 for annotating the transaction boundaries.

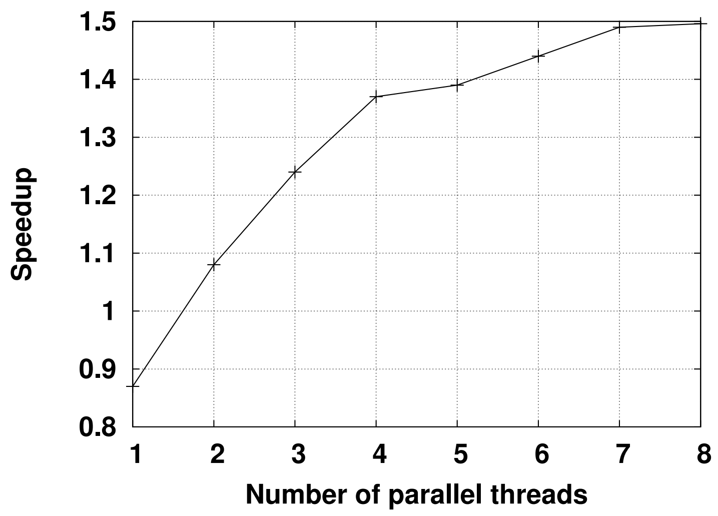


Figure 3.20: Speedup of ITI over its sequential version.

Figure 3.20 shows the speedups achieved by the SpiceC for `ITI`. The baseline is the single-threaded original version of `ITI`. Data set `agaricus lepiota` [6] was used as input. The SpiceC compiler achieves a 1.50x speedup using 8 threads, which demonstrates that it can be used to parallelize real applications that contain both precompiled and irreversible functions.

3.6 Summary

This chapter has presented the SpiceC compiler, which automatically inserts STM constructs into C/C++ code to enable speculative execution. Compiler directives are provided to support parallel execution of transactions that contain precompiled and irreversible library functions. A set of compiler optimizations are proposed to improve transaction performance.

Chapter 4

Support for Dynamic Linked Data Structures

Computation distribution is crucial to the performance of parallel applications. Efficient parallel execution requires exploiting locality of data references in the process of computation distribution. Many sequential programs execute the computations in an order that exploits spatial data locality. However, when parallelizing these programs, if consecutive computations are assigned to different threads, the resulting data contention will harm performance.

Data partitioning based computation distribution [9] has been proposed to improve the performance of parallel programs in distributed memory systems. This approach first partitions the data, then assigns partitions to threads, and finally assigns computations to threads such that the thread that owns the data required by a computation performs the computation. Programming this strategy is not a trivial task. Developers need to write code for: partitioning and assigning the data to threads; and distributing computation among threads based on the partitioning. They also need to enforce synchronization between computations for different partitions. A few programming models have been proposed to explore data and computation distribution. Most of them focus on array-based parallel programs [66, 110, 34, 44, 57, 27, 28]. The Galois system [71] employs data partitioning for irregular JAVA applications. It is designed for exploiting worklist-based parallelism where an iterative program processes work items from a worklist. Galois requires programmers to specify the relationship between method calls and how to undo modifications for shared data structures. Compared to C++ programs, JAVA programs are less challenging since JAVA supports more OO features (e.g., *properties*) and does not have pointer variables.

This chapter presents SpiceC support for exploiting parallelism in C++ programs that rely upon the use of dynamic linked data structures. In dynamic linked data structures each node is connected to a fixed number of neighboring nodes and the computation performed on the nodes progresses across the nodes by traversing the links. SpiceC exploits this characteristic in carrying out data distribution and computation distribution among threads. Consequently, when speculation is employed, high misspeculation rates are avoided.

4.1 Overview

SpiceC provides efficient support for exploiting data parallelism in a program that relies on the use of a dynamic linked data structure. With SpiceC, computations can be distributed across threads that operate upon different data partitions created by dividing the dynamic linked data structure. The runtime execution flow of the program is as follows. A set of threads are created at the beginning of the program. Each thread is bound to a unique processor core. A master thread executes the sequential part of the program. When encountering a parallel region identified by the programmer, the master thread divides the data into multiple partitions via partitioning strategy selected by the programmer and maps each partition to one of the threads. Within a parallel region, each thread typically performs the computations that work on its assigned data partition and it either skips or redistributes the computations that work on other partitions. To support the above execution model, SpiceC provides the following features:

I. Partitioning support. SpiceC provides four data partitioning strategies to handle different data structures and computation patterns. METIS and HASH are two partitioning strategies for graph data structures with each computation working on one or more nodes. SYMM_SUBTREE and ASYMM_SUBTREE are two partitioning strategies for tree data structures with each computation exhibiting locality on a subtree. SpiceC also allows the programmers to specify custom data partitionings.

II. Homogeneous parallel regions. In this form of parallelism, every thread executes the same code after entering the parallel region. Although all threads execute the same code, different threads perform computations on different data partitions. This parallelism is similar to the Single Program Multiple Data (SPMD) parallelism. The core SpiceC programming model is extended to provide constructs for checking if the data required by a computation (i.e., an iteration of the loop) is located in the current thread's data partition. At runtime, a thread performs this check at an early stage of each computation. If the data required by a computation is located in the thread's partition, the thread continues executing the computation. Otherwise, the thread skips to the next computation. This execution model is often used for graph data structures with each computation starting from a different node.

III. Heterogeneous parallel regions. This form of parallelism is suitable for computations that always start from the same node (e.g., a search from the root of a tree). Homogeneous parallelism is not suitable here since the thread owning the starting node will end up performing all computations. After entering a heterogeneous parallel region, only the master thread starts to execute the code in the parallel region and all other threads are put in idle state. When the master thread comes across a computation that requires data from another thread's partition, it assigns the computation to the corresponding thread and continues to execute the code following the assigned computation. The thread that receives the computation then performs it in parallel with other threads. All threads are able to distribute computations to other threads based on the data required by the computations.

IV. Speculative parallelism. Sometimes, a computation may need to access and update the data from multiple partitions, where some partitions belong to other threads. For example, in a graph data structure, a computation may start from one node but end up with updating multiple nodes around the starting node. The thread that performs the computation may have to access the data from a partition assigned to another thread. In this case, it is possible that multiple parallel threads access the same data simultaneously. The core SpiceC system provides speculation support for this type of computation to resolve any data conflict between two threads. To improve the performance of speculative computation on dynamic linked data structures, a speculation mechanism called *conditional speculation* is introduced to SpiceC. It selectively applies speculation on computations to reduce the speculation cost.

The following sections present the partitioning strategies supported by SpiceC and programming support given to the developer in form of pragmas.

4.2 Partitioning Support

Let us consider the strategies for partitioning data objects belonging to a dynamically created linked data structure. SpiceC provides programmers four partitioning strategies that can be used to handle commonly occurring scenarios involving graphs and trees used by a wide range of applications. These strategies include: METIS and HASH for graphs; and SYMM_SUBTREE and ASYMM_SUBTREE for trees (see Table 4.1). SpiceC also provides the programmer with the ability to specify custom data partitioning strategies. The remainder of this section discusses the partitioning strategies in detail.

Strategy	Struct.	Selection Criterion	Benefits
METIS	Graph	Spatial Locality Present	Locality[1]; Speculation[2]
HASH	Graph	No Spatial Locality	Locality[1]
ASYMM_SUBTREE	Tree	Searches Originating at Internal/Leaf Node	Locality[1]; Speculation[2]
		Recursively Parallel Computation	Locality[1]
SYMM_SUBTREE	Tree	Searches Originating at Root Node	Locality[1]; Speculation[2]

[1]Improved cache locality. [2]Reduced speculation cost.

Table 4.1: Summary of partitioning strategies.

4.2.1 Graph Partitioning

Graphs are widely used in the design of algorithms. A graph data structure is defined as a set of data objects connected by edges. Each object is called a node or a vertex belonging to the graph. A graph may be directed or undirected. The access patterns of graph data structures are divided into two categories according to the presence or absence of *spatial locality* in the access patterns. *Spatial locality means that if a node is accessed by a computation, then it is likely that nodes adjacent to it in the graph will also be accessed during the same computation.* The remainder of this section gives examples that show that in the presence of spatial locality, METIS is a good partitioning strategy while in the absence of spatial locality HASH is a good strategy.

I. METIS. A computation on a graph data structure often requires accessing a set of nodes that are connected by edges. Often a computation begins by accessing one or more nodes in the graph. It then gradually involves more and more nodes that are adjacent to the beginning nodes. Thus, the access pattern exhibits spatial locality with respect to the graph structure. Figure 4.1 shows an example of the graph data structure where accesses exhibit spatial locality. A dark node indicates that the node is required by two different computations. Each computation requires accessing a connected subgraph and thus exhibits spatial locality with respect to the graph.

For this access pattern, if the computations are randomly assigned to threads, many cache coherence misses may occur. To improve the locality, users can group nodes that are near each other into partitions and assign each partition to a thread. Each thread

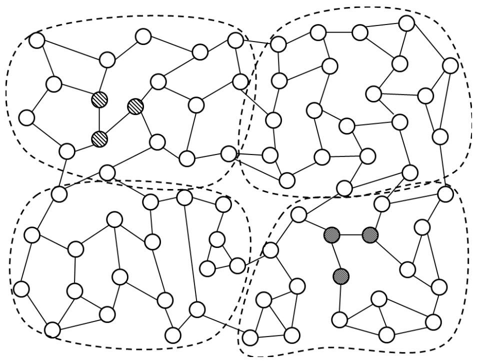


Figure 4.1: Parallel computation in a graph with spatial locality.

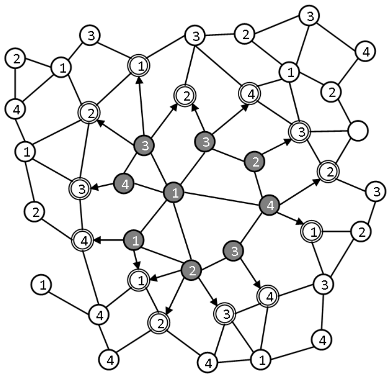


Figure 4.2: Breadth first search - an example of parallel computation in a graph without spatial locality.

can then perform computations that start from the nodes in its own partition. Since there is greater chance that each thread will access its own partition, the cache locality is enhanced. When threads are executed speculatively in parallel, this approach will result in low misspeculation rate since it is less likely that two parallel threads will access the same nodes.

Many algorithms have been proposed for partitioning graph data structures. SpiceC adopts the METIS algorithm, which is based on the state-of-the-art multilevel graph partitioning [63]. The METIS produces very high quality partitionings for large graphs. As it operates with a reduced-size graph, it is extremely fast compared to traditional partitioning algorithms. Moreover, METIS has a parallel implementation, which is suitable for parallel programs.

In addition to partitioning an existing graph, users must also consider *repartitioning* as a result of node insertions. Since node insertions occur frequently for some applications (e.g., Delaunay Mesh Refinement [59]), the repartitioning is required to be inexpensive. Instead of repartitioning the whole graph, we simply incorporate the new nodes into existing partitions. A new node is assigned to the partition, to which most of its neighbors belong. In this way, SpiceC minimizes the number of edges that cross partition boundaries.

II. HASH. In some parallel applications, a computation block (e.g., a loop iteration) only accesses one node in the graph data structure. Thus this access pattern is without spatial locality. Figure 4.2 shows an example of parallel breadth first search (BFS) (see Figure 4.8 for pseudocode). The dark nodes are the nodes that have been visited. The double circles mark the nodes that need to be visited in parallel. Since each computation only accesses one node, grouping nearby nodes cannot improve the cache locality in this case. Besides, grouping nodes that are close to each other may even make the workload unbalanced since it is likely that the nodes that need to be processed in parallel are connected. For this type of access pattern, users just need to hash the nodes into partitions. This partitioning strategy is called as HASH. Each thread is assigned a partition constructed via hashing and the thread only processes the nodes in its own partition. This approach not only balances the workload but also improves cache locality if multiple BFSs need to be performed on the graph. In this scheme, partitioning and *repartitioning* are both performed via hashing.

4.2.2 Tree Partitioning

Tree data structures have been widely used to represent hierarchical structures. Two partitioning strategies are proposed for tree structures which are described next.

I. ASYMM_SUBTREE. This partitioning strategy supports two types of tree computations: computations that start from an internal/leaf node and recursive computations. First let us consider the computations that start from an internal/leaf node and then travel within a local subtree that contains the node. Figure 4.3 shows two examples of such computations – one computation begins at an internal node and the other at a leaf node but both computations only touch a subtree instead of the whole tree. This behavior is typical of searches on trees such as quadtrees and KD-trees. For a program that performs a large number of such computations at runtime, performing these computations in parallel could greatly improve the performance. In this scenario users would like repeated computations on the same subtree to be performed by the same thread to achieve improved cache locality and also reduce the chance of two parallel threads accessing the same data. To achieve this SpiceC supports a partitioning strategy that forms partitions from subtrees as illustrated by the partitions shown in Figure 4.3 by dotted lines. Since this strategy creates subtrees that are not of exact same size it is named ASYMM_SUBTREE. The resulting subtrees are assigned to different threads, which can operate on their respective partitions in parallel. A computation is performed by the thread that owns the starting node. For example, in Figure 4.3, the computation on the left side is performed by the thread owning the leftmost partition and the computation on the right side is performed by the thread owning the rightmost partition.

Now let us consider the case of recursive parallel computations. In such computations, typically, the starting thread first assigns all of the child subtrees except one to other parallel threads and then continues to process the one left. The threads assigned the child

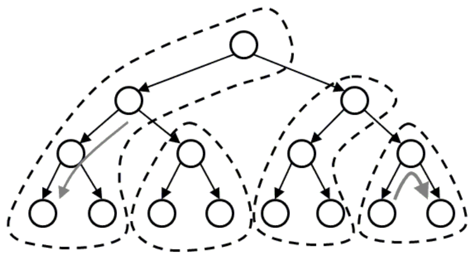


Figure 4.3: Parallel computations starting at an internal/leaf node.

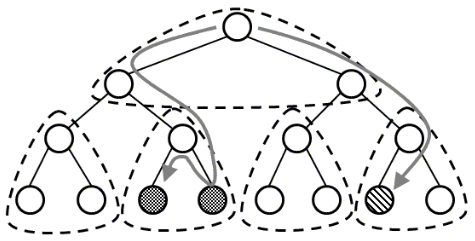


Figure 4.4: Parallel computations starting from the root node.

trees repeat the above procedure until no more threads are available to assign part of the work. Such recursive parallel code is invoked multiple times in many programs that have been examined. If threads are randomly assigned to subtrees for each invocation of the parallel code, the opportunities of reusing the data in the cache across multiple invocations will be lost. Even if the subtrees are assigned to threads in the same fixed order during each invocation, the mapping of partitions to threads may vary across different invocations. To improve data reuse, the tree can be partitioned as shown in Figure 4.3, and at runtime, require each thread to process the same subtree.

Figure 4.5 shows the algorithm for asymmetric tree partitioning shown in Figure 4.3. The algorithm starts with creating a new partition for the root. When the number of partitions is smaller than the limit, the algorithm assigns a node's leftmost child to the same partition and creates new partitions for all other children. Once the number of partitions reaches a limit, no new partitions are created. Each of the remaining nodes is assigned to its parent's partition. The *repartitioning* strategy allows for leaf node insertion and it assigns a new leaf node to its parent's partition.

```
root.partition $\leftarrow$ 0;
partitions $\leftarrow$ 1;
nodequeue $\leftarrow$ {root};
while ( nodequeue $\neq \phi$ ) {
  nodequeue.pop(n);
  foreach child $\in$ n.childset do {
    nodequeue.push(child);
    if (child is leftest OR partitions = total_partitions)
      child.partition $\leftarrow$ n.partition;
    else {
      child.partition $\leftarrow$ partitions;
      partitions $\leftarrow$ partitions + 1;
    }
  }
}
```

Figure 4.5: Asymmetric tree partitioning algorithm.

II. SYMM_SUBTREE. Most tree algorithms always start at the root node. Figure 4.4 shows two examples of tree computations starting at the root node. The right arrow indicates a tree computation that goes from the root to a leaf node and updates the value at the leaf node (e.g., binary insertion). The left arrow indicates a tree computation that goes from the root, searches a subtree, and finally updates the values at several nodes (e.g., during local tree balancing). A tree-based program typically begins many computations starting from the root node at runtime. Performing these computations in parallel can greatly improve its performance. For such computations, if each thread only processes the operations on a certain subtree, then it is possible to reduce the misspeculation rate or even do without speculation (e.g., during binary search). This also improves the cache locality. Since every computation starts from the root node, all computations will be started in the thread that owns the root node. Poor performance will occur if ASYMM_SUBTREE partitioning algorithm is used in this case. For example, under the partitioning of Figure 4.3, all computations will start in the thread owning the leftmost partition. If a computation goes right, the thread will assign it to another thread. However, if a computation goes all the way left, all subsequent computations will be blocked until it is completed. Suppose a computation has equal chance of going left and right, around 1/4 of the computations will block subsequent computations. This will greatly reduce the parallelism. Moreover, if a computation goes all the way to the right, it will go through three threads and incur communication overhead.

To address the above problem SpiceC introduces a strategy that partitions the tree into symmetric subtrees as shown in Figure 4.4 — it is named SYMM_SUBTREE as it partitions the tree in symmetric subtrees. A *root partition* is created to contain the nodes at the top few levels of the tree while each *subtree partition* contains a subtree under the root partition. At runtime, each thread is assigned a subtree partition. The root partition can be assigned either to the master thread or shared by all threads. Figure 4.6 presents the symmetric tree partitioning algorithm. The algorithm starts with assigning nodes to the root partition from the top of the tree until reaching a level, at which the number of nodes is greater that the number of desired partitions. It then evenly distributes those nodes among the partitions. It is not necessary to assign the remaining nodes to the partitions since the searches always go from the top to the bottom.

4.2.3 Programmer-specified Partitioning

Programmers often have the knowledge of the shape of the data structures that can be exploited by them during partitioning. To support this scenario SpiceC also allows programmers to specify their own partitioning strategies. Figure 4.7 shows two partitioning examples that can be easily specified by programmers. Figure 4.7(a) shows a hash table that consists of an array of linked lists. The programmer can easily group the linked lists by hashing the array indices into partition IDs. At runtime, each thread only processes hash operations in its own partition. This partitioning will improve the cache locality and avoid data contention. Figure 4.7(b) shows an example of a grid computation. The computation on each node requires the data from its neighbors. The programmer can easily group nodes near each other into the same partition given the position of each node. Although both examples can be partitioned using METIS, programmer specified partitionings as shown

```
nodeset ← root;
while ( |nodeset| < total_partitions ) {
  childset ← $\phi$;
  foreach node ∈ nodeset do
    childset ← childset + node.childset;
  nodeset ← childset;
}
size ← ⌈total_partitions / |nodeset|⌉;
partitions ← 0;
while ( partitions < total_partitions ) {
  for i ← 1 to size do
    if ( nodeset ≠ $\phi$ ) {
      nodeset.pop(n);
      n.partition ← partitions;
    }
  partitions ← partitions + 1;
}
```

Figure 4.6: Symmetric tree partitioning algorithm.

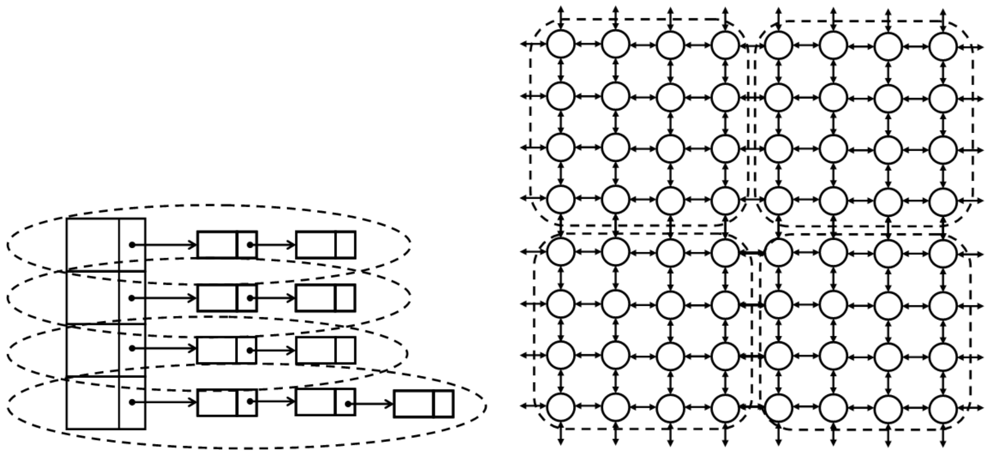

(a) Hash table (b) Grid computation

Figure 4.7: Example of manual partitioning.

are even superior.

4.3 Programming Support

This section presents the SpiceC directives designed for using partitioning based approach in programming applications. The presentation of these pragmas is divided according to the types of parallelism they support: *homogeneous*, *heterogeneous*, and *speculative*. Examples are given to illustrate the use of these pragmas in variety of application scenarios.

To enable partitioning of a linked data structure, the partitioning algorithm must be able to identify the data structure, i.e, its nodes and edges. To allow this SpiceC requires that the data structures that need to be partitioned inherit a base class, called `BaseNode`. The `BaseNode` class defines two interfaces (i.e., virtual functions in C++) – `GetNeighborNum` and `GetNeighbor` which developers must implement. The first interface returns the total number of neighbors and the second returns a specified neighbor. The SpiceC runtime library can acquire the graph information by calling these two functions. Besides, the implementation of the `BaseNode` should automatically maintain a list of nodes that have not been partitioned. The `BaseNode` class should also provide two member functions – `GetPartition` and `SetPartition`. These two functions are used to get and set the partition to which a data object belongs.

4.3.1 Non-speculative Homogeneous Parallelism

```
#pragma SpiceC parallel homo partition ([DataType],[PartitionType])
#pragma SpiceC region [RegionName] inpartition ([Partition])
```

The first pragma is used to mark the parallel code region, such as a loop, that is to be executed in parallel and where partitioning is to be applied. The `homo` clause indicates that homogeneous parallelism is to be used and the `partition` clause indicates the type of partitioning where the parameter `DataType` gives the type of the data structures to be partitioned and the parameter `PartitionType` identifies the partitioning strategy to be used. The second pragma is used to check in which thread the subsequent region is to be performed. The pragma checks if the partition specified by the parameter belongs to the current thread. If so, the current thread continues the subsequent computation. Otherwise, the current thread skips the rest of the computation and goes to the next computation. The `Partition` parameter can be given in the form of constant, variable, or return value of a function. Next the use of above pragmas is illustrated through examples.

```
read(graph);
vertexset.add(source);
while( !vertexset.empty() ) {
  nextset.clear();
  #pragma SpiceC parallel homo partition(Vertex, HASH)
  for(int i=0; i<vertexset.size(); i++) {
    #pragma SpiceC region R1
      v = vertexset[i];
    #pragma SpiceC region R2 inpartition(v.GetPartition())
    if ( !v.visited) {
      v.visited = true;
      process(v);
      nextset.lockedAdd(v.adjacentset());
    }
  }
  vertexset = nextset;
}
```

Figure 4.8: Pseudocode for parallel BFS with graph partitioning.

I. Parallel loop with graph data structures. Figure 4.8 shows the pseudocode of a parallel implementation of Breadth First Search (BFS). The vertexset and nextset are containers that store unique vertices. The starting node is placed in the vertexset at the beginning. In the parallel loop, each iteration processes one node in the vertexset and adds pointers to its neighbors to the nextset. The program does not examine the visited bit of a neighbor when adding it to the nextset to avoid excessive cache coherence misses. After the loop has completed, the program moves the nextset to the vertexset and repeats the loop. The program ends when all nodes have been visited.

HASH partitioning can be used for the graph since each computation of the parallel BFS just accesses one node. Partitioning can improve the data reuse if the BFS is called multiple times. Two pragmas are used to realize the partitioning-based parallelism in this case. The first pragma declares that the code region must be executed in the form of homogeneous parallelism and the graph must be partitioned using the HASH strategy. Partitioning will only be performed once by the program since the unpartitioned node list will be empty after the first partitioning. The second pragma checks if the vertex belongs to the current thread's partition. If so, the current thread continues the iteration. Otherwise, the current thread skips to the next iteration. The function `GetPartition` returns the vertex's partition that is computed by the first pragma. At runtime, the master thread executes the program until reaching the first pragma. After data partitioning and mapping, all threads execute the code region in parallel. Each thread performs the computations on the vertices in its own partition.

II. Programmer specified partitioning. The pragmas also support programmer specified partitioning. Figure 4.9 shows an example of parallel hash table lookup. Before entering the loop, the programmer uses the function `SetPartition` to manually assign each entry in the hash table to a partition. The pragma *homo* indicates that the loop will be carried out by all threads. Each thread performs the hash function for every work in the worklist but only does the insertion for the work that is hashed into its partition. If the hash table is not partitioned and the computations are randomly distributed among threads, speculation will be needed since two threads may insert nodes into the same part of the hash table. Partitioning eliminates the need for speculation.

4.3.2 Non-speculative Heterogeneous Parallelism

```
#pragma SpiceC parallel hetero partition ([DataType],[PartitionType])
#pragma SpiceC region [RegionName] distribute ([Partition])
#pragma SpiceC join ([Partition])
```

The first pragma is used to mark the parallel code region such as a loop that is to be executed in parallel and where partitioning is to be applied. The `hetero` clause indicates that heterogeneous parallelism is to be used and the `partition` clause indicates the type of partitioning to be used. The second pragma is used to check in which thread the subsequent computation should be performed. The `distribute` clause indicates that the pragma will try to assign the subsequent computation to another thread to which the partition belongs so that the subsequent computation can be performed in parallel. If the partition belongs to the current thread, the subsequent computation is skipped. Finally,

```
for(index=0; index < table.size(); index ++)
   table[index].SetPartition(index % totalthreads());
#pragma SpiceC parallel homo partition(NONE)
foreach w in worklist do {
   #pragma SpiceC region R1
   index = hash(w);
   #pragma SpiceC region R2 inpartition(table[index].GetPartition()) {
     if ( table[index].lookup(w) == NULL )
       table[index].insert(w);
   }
}
```

Figure 4.9: Parallel hash table lookup with partitioning.

the third pragma, `#pragma join`, is used for synchronization between threads. The current thread will sleep until the thread owning the given partition finishes its computation. If the current thread owns the given partition, the pragma does nothing. Next the use of above pragmas is illustrated through examples.

I. Parallel loop with tree data structures. Figure 4.10 shows the pseudocode for parallel binary search and insertion. For each work item in the worklist, the program always starts search from the root node to a leaf node. If the tree is not partitioned, the search procedure requires speculation since multiple searches may go through the same path. To realize partitioning-based parallelism for this program, the program needs to be modified a little as shown in highlight. The pragma in the main function partitions the tree using the SYMM_SUBTREE partitioning and maps the partitions to the processor cores. Since it is unknown in which subtree each work in the worklist will be located before performing the search, the loop is marked heterogeneous so it is only executed by the master thread. In each iteration, the master thread performs searches until it reaches a node that is not in its partition. Then the `distribute` pragma distributes (assigns) the continuation of the search to a parallel thread corresponding to the node's partition. By partitioning the tree in this manner, the searches that are located in the same subtree will be performed by the same thread. Therefore, SpiceC improves the performance of the program without requiring the use of speculation.

II. Recursive computation with tree data structures. Figure 4.11 shows a simple example of recursion parallelism. The recursive function is called for multiple time at runtime. To

```
void search(Node node, int tag) {
  #pragma SpiceC region R1 inpartition (node.getPartition()) {
    if ( node.tag == tag )
      output (node);
    else if ( node.tag > tag )
      search (node.left, tag);
    else
      search (node.right, tag);
  }
  #pragma SpiceC region R2 distribute (node.getPartition())
    search (node, tag);
}
...
// in main function
#pragma SpiceC parallel hetero partition(Node, SYMM_SUBTREE)
for(int i=0; i<worklist.size(); i++) {
  search(root, worklist[i]);
}
```

Figure 4.10: Pseudocode for parallel binary search with tree partitioning.

```
function update(node) {
  pre_process(node);
  foreach child in node.children
    #pragma SpiceC region R1 distribute(child.GetPartition())
      update(child);
  foreach child in node.children
    #pragma SpiceC region R2 inpartition(child.GetPartition())
      update(child);
  foreach child in node.children
    #pragma SpiceC join(child.GetPartition())
  post_process(node);
}
...
// in main function
#pragma SpiceC parallel hetero partition(Node, ASYMM_SUBTREE)
update(root);
...
```

Figure 4.11: Pseudocode for parallel recursion computation with tree partitioning.

improve the data reuse, users can partition the tree using the ASYMM_SUBTREE strategy. The tree is partitioned in the main function. The recursive function checks the partition of each child of the node. If a child belongs to another thread's partition, the update for that child is assigned to that thread. After assigning all the children that are not in the current thread's partition, the current thread continues to update the rest children. After the current thread completes its work, it waits for other threads to finish their work using `join` and then performs the `post_process` work. Since the same thread always updates the same child, the cache performance of the program is improved.

4.3.3 Speculative Parallelism

For parallel programs where each computation may touch data across multiple partitions simultaneously, SpiceC provides support for speculation via the following pragmas.

```
#pragma SpiceC repartition

#pragma SpiceC touchpostpone(partition)
```

The first pragma is used to incrementally assign partitions to newly created nodes. Although this pragma is illustrated in context of speculation, it can also be used in non-speculative situations as needed. The second pragma is used to assist in conditional speculation. When the `touchpostpone` clause is used in a loop, the conditional speculation mechanism is used for the loop where speculation is only used for computations that dynamically are determined to require access to multiple threads' partitions.

It postpones the execution of current computation if the given partition is not a local partition. The postponed computation is later executed using standard (unconditional) speculation mechanism. The following paragraphs illustrates the usage of above pragmas using an example.

Figure 4.12 shows the pseudocode of a parallel implementation of Delaunay Mesh Refinement [59]. The program is designed to refine bad triangles (i.e., triangles whose circumradius-to-shortest edge ratios are larger than some bound) in a mesh. It takes a triangular mesh as input. Each iteration of the parallel loop first searches the triangles around one bad triangle to form a cavity, then retriangulates the cavity, and finally updates the mesh. If the procedure generates a new bad triangle, it is placed in the *todolist*. The parallel loop requires speculation since two iterations may access the same triangles. After the parallel loop is completed, the program moves the todolist to the worklist and repeat the parallel loop. The program ends when there is no bad triangle in the mesh. Since the bad triangles can be processed in any order, there is no need to enforce any order in the parallel implementation.

The data accesses of the Delaunay Mesh Refinement algorithm exhibit spatial locality since if a triangle is accessed in a computation its neighbors are likely accessed in the same computation. Grouping nearby nodes into partitions can improve the program's performance. The partitioning-based parallelism is realized via the use of three pragmas. The first pragma declares the code region that will be executed by all threads, partitions the mesh (a graph where each node is a triangle) with the METIS algorithm, and maps the partitions to the threads. The partitioning is only done once in the program since the

```
read(mesh);
worklist.add(mesh.getBads());
while( !worklist.empty() ) {
  todolist.clear();
  #pragma SpiceC parallel homo partition(Triangle, METIS)
  for(int i=0;i<worklist.size();i++) {
    #pragma SpiceC region Refine inpartition(worklist[i].GetPartition()) {
      Cavity c = new Cavity(worklist[i]);
      c.build(mesh);
      c.retriangulate(mesh);
      mesh.update(c);
      #pragma commit atomicity_check repartition
    }
    #pragma SpiceC region Add inpartition(worklist[i].GetPartition())
      todolist.lockedAdd(c.newBads());
  }
  worklist = todolist;
}
```

Figure 4.12: Parallel Delaunay Mesh Refinement with graph partitioning.

list of unpartitioned nodes will be empty in the subsequent execution. The second pragma checks if the computation should be done in the current thread. If the computation should be done in another thread, the current thread moves on to the next iteration. The function `GetPartition` returns the bad triangle's partition that is computed by the first pragma. The third pragma incrementally updates the partitioning after new triangles are added into the mesh. At runtime, the master thread executes the program until reaching the first pragma. After data partitioning and mapping, all threads execute the parallel code region in parallel as indicated by the `homo` clause. Each thread only performs the computations that begin from its own partition and skips the rest computations. After retriangulating a cavity, a thread incrementally partitions the newly created triangles using the `repartition` pragma.

Conditional speculation. Although data partitioning can reduce the misspeculation rate, the high overhead of speculation can still hurt the parallel performance. In fact, under partitioning-based parallelism, most computations are likely to touch their local partitions as long as each partition is large enough. When all threads are performing computations that only touch their own partitions, the speculation is not needed. Motivated by this observation, SpiceC provides support for partitioning-based *conditional speculation*. In this approach, by default a thread does not start speculation at the beginning of each computation. If a computation only touches data in its local partition, then it succeeds without speculation. If the thread detects that a computation will touch other partitions, the thread postpones its execution for later. When all threads complete the computations on their local partitions, they use speculation in redoing all postponed computations. The conditional use of speculation cuts down the percentage of computations executed under the speculation though it causes extra overhead for the computations involving multiple partitions. Therefore, as long as most computations only touch local partitions, the overall cost of speculation is reduced.

As previously mentioned, for the computations that require a subset of nodes in the graph, usually start with collecting the nearby nodes around the starting node. Since the collection procedure usually does not write to any global variable such as the graph data structure, it can be executed without speculation. In the collection procedure, a computation decides which nodes it will use later. Therefore, during the collection procedure, SpiceC can determine which partitions a computation will touch. If a computation will touch a remote partition, the thread postpones the computation, saving the loop index in a queue for later, and continues to the next computation. These saved computations are executed using speculation after all other computations are done.

For example, in Figure 4.12, the triangles that will be touched are determined in function `build`. Figure 4.13 shows the pseudocode of the `build` function. The function collects the triangles that are in or around the cavity. The function only writes to the variables local to the threads. Therefore, the function can be executed without speculation. One pragma (highlighted) is added into the function to realize adaptive speculation. The pragma first checks the partition of variable `neighbor`. If `neighbor` is in the local partition, the computation continues. If it is in a remote partition, the pragma saves the loop index of the computation (i.e., variable `i` on line 6 in Figure 4.12) in a queue and skips to the next computation. In the parallel loop shown in Figure 4.12, developers just need to wrap the

```
frontier.add(center);
foreach node in frontier do {
  foreach neighbor in node.getNeighbors() do {
    #pragma SpiceC touchpostpone(neighbor.GetPartition())
    if ( neighbor is part of the cavity ) {
      cavity.add(neighbor);
      frontier.add(neighbor);
    } else {
      border.add(neighbor);
} } }
```

Figure 4.13: Pseudocode for building cavity in Delaunay Mesh Refinement.

speculative code using `#pragma conditional speculate` instead of `#pragma speculate`. No other change needs to be made. At runtime, in the main parallel loop of Delaunay Mesh Refinement (as shown in Figure 4.12), the `#pragma conditional speculate` does nothing by default. Each thread checks the partition of each triangle at the pragma `touchpostpone`. After all threads complete the parallel loop, they redo the computations for the saved indices. During this redo procedure, the `#pragma conditional speculate` starts and ends speculative execution normally.

For the correctness purpose, write to any global variable is not allowed before the collection procedure is finished. The SpiceC compiler can detect whether there is a write to global variable along the paths from “#pragma speculate” to “#pragma touchpostpone”. If there is such a write, the compiler will give a warning to the programmer.

4.4 Evaluation

This section evaluates the SpiceC support for dynamic linked data structures. The experiments were conducted on an 8-core DELL PowerEdge T605 machine. Table 4.2 lists the details of the machine. The machine ran *CentOS v5.5*.

Processors	2×4-core AMD Opteron processors (2.0GHz)
L1 cache	Private, 64KB for each core
L2 cache	Private, 512KB for each core
L3 cache	Shared among 4 cores, 2048KB
Memory	8GB RAM

Table 4.2: Machine details.

4.4.1 Implementation

I. Pragmas. The `inpartition` and `touchpostpone` pragmas can cause a thread to skip the rest of the current computation and jump to the next computation. They are allowed to be placed inside a function called in the parallel region. They are implemented using the exception support in C++. When these pragmas decide to skip the current computation, they throw an exception. The parallel code region is wrapped with an exception handler. Once it captures an exception from those pragmas, it jumps to the next computation.

II. Threads-to-cores mapping. In a parallel region using the *speculation* pragma, each computation may access multiple adjacent partitions. In modern multicore processors, the caches are organized in a hierarchy that causes closer cores to share more efficiently. To improve the utilization of the cache hierarchy, it is necessary to place adjacent partitions to close cores so that accessing adjacent partitions will be faster. The cache topology aware mapping algorithm proposed in [62] is used to map the threads to the processor cores for parallel region using speculation.

4.4.2 Benchmarks

Ten benchmarks are used in the experiments. Two out of ten benchmarks are real applications. The rest eight benchmarks are from three benchmark suites – Lonestar [73], Olden [25], and Shootout. All benchmarks either already use pointer-linked data structures or were ported to make use of them. For all benchmarks, the order in which the computations are performed does not affect the correctness of the output though different orders may yield different outputs. Table 4.3 shows the details of the benchmarks.

The benchmarks `Delaunay Refinement` and `Agglomerative Clustering` are from the Lonestar benchmark suite [73] and originally written in JAVA. They are ported into C++ in the experiments. The original `Delaunay Refinement` uses a few hash table-based containers to store the graph. Therefore, even if two computations require different parts of the graph, they may share some data in the containers. The program was rewritten using pointer-linked data structures to solve this problem. `Agglomerative Clustering` is a hierarchical clustering algorithm. It uses a KD-tree for nearest neighbor search. The code was reorganized so that a node collection procedure is conducted at an early stage of each computation. `Barnes-Hut` is an implementation of the Barnes-Hut n-body algorithm [13] that simulates the gravitational forces in a galactic cluster. It uses an octree for storing nodes in the graph. The force-computation step was parallelized in the main loop. The benchmarks `Voronoi` and `TreeAdd` are from the Olden benchmark suite [25]. `Voronoi` implements a recursive Delaunay Refinement algorithm. `TreeAdd` calculates the sum of values in a balanced B-tree. Both benchmarks are parallelized using recursion parallelism. In the experiments, the sum calculation was repeated for ten times during the execution of `TreeAdd`. `Boykov-Kolmogorov` is a maxflow algorithm used for image segmentation. Two different loops were parallelized in this program using different partitionings – METIS and HASH, respectively. For HASH-based parallelism, although there is no data contention between threads on the graph structure, two threads may share other global variables. Locks are used to serialize the accesses to these variables. `AVL` is a self-balancing binary tree algorithm designed for addressing the issue of deletions. During execution, bal-

Benchmark	Par. Type	LOC + #Pragmas	Data Structures	Partitioning	Spec.? - Cond.?
Delaunay-Refinement	homo.	108 + 5	graph	METIS	yes - yes
Boykov-Kolmogorov	homo. homo.	93 + 5 93 + 3	graph graph	METIS HASH	yes - yes no - no
Barnes-Hut	homo.	107 + 2	graph	METIS	no - no
Agglomerative-Clustering	homo.	61 + 4	tree	ASYMM	yes - yes
Voronoi	heter.	106 + 3	tree	ASYMM	no - no
TreeAdd	heter.	23 + 3	tree	ASYMM	no - no
AVL	heter.	87 + 4	tree	SYMM	yes - no
ITI	heter.	60 + 4	tree	SYMM	yes - no
Hash	homo.	31 + 2	hash table	customized	no - no
Coloring	homo.	32 + 5	graph	customized	yes - yes

Table 4.3: Benchmark details. From left to right: benchmark name; function where the parallel region is located and type of parallelism (homogeneous and heterogeneous); lines of code in the function and number of pragmas introduced; type of data structure used in the benchmark; partitioning strategy employed; whether speculation is used and whether conditional speculation is used.

ancing usually takes place locally. It was parallelized using SYMM_SUBTREE partitioning. `ITI` [131] is a real application that constructs decision tree automatically from labeled examples. The `batch_train` function was parallelized speculatively. The `Hash` benchmark is from the Shootout benchmark suite. The main loop performs a large number of hash table lookups and insertions. The hash table can be easily partitioned by programmers. `Coloring` is an implementation of the scalable graph coloring algorithm in [21]. The original program does not use graph partitioning or speculative parallelism. In the original program, threads communicate node information with each other. In the experiments, the program was parallelized using user-specified data partitioning and speculative parallelism.

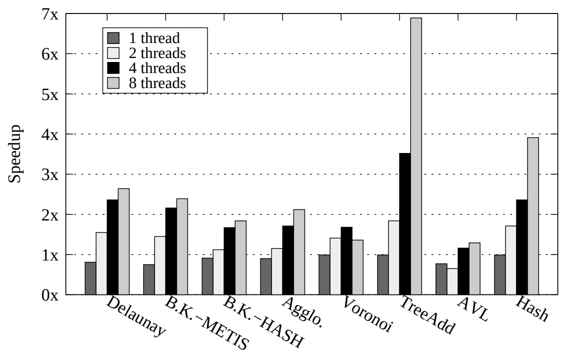


Figure 4.14: Speedup with partitioning.

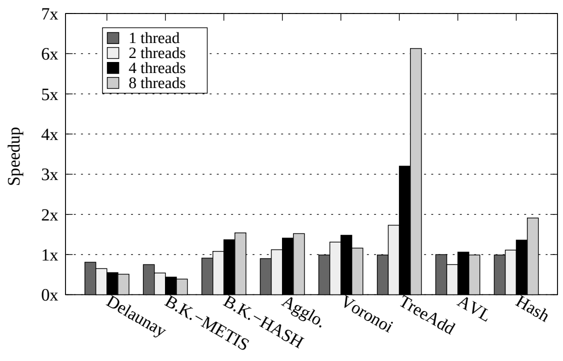


Figure 4.15: Speedup without partitioning.

4.4.3 Performance

Figure 4.14 shows the speedup achieved by SpiceC with data partitioning. Performance is shown for different number of threads. The original sequential versions of these benchmarks are used as baseline. For every benchmark except `Voronoi`, the performance of the benchmark improves with the increase of thread number. The performance of `Voronoi` decreases at 8 threads due to the high communication overhead (as shown in Figure 4.17). `AVL` and `ITI` gets slowdown with two threads since it employs a master/worker execution model similar to the example given in Figure 4.10. With two threads, they have only one worker thread to perform computations. Figure 4.15 shows the speedup without partitioning, where computations are assigned to threads in a round-robin manner. *As we can see, the speedups achieved with data partitioning are much higher than those without partitioning.* Without partitioning, parallelization causes slowdown for three benchmarks, for all of which speedup is achieved with data partitioning.

Table 4.4 compares the misspeculation rate with and without partitioning. *With*

Benchmark	w/o par.	w/ par.
DelaunayRefinement	90.42%	0.07%
Boykov-Kolmogorov	20%	0.09%
AgglomerativeClustering	3.21%	0.25%
AVL	21.84%	3.19%
ITI	16.58%	0.73%
Coloring	6.15%	0.17%

Table 4.4: Comparison of misspeculation rates with 8 threads. From left to right: benchmark name, misspeculation rate without partitioning, and misspeculation rate with partitioning.

partitioning, the misspeculation rates of all six benchmarks are greatly reduced. The misspeculation rates of five out of six benchmarks almost decrease to zero. `Delaunay Refinement` has the highest misspeculation rate when partitioning is not used. This is because close triangles are processed consecutively in the original program. With partitioning, most misspeculations of `Delaunay Refinement` are eliminated.

4.4.4 Effectiveness of Conditional Speculation

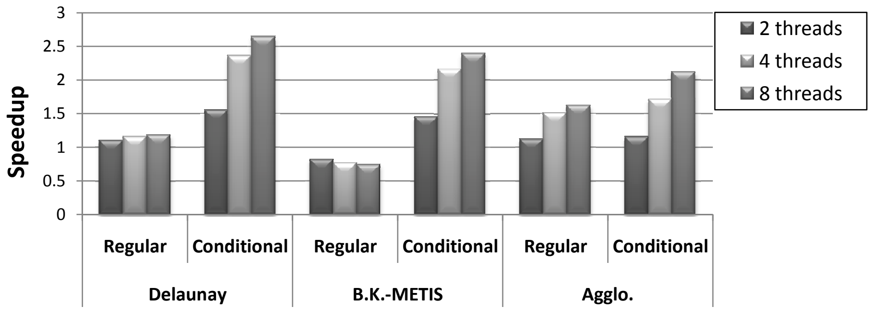


Figure 4.16: Regular speculation vs. Conditional speculation

Figure 4.16 shows a performance comparison between regular speculation and conditional speculation on the four benchmarks – `Delaunay Refinement`, `Boykov-Kolmogorov`, `Agglomerative Clustering`, and `Coloring`. Without conditional speculation, parallelization causes slowdown for `Boykov-Kolmogorov` due to its high speculation overhead (as shown in Figure 4.17). Conditional speculation improves performance for all four benchmarks. *With 8 threads, conditional speculation improves the performance for the four benchmarks by 2.23x, 3.91x, 1.31x, and 1.46x over regular speculation, respectively.*

Table 4.5 shows the percentages of postponed computations for the four benchmarks. *For all four benchmarks, only less than one-sixth of computations requires speculation since most computations only access local data partitions.* The postpone rates increase as the number of threads is increased. This is because the chance of accessing multiple threads' partitions goes up as the size of each partition decreases. `Delaunay Refinement` and `Boykov-Kolmogorov` have similar postpone rates. The postpone rates of `Agglomerative Clustering` and `Coloring` are a little lower. However, due to the high cost of the node collection procedures in `Agglomerative Clustering`, its speedup is lower than that of the other two.

Benchmark	2 threads	4 threads	8 threads
DelaunayRefinement	3.5%	6.9%	13.1%
Boykov-Kolmogorov	4.6%	7.6%	15.4%
AgglomerativeClustering	1.7%	3.2%	6.0%
Coloring	2.1%	3.9%	8.7%

Table 4.5: Postpone rate.

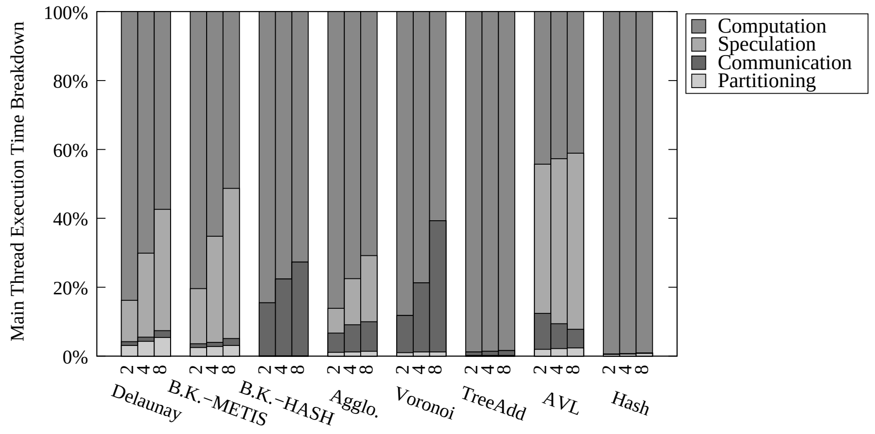


Figure 4.17: Time breakdown of the master thread.

4.4.5 Overhead

Figure 4.17 shows the time breakdown for the master thread with different numbers of threads. The time is divided into four categories: *computation, speculation, communication and partitioning*. Among the four categories, partitioning spends the least time. For the four benchmarks that use conditional speculation, their speculation overhead rises with the increase of thread number. This is mainly due to the increasing postpone rate as shown in Table 4.5. Among all the benchmarks, `AVL` has the highest overhead. Its master thread spends half of the time on speculation since it uses regular speculation. `B.K.-HASH` and `Voronoi` have higher communication overhead which hurts their performance. Three benchmarks, Barnes-Hut, `Hash`, and `TreeAdd`, have very low overhead. Partitioning eliminates the speculation cost for `Hash`.

4.5 Summary

This chapter has presented SpiceC support for dynamic linked data structures. Data partitioning and conditional speculation is introduced into SpiceC to improve cache locality and reduces misspeculation rate and speculation cost. The chapter also presented a set of SpiceC directives for efficiently parallelizing programs that rely on the use of dynamic linked data structures.

Chapter 5

Support for Parallelism in the Presence of I/O Operations

A majority of the parallel programming models (e.g., Threading Building Blocks [108], OpenMP [36], and Galois [72]) focus on exploiting data parallelism in loops including DOALL, DOACROSS, pipelined, and speculatively parallelized loops. However, these programming models only target loops that contain pure computations, i.e., they are free of I/O operations. Since many applications contain loops with I/O operations, they fail to yield much speedup due to these loops still being sequential. Therefore, it is highly desirable to support parallel programming models which allow parallel execution of *hybrid loops*, i.e., loops with both computation and I/O operations. For example, `Velvet` [143] is a popular bioinformatics application. In its version 1.1 (i.e., the latest version when this work is conducted), 18 pure computation loops have been parallelized using OpenMP. However, since OpenMP lacks support for parallelizing loops with I/O operations, none of the hybrid loops in `Velvet` have been parallelized. Actually, `Velvet` has 39 hybrid loops, out of which 26 loops can be parallelized by the approach presented in this chapter. These hybrid loops take a significant portion (27%–53%) of the total execution time in the experiments on a 24-core machine. Therefore, to further improve the performance of `Velvet`, developers must exploit the parallelism in hybrid loops. Interestingly, while this work is being conducted, the developers of `Velvet` were working independently on manually parallelizing the hybrid loops, which only confirms the above observation of the need to parallelize hybrid loops.

In Chapter 2 where hybrid loops are parallelized (as shown in Figure 2.4), DOACROSS parallelism is used (i.e., synchronization is imposed to deal with cross-iteration dependences) even if the computation part of the loop can be performed using DOALL (i.e., no cross-iteration dependence in the computation part). However, DOACROSS execution is not as efficient as DOALL. Because DOACROSS execution assigns only one iteration per scheduling step, it results in significant synchronization and scheduling overhead. In addition, the speculation-based loop parallelization techniques proposed in these works are not applicable when the loop contains I/O operations. While the work on parallel I/O [119, 124, 125] provides a set of low-level techniques that change the I/O subsystem to improve the performance of multiple simultaneous I/O operations, it does not provide any means for programmers to parallelize hybrid loops.

This chapter presents programming and compiler support for efficiently parallelizing *hybrid loops*, i.e., loops that contain I/O operations in addition to computation. The SpiceC programming model is extended by providing I/O related pragmas that are merely inserted preceding the hybrid loops being parallelized. The compiler techniques break the cross-iteration dependences involving I/O operations. Therefore, DOALL parallelism can be employed whenever there is no cross-iteration dependence in the computation part of the loop. Speculative execution of I/O operations is also supported to allow for speculative parallelization of hybrid loops. Finally, helper threading is employed to reduce the I/O bus contention resulting from aggressive parallelization of hybrid loops. The programmer can use helper threading by simply calling helper threading APIs preceding the loops.

5.1 Parallelizing Hybrid Loops

This section begins by discussing the challenges in parallelizing hybrid loops and then describes the approach to overcoming these challenges. First, it discusses why existing techniques for DOALL parallelization and speculative loop parallelization cannot be directly applied if a loop also contains I/O operations. The goal is to generalize these techniques so that they can be applied to hybrid loops. Second, this section shows that parallelization of hybrid loops can lead to I/O contention which must be effectively handled to realize the full benefits of parallelization.

I. Enabling DOALL Parallelization of Hybrid Loops. Figure 5.1(a) shows a typical loop with I/O operations that can be found in many applications (e.g., bzip2, parser [55], and stream encoder/decoder). Each loop iteration first checks if the end of the input file has been reached. If not, data is read from the file, computation on the data is performed, and finally results are written to the output file. When the computation within a loop does not involve cross-iteration dependences, maximum parallelism can be exploited via DOALL paralleliza-

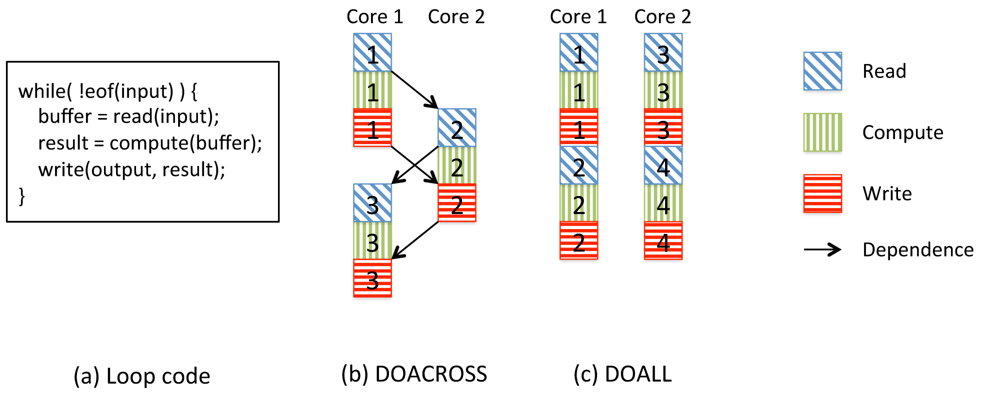


Figure 5.1: Execution of the loop example.

tion where all loop iterations can be executed in parallel. However, in a hybrid loop, even when the computation does not involve cross-iteration dependences, DOALL parallelization is not possible because the file read/write operations introduce cross-iteration dependences due to the movement (in the example, advancement) of the file pointer. In Chapter 2, such loops (e.g., in Figure 2.4) are parallelized using DOACROSS parallelism which incurs high scheduling and synchronization overhead—see Figure 5.1(b) for DOACROSS loop execution.

To fully exploit the parallelism in the loop, it is necessary to find a way to break the cross-iteration dependences due to I/O operations. In this chapter, SpiceC is extended to enable DOALL parallelization, which leads to the execution shown in Figure 5.1(c). DOALL parallelization eliminates the synchronization between consecutive iterations and enables more efficient scheduling polices such as Guided Self Scheduling [100]. Figure 5.2 compares the performance of DOACROSS with the performance of DOALL on a microbenchmark constructed based on the example loop type. In the microbenchmark, the *compute* function is composed of a loop. The ratio of I/O workload vs. computation workload can be adjusted by varying the loop size. The figure shows the performance comparison for both I/O-dominant workload and computation-dominant workload. In the I/O-dominant workload, the I/O calls take around 75% of the execution time of the sequential loop while in the computation-dominant workload the I/O calls take only 25% of the loop execution time. In both cases, DOALL performs better than DOACROSS, especially with large number of parallel threads. *Therefore, the first challenge addressed in this work is to efficiently parallelize hybrid loops by enabling DOALL parallelization.*

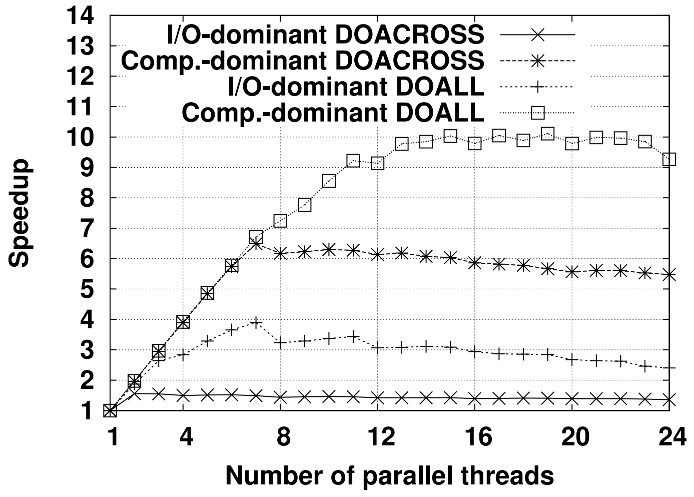


Figure 5.2: Performance comparison of DOACROSS and DOALL on the example loop.

II. Performing Speculative Parallelization of Hybrid Loops. Now let us consider the situation in which cross-iteration dependences exist in the computation part of the loop. It is impossible to perform DOALL parallelization of such loops even if the dependences introduced by I/O operations are broken. Recent works have shown that an effective approach to handling

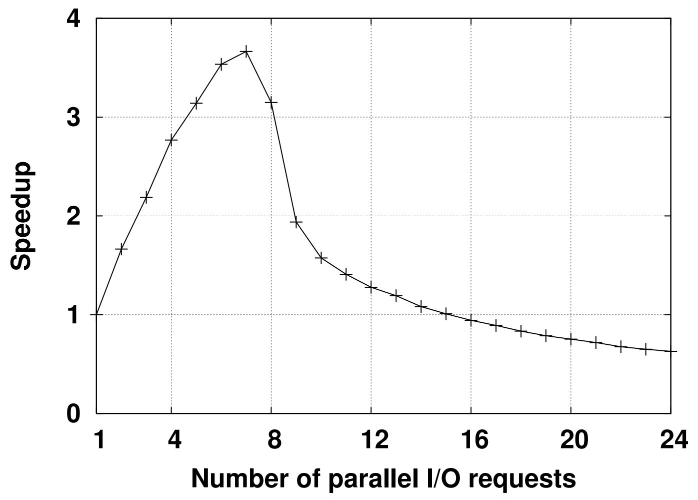


Figure 5.3: Performance of reading a 1 GB file using different numbers of parallel I/O requests.

cross-iteration dependences in the computation part is to employ speculative parallelization of loops. This approach is very effective when the cross-iteration dependences manifest themselves infrequently. Previous works [126, 43] have shown that speculative parallelization works better than non-speculative DOACROSS parallelization; however, these works also assume that the loops do not contain I/O operations. To apply speculative parallelization to the loops with I/O operations, it is necessary to enable the speculative execution of the I/O operations. *Therefore, the second challenge of efficiently parallelizing hybrid loops is to develop solutions for speculative execution of I/O operations.*

III. I/O Contention due to Parallelization. Another challenge arises once DOALL and speculative parallelization of hybrid loops have been achieved. Increasing parallelism also raises I/O bus contention. Figure 5.3 shows the performance of a microbenchmark that reads a 1 GB file from the disk. The microbenchmark creates multiple parallel threads, each

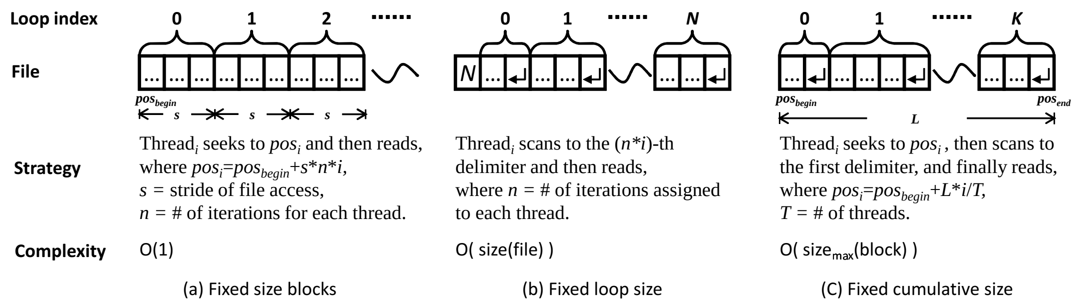

Loop index
File
Strategy
Complexity
0
1
2
pos_{begin}
s
s
s
Thread$_i$ seeks to pos_i and then reads, where $pos_i=pos_{begin}+s*n*i$, s = stride of file access, n = # of iterations for each thread.
O(1)
(a) Fixed size blocks
0
1
N
N
Thread$_i$ scans to the ($n*i$)-th delimiter and then reads, where n = # of iterations assigned to each thread.
O(size(file))
(b) Fixed loop size
0
1
K
pos_{begin}
pos_{end}
L
Thread$_i$ seeks to pos_i , then scans to the first delimiter, and finally reads, where $pos_i=pos_{begin}+L*i/T$, T = # of threads.
O($size_{max}$(block))
(C) Fixed cumulative size

Figure 5.4: Commonly used file access patterns of loops and strategies for locating the starting file position for each parallel thread.

of which then sends an I/O request to read a portion of the file. The number of parallel threads (i.e., number of parallel I/O requests) is varied to examine the impact of I/O bus contention. We can see that the I/O performance degrades quickly with more than 7 parallel I/O requests due to the I/O bus contention. Using more than 16 parallel I/O requests actually causes a slowdown compared to the performance with just one I/O request. Figure 5.2 also shows that the DOALL performance of the example loop degrades slightly with large number of parallel threads. *Therefore, to effectively parallelize hybrid loops, it is required to develop techniques for reducing I/O bus contention.*

5.1.1 DOALL Parallelization of Hybrid Loops

To apply DOALL parallelization to a hybrid loop, it is necessary to break the cross-iteration dependences introduced by the I/O operations in the loop. The dependence-breaking strategies for input operations differ from those for output operations. Thus, this section discusses them separately below.

I. Cross-iteration dependences caused by the input operations. These dependences arise because the starting file position for an iteration depends on the input operations performed in the previous iteration. The DOALL parallelization assigns each parallel thread a chunk of consecutive iterations for execution. Breaking the dependences requires identifying the starting file position corresponding to each parallel thread. Therefore, the starting file positions should be calculated before the execution of the loop. The method for computing the starting file positions depends upon the file access pattern used in the loop. An examination of `Velvet` [143] and the programs in two benchmark suites, SPEC CPU 2000 and PARSEC [16], has identified three commonly-used file access patterns which are described next.

(i) FSB: Fixed Size Blocks. The first access pattern, shown in Figure 5.4(a), is called the fixed-size blocks pattern as in this pattern each loop iteration reads a fixed-size block of data. The stride of the file pointer is equal to the size of the data block read by each iteration. Since the stride of the file pointer is known before entering the loop, it is easy to calculate the size of data to be read by each parallel thread. Thus the starting file position of a parallel thread can be calculated by summing up the size of data to be read by previous parallel threads. Given the starting file position of each parallel thread, the time complexity of moving the file pointer to the starting file position via a *seek* operation is O(1).

(ii) FLS: Fixed Loop Size. In the second pattern, as shown in Figure 5.4(b), the loop first reads the total number of blocks from the file and then accesses one delimited block during each loop iteration. Since the blocks are of variable size, delimiters are used

to separate them (e.g., in a text file, the delimiter is '\n'). The number of blocks, n, to be read by each parallel thread can be calculated from the total number of blocks. Using a *scan* operation, the starting file position of parallel thread i can be located by skipping $n * i$ occurrences of the delimiter. The time complexity is O(N), where N is the file size. This strategy is slow because it requires a *scan* as opposed to a *seek*.

(iii) FCS: Fixed Cumulative Size. In the third pattern, shown in Figure 5.4(c), each loop iteration accesses one delimited block that is encountered until the total size of data read by the loop reaches a given number. Because the data blocks are of variable size, delimiters are used to separate them. Since the total size of data to be read by the loop and the total number of parallel threads (T) are known, the starting file position of parallel thread i can be located by first skipping the i/T fraction of the data and then looking for the first occurrence of the delimiter. In this case, each thread is actually assigned with equal amount of data instead of equal number of iterations, which is different from typical DOALL parallelization. This strategy requires two operations—a *seek* and a *scan*. This *scan* is faster than the scan performed in *FLS* since it only scans to the first occurrence of the delimiter. The time complexity of this strategy is O(L), where L is the maximum length of a data block. This time complexity is much lower than that of assigning equal number of iterations to every thread—O(N), where N is the file size.

II. Cross-iteration dependences caused by the output operations. Simply computing the starting file position cannot help break cross-iteration dependences caused by output operations. This is because the calculated file position does not exist until all previous output operations

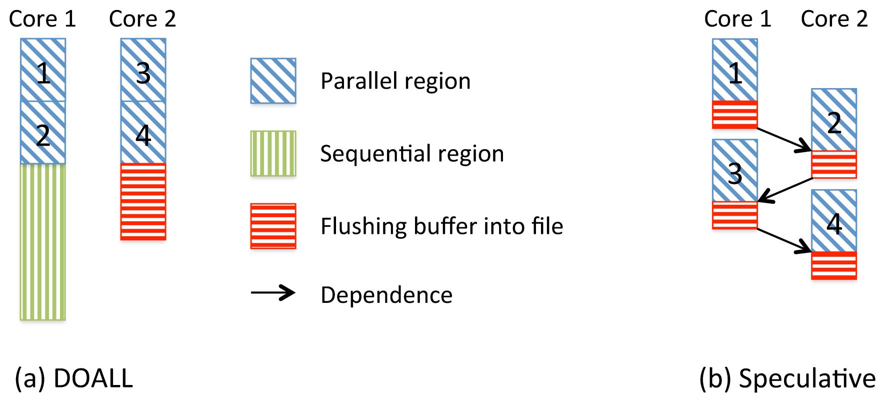


Figure 5.5: Strategies for flushing the output buffer.

have completed. Therefore, a different approach is proposed to break these dependences. An output buffer is created for each parallel thread. Each thread writes the outputs into its output buffer during the parallel execution of the loop. As a result, the output operations in one thread no longer depend on those in other threads. Flushing these buffers can be performed in parallel with the sequential code following the loop, as shown in Figure 5.5(a).

5.1.2 Speculative Parallelization of Hybrid Loops

Speculative parallelization is a way to efficiently exploit potential, but not guaranteed, parallelism in loops. Software speculative execution of non-I/O code has been studied in previous work [126, 43]. However, I/O operations cannot be executed speculatively in the same way because they use system calls. The code executed by the system calls is hidden from the compiler and runtime library, which thus cannot monitor the execution of system calls. Moreover, the results of I/O operations cannot be simply reversed once they are done. Therefore, efficiently parallelizing loops with I/O operations using speculative parallelism requires support for speculative execution of I/O operations.

I. Speculative execution of input operations. Speculative execution of the input operations in each iteration is enabled by creating a copy of the file pointer at the start of each iteration and then using the copy to perform all input operations in the iteration. If speculation succeeds, the original file pointer is discarded and the copy is used in the subsequent iterations. If speculation fails, the copy is simply discarded. One way of creating the copy is to instantiate a new file pointer and then *seek* to the current file position.

II. Speculative execution of output operations. Speculative execution of a loop iteration that contains output operations should satisfy the atomicity semantics, i.e., either all output operations occur or none occur. Therefore, the output operations in the loop iteration should not actually write to the file since that cannot be reversed. Speculative execution of the output operations in an iteration can be realized by using the output buffering described in the previous section. If speculation succeeds, the buffered output is kept. Otherwise, the buffered output is simply discarded. The output buffer is flushed at the end of each iteration as shown in Figure 5.5(b). Flushing the buffer at the end of each iteration requires small amount of memory since the buffer does not need to hold the outputs from the entire loop, but it incurs synchronization overhead because the flush operations need to be performed sequentially. The flush operation in an iteration needs to wait for the completion of the flush operation in the previous iteration as shown in Figure 5.5(b).

5.1.3 I/O Contention Reduction via Helper Threading

Parallelizing hybrid loops can lead to increased contention on the I/O bus. Helper threading is introduced to reduce this I/O contention. Before entering a hybrid loop, a helper thread is created which monitors the file buffers of the associated file pointers and performs the following tasks. For an input file pointer, the helper thread refills the file buffer when the size of remaining data in the buffer is less than a predefined threshold. For an output file pointer, it flushes the file buffer when the size of buffered output is larger than a predefined threshold. The use of a helper thread actually causes the I/O requests sent to the I/O bus to be serialized. Therefore, it eliminates the slowdown caused by the bursty nature of I/O requests. Moreover, since the parallel threads only access the buffer in the memory instead of the file on the disk, the I/O latency is also hidden by the helper thread. The helper threading can also be used in sequential loops to reduce I/O latency.

5.2 SpiceC Support for Parallelizing Hybrid Loops

The strategies described to enable parallelization of hybrid loops have been incorporated into the SpiceC programming system. The SpiceC programming model (as described in Chapter 2) has been extended to allow parallelization of hybrid loops. This section first presents the approach to programming parallel loops in the presence of I/O and illustrate it using several examples from real applications. Next it describes the implementation of the programming model.

5.2.1 Programming Parallel Hybrid Loops

Let us first consider the programming model for parallelizing hybrid loops. Code examples are presented to show how parallel hybrid loops are programmed and describe the use of I/O helper threads to boost the I/O performance on multicores. In the examples, the

C standard I/O library (`stdio`) is used for illustrating the programming model. All the proposed APIs can be used for other I/O libraries, e.g., the C++ I/O library (`iostream`). The SpiceC programming constructs used to express parallelism in this chapter are summarized in Table 2.1.

Parallelizing Loops with Input Operations

To enable DOALL parallelization of loops with input operations, the *pinput* clause is introduced:

```
#pragma SpiceC parallel doall pinput(file, stride, start, end)
```

The *pinput* clause is designed to be used with the SpiceC DOALL construct. To parallelize a loop with input operations, programmers just need to insert the DOALL construct combined with the *pinput* clause. The *pinput* clause has four parameters: *file*, *stride*, *start*, and *end*. Parameter *file* is the input file pointer that causes cross-iteration dependences. Parameter *stride* gives the stride of *file* in the loop; it can be either the size of data block read by each iteration or the delimiter between data blocks. They are distinguished from each other based on the parameter's type. Parameter *start* is the initial position of *file* at the beginning of the loop. By default (i.e., when this parameter is empty), the current position pointed by *file* is used as the initial position. Parameter *end* is the ending position that *file* will reach at the end of the loop. The *end* should be left empty if the ending position of *file* is unknown before entering the loop (e.g., the FLS file access pattern shown in Figure 5.4(b)). Given these parameters, the compiler can then calculate the starting file position for each parallel thread using the strategies described in Section 5.1.1.

Pattern	Pragma Example
FSB	pinput(file, size, 0, EOF)
FCS	pinput(file, delimiter, 0, EOF)
FSB	pinput(file, size)
FLS	pinput(file, delimiter)

Table 5.1: Examples of the *pinput* clause.

Table 5.1 shows four examples of the *pinput* clause covering four different file input patterns. In the first two examples, the loop reads the whole file. The initial file position is set to 0 so that the compiler knows that the file pointer starts from the beginning of the file. The use of *EOF* as the ending file position tells the compiler that the file pointer will reach the end of the file. In the last two examples, the loop reads a fixed number of data blocks from the current file position. The initial file position is not given in the *pinput* clause since the current file position is used as the initial position. The ending file position is left empty because it is unknown. The first and third examples exhibit the FSB pattern (as shown in Figure 5.4(a)) since they use a constant block size as the stride of the file pointer. In the second example, a delimiter is used as the stride and the total data size read by the loop can be calculated by the initial and ending file position. Therefore, it exhibits the FCS pattern (as shown in Figure 5.4(c)). The last example exhibits the FLS pattern (as shown in Figure 5.4(b)) since a delimiter is used as the stride and the loop size is fixed.

Figure 5.6 shows a real example of DOALL input loop that is similar to the fourth case given in Table 5.1. The original input loop is from the *DelaunayRefinement* benchmark [73]. Although the computation part of this benchmark has been parallelized in various ways [117, 72], its input loop, which contains I/O, has never been parallelized. The

```
file=fopen("input", "r");

fscanf(file, "%d", &ntuples);

#pragma SpiceC parallel doall pinput(file, "\n") {

  for( i=0; i<ntuples; i++) {

    fgets(line, maxsize, file);

    read_line(line, &index, &x, &y);

    tuples[index] = create_tuple(x,y,0);

  }

}
```

Figure 5.6: An input loop of benchmark *DelaunayRefinement*.

input loop reads an array of *ntuples* tuples from the file. Each iteration reads a line from the file and then creates a tuple structure from the input. In the example, the pragmas inserted to parallelize the loop are highlighted in bold. The SpiceC DOALL construct is used to identify the parallel region and type of parallelism. The *pinput* clause is used to specify the file input pattern. Since each iteration reads one line from the file, "\n" is given as the delimiter in the *pinput* clause.

Parallelizing Loops with Output Operations

To parallelize a loop with output operations using DOALL parallelism, it is necessary to buffer the output of each iteration. Programmers can achieve this by using the *boutput* clause with the SpiceC DOALL construct as follows.

```
#pragma SpiceC parallel doall boutput(file, isparallel)
```

The *boutput* clause tells the compiler that the output to *file* in the loop is written into its buffer. The buffer will not be flushed until the end of the loop, as shown in Figure 5.5(a). The *boutput* clause has two parameters: *file* and *isparallel*. Parameter *file* is the file pointer whose output needs to be buffered. Parameter *isparallel* specifies whether buffer flushing is performed in parallel with the computation threads or in a sequential fashion.

Figure 5.7 shows an output loop of `Velvet` [143], a widely-used bioinformatics application (genomic sequence assembler). Although the computationally-intensive part of `Velvet` has recently been parallelized with OpenMP, its hybrid loops have not been parallelized. In the original output loop, each iteration of the loop gets a nucleotide from the descriptor and outputs a character based on the type of the nucleotide. The pragmas

```
#pragma SpiceC parallel doall boutput(outfile, true) {
  for (index = 0; index<length; index++) {
    nucleotide = getNucleotide(descriptor, index);
    switch (nucleotide) {
      case ADENINE:
        fprintf(outfile, “A”);
        break;
      case CYTOSINE:
        fprintf(outfile, “C”);
        break;
      ...
    }
  }
}
```

Figure 5.7: An output loop of bioinformatics application `Velvet`.

used to parallelize the loop are highlighted in bold. The SpiceC pragma is used to mark the parallel region and the *boutput* clause is used to buffer all outputs of *fprintf* so that the output operations do not cause any cross-iteration dependence on the file pointer. Buffer flushing is programmed to be performed in parallel with the sequential code after the loop.

The *boutput* clause can also be used to program DOACROSS loops with output operations. For DOACROSS parallelism, the buffer is flushed at the end of each iteration, as shown in Figure 5.5(b). Figure 5.8 shows the kernel of the benchmark *Parser*. Each iteration first reads a line from *stdin* and then parses the line. Because of the cross-iteration dependences in the *parse* portion of the loop, the loop cannot be parallelized using DOALL parallelism. However, since these dependences rarely manifest themselves at runtime, the loop can be parallelized using speculative DOACROSS parallelism. Because the *parse* portion of the loop calls *printf* to output the results, programmers need the *boutput* clause when applying speculative execution to the *parse* portion. In the figure, the pragmas inserted to speculatively parallelize the loop are highlighted in bold. The first SpiceC pragma is used to identify the parallel region and type of parallelism. The loop is divided into two subregions by the rest of the SpiceC pragmas. The first subregion, *READ*, performs the input operations. The *pinput* clause is used at the beginning of the loop to tell the compiler the file input pattern. The compiler can then calculate the starting position of *infile* for each iteration and break the cross-iteration dependences introduced by *fgets*. The second subregion, *PARSE*, parses the input; it is executed speculatively as specified by the *commit* pragma at the end of the *PARSE* subregion. Since *printf* cannot be executed speculatively, the *boutput* clause is used with the DOACROSS construct to buffer the outputs to stdout

```
#pragma SpiceC parallel doacross \
  pinput(infile, “\n”, 0, EOF) boutput(stdout, false) {
  for(index=0; !feof(infile); index++) {
    #pragma SpiceC region READ {
      fgets(line, max_line, infile); }
    #pragma SpiceC region PARSE {
      if ( special_command(line) ) continue;
      first_prepare_to_parse(1);
      while ( !success ) {
        /* parser code here */
        printf(“ Linkage %d”, index+1);
        /* parser code here */
      }
      #pragma SpiceC commit atomicity_check \
        order_after(ITER-1, PARSE)
    }
  }
}
```

Figure 5.8: Speculative parallelization of a kernel from *Parser*.

in the loop. This enables speculative execution. Buffer flushing is performed at the end of each iteration in sequential order, as specified by parameter *isparallel* in the *boutput* clause.

Programming I/O Helper Threads

Helper threading is introduced to reduce the I/O contention caused by the hybrid loop parallelization. Table 5.2 summarizes the API for programming I/O helper threads. Function *inithelper* is used to create a new I/O helper thread; it returns the handle of the created helper thread which can then be bound to a file pointer using the function *sethelper*. Once bound to a file pointer, the helper thread continues to monitor the buffer corresponding to that file pointer.

API	Description
inithelper()	initialize a I/O helper thread
sethelper(file, helper)	bind a helper thread to a file pointer

Table 5.2: APIs for programming I/O helper thread.

Figure 5.9 shows an example of using I/O helper threading in a parallel loop taken from *DelaunayRefinement* as shown in Figure 5.6. A helper thread is created and bound to the input file pointer before entering the loop. Upon entering the parallel thread, the helper thread will automatically monitor the buffer corresponding to the file pointer copy in each parallel thread. Programmers do not need to code the binding of the helper thread to each copy of the file pointer.

I/O helper threading can also be used in sequential loops to reduce the I/O latency. Similar to the example of parallel loop, use of I/O helper thread in sequential loops is

```
file=fopen("input", "r");

fscanf(file, "%d", &ntuples);

helper = inithelper();

sethelper(file, helper);

#pragma SpiceC parallel doall pinput(file, "\n") {

  for( i=0; i<ntuples; i++) {

    fgets(line, maxsize, file);

    read_line(line, &index, &x, &y);

    tuples[index] = create_tuple(x,y,0);

  }

}
```

Figure 5.9: Example of using I/O helper threads.

straightforward: programmers just need to call *inithelper* and *sethelper* before entering the loop.

5.2.2 Implementation

Code translation for the new pragmas is incorporated into the SpiceC compiler. The compiler infrastructure, ROSE [104], is used to analyze the code annotated with the new pragmas and generate explicitly parallel C/C++ code. The SpiceC runtime library is extended to implement output buffering and helper threading. The following sections describe the code transformation performed by the SpiceC compiler and then elaborate how output buffering and helper threading are implemented.

Loop Transformation

API	Description
bwrite(index, data, size, file)	write *data* of *size* into the buffer of *file* in iteration *index*
bputs(index, string, file)	write *string* into the buffer of *file* in iteration *index*
bprintf(index, file, format, ...)	write formatted data into the buffer of *file* in iteration *index*
bflush(file, ALL/index, ispar)	flush the buffer of *file* using a separate thread or not

Table 5.3: Low-level functions for buffering outputs.

The code transformation from DOALL hybrid loops to C/C++ code is done automatically by the SpiceC compiler. Figure 5.10 shows an example of the code transformation. Figure 5.10(a) shows a DOALL hybrid loop parallelized by the extended SpiceC directives. Each iteration of the loop reads 100 bytes from the input file, then processes them, and

```
in = fopen("input", "r");
out = fopen("output", "w");
#pragma SpiceC parallel doall \
  pinput(in, 100, 0, EOF) boutput(out, true) {
  for( ; !feof(in); )
    fread(buf, 1, 100, in);
    process(buf);
    fputs(buf, out);
}
```

(a) Original code with SpiceC pragmas

```
init_parallel_threads();
...
in = fopen("input", "r");
out = fopen("output", "w");
args->in = in;
args->out = out;
start_doall(wrapper, args);
join_doall();
bflush(out, ALL, true);
...
close_parallel_threads();
```

(b) Transformed main program

```
void wrapper(void* args) {
  in = args->in;
  out = args->out;
  tid = get_thread_id();
  f = local_start(tid, in, 100, 0, EOF);
  c = local_count(tid, in, 100, 0, EOF);
  for( i=0; i<c; i++ )
    fread(buf, 1, 100, f);
    process(buf);
    bputs(i, buf, out);
}
```

(c) Transformed parallel loop

Figure 5.10: Example of code transformation.

finally outputs them into the output file. Figure 5.10(b) shows the transformed main program. The compiler inserts initialization of the parallel threads at the beginning of the program and close the parallel threads at the end of the program. The DOALL loop is outlined into function *wrapper*. All variables used in the DOALL loop are wrapped into a structure which is then passed as a parameter to the outlined loop. Function *start_doall* is called to execute the outlined loop in the parallel threads. Function *join_doall* is a synchronization method that waits until all parallel threads finish their work. Function *bflush* is called after the loop to flush the buffer of file pointer *out*. Figure 5.10(c) shows the outlined loop. Before the loop, three functions are called to prepare the workload for the current thread. Function *get_thread_id* is used to get the ID of the current thread. Function *local_start* is called to calculate the starting file position of the current thread. Function *local_count* is called to calculate the number of iterations to be performed in the current thread. Function *fputs* is replaced with function *bputs* for buffering the outputs. Table 5.3 lists the substitute for the C standard I/O functions. They are designed to buffer and flush the outputs for breaking the cross-iteration dependences introduced by the output operations.

Output Buffering

To enable output buffering (e.g., *bputs*, *bflush*), an output buffer is created for each parallel thread. The structure of each output buffer is a linked list, as shown in Figure 5.11. Each node in the linked list is a buffer of predefined length which can typically hold the output from several iterations. Once a node is full, a new node is created and appended to the linked list.

Flushing the output buffers takes $O(n)$ time, where n is the total size of all output buffers. Figure 5.11(a) shows the data layout of the output buffers for a DOALL loop with two parallel threads. The output buffer of thread 1 stores the output from iteration 1 to 6 and the output buffer of thread 2 stores the rest. It is straightforward to flush the output buffers with this data layout. Flushing the output buffers can start from the first thread and end with the last thread, which takes $O(n)$ time. Figure 5.11(b) shows the data layout of the output buffers for a DOACROSS loop. The output buffer of thread 1 stores the output from odd iterations and the output buffer of thread 1 stores the output from even iterations. In this case, the output buffers can be flushed in a round-robin manner, which also takes $O(n)$ time. The output of the current iteration in a DOACROSS loop can be flushed efficiently (as shown in Figure 5.8) since the output of the current iteration is always pointed to by the tail pointer of the output buffer.

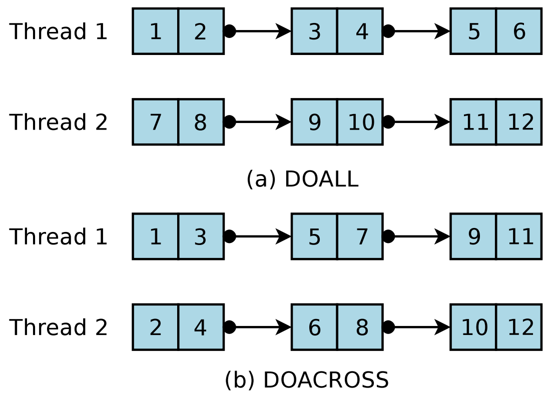


Figure 5.11: Buffer layout.

I/O Helper Threading

To enable I/O helper threading, the SpiceC runtime library is extended to implement an extended version of the standard file pointer by dividing the file buffer into two parts of equal size: the f-buffer and the h-buffer, where the f-buffer is used as the file buffer directly accessed by the I/O operations and the h-buffer is the helper buffer used by the helper thread.

For input file pointers, all input operations read data from the f-buffer. Once the f-buffer is empty, it is switched with the h-buffer. The helper thread keeps monitoring the h-buffer. If the h-buffer is empty, it refills it by calling I/O system calls. Output file pointers work in a similar way.

The SpiceC runtime typically does not put the helper thread to sleep to minimize the refilling latency. However, when the number of parallel threads is equal to the number of processor cores in the system, the helper thread will compete with the parallel threads for CPU resources. Therefore in this case, instead of busy idling the helper thread, the helper thread goes to sleep when it does not find any buffer that needs refilling. The helper thread is then woken when an f-buffer is switched with the h-buffer in a buffer pair.

5.3 Evaluation

This section evaluates the SpiceC extension for the parallelization of hybrid loops. The experiments were conducted on a 24-core DELL PowerEdge R905 machine. Table 5.4 lists the machine details.

Processors	4×6-core 64-bit AMD Opteron 8431 Processor (2.4GHz)
L1 cache	Private, 128KB for each core
L2 cache	Private, 512KB for each core
L3 cache	Shared among 6 cores, 6144KB
Memory	32GB RAM
OS	Ubuntu server, Linux kernel version 2.6.32

Table 5.4: Dell PowerEdge R905 machine details.

Name	Loops	Input	Output?	Spec.?	Helper?	% runtime	# stmts
velveth	8	FCS	Yes	No	Yes	53%	18
velvetg	18	FCS	Yes	No	Yes	27%	46
spacetyrant	4	–	Yes	No	No	95%	8
Delaunay	3	FLS	No	No	Yes	23%	7
bzip2	1	FSB	Yes	No	Yes	99%	13
parser	1	FCS	Yes	Yes	No	99%	8
blackscholes	1	FLS	No	No	Yes	45%	6
fluidanimate	3	FLS	Yes	No	Yes	36%	10

Table 5.5: Benchmark summary. From left to right: benchmark name, number of parallelized hybrid loops, input file access pattern, whether output buffering is used, whether speculative parallelization is used, whether helper threading is used, percentage of total execution time taken by the hybrid loops, number of statements added or modified for parallelization.

5.3.1 Benchmarks

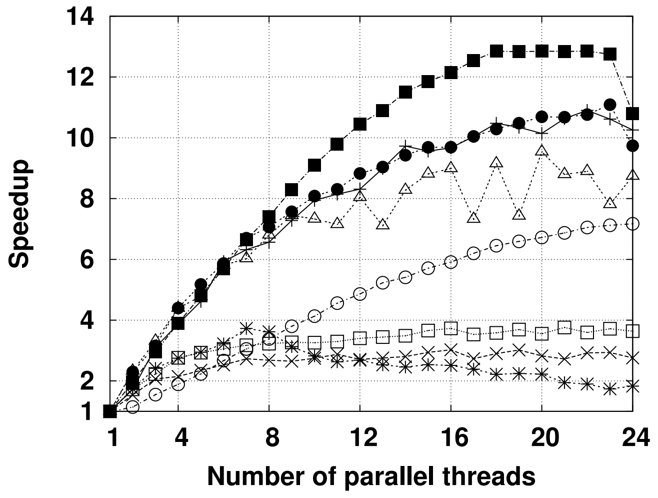
The new SpiceC programming constructs was applied to eight applications. Two are real-world applications, while the others are from the PARSEC [16] and SPEC CPU2000 suites. These applications were selected using the following criteria: (1) the applications must have at least one hybrid loop that can be efficiently parallelized (i.e., there is no frequent cross-iteration dependence in the computation part); and (2) the hybrid loop(s) must take a significant portion of execution time. Applying the SpiceC extension to applications that do not satisfy these criteria would diminish the capacity to measure and evaluate the hybrid loop parallelization techniques. DOALL parallelism was applied to the hybrid loops in seven applications except `parser` (speculative parallelism was required to parallelize `parser`). Table 5.5 shows the details of the benchmarks.

`Velvet` [143] is a popular genomic sequence assembler. It contains two applications—`velveth` and `velvetg`. `Velveth` constructs the dataset and calculates what each input sequence represents. `Velvetg` manipulates the de Bruijn graph that is built on the dataset. 18

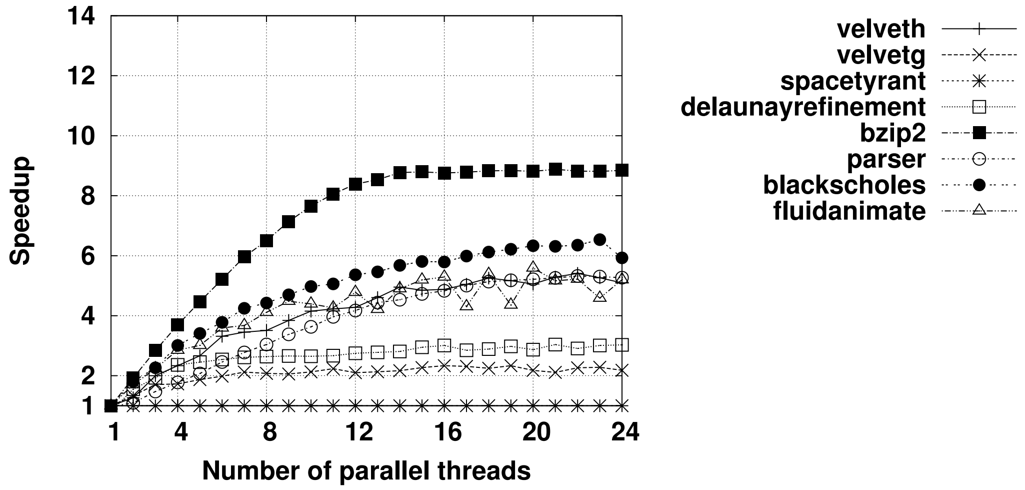
computation loops in `velveth` and `velvetg` have already been parallelized using OpenMP. In the experiment, their hybrid loops were parallelized. All parallelized input loops have the FCS pattern. Output buffering was used to parallelize the loops that contain output operations. Nucleotide sequence *SRR027005* [2] was used as input. `SpaceTyrant` [5] is an online multi-player game server. Its *backup* thread was parallelized which executes the *backupdata* function to backup game data. The *backupdata* function has 4 output loops for storing different types of data. Output buffering was used to parallelize these loops. In the experiments, every data block was assumed to be dirty and needed to be written to the file. `DelaunayRefinement` [73] is a meshing algorithm for two-dimensional quality mesh generation, originally written in JAVA. It was ported to C++. Its computation loop has been parallelized in previous work [72]. The three input loops in the *read* function were parallelized. They read different aspects of the input graph. All the loops have the FLS pattern. DOALL parallelism was applied to them by breaking the I/O dependences. `Bzip2` is a tool used for data compression and decompression. In the experiments, its compression loop was parallelized using DOALL. There are many superfluous cross-iteration dependences on global variables in `bzip2`. To remove these dependences, buffers were replicated for each iteration and many global variables were made local to each iteration. Some of the local variables are summarized into global variables after the loop. `Parser` is a syntactic parser for English. Speculative parallelism was used to parallelize its *batch_process* function which reads and parses the sentences in the input filed. The function contains a FCS loop. The I/O dependences were broken by calculating the starting file position for each iteration and using output buffering. It is required to speculate on dependences for control variables which may be altered by the special commands in the input file. `Blackscholes` is a computational finance application. The input loop in the *main* function was parallelized. The loop is a FLS loop that contains only input operations. `Fluidanimate` is designed to simulate an incompressible fluid in parallel. In its original *Parsec* version, the number of threads supplied by users must be a power of 2. The workload partitioning was modified to enable an arbitrary number of threads. For the PARSEC benchmarks, their pthread-based parallel versions were used in the experiments.

5.3.2 Performance

Figure 5.12 shows the absolute speedup of the parallelized applications over their sequential versions for varying number of parallel threads. Figure 5.12(a) shows the speedup when applying the hybrid loop parallelization techniques. The speedup ranges from 3.0x to 12.8x. On average, the performance of these applications is improved by 6.6x on the 24-core machine. For some benchmarks, the performance degrades with 24 parallel threads. This is caused by the contention between the helper thread and parallel threads. `Fluidanimate` has unstable performance across varying the number of parallel threads because its workload cannot be evenly partitioned with certain numbers of threads. The performance of `SpaceTyrant` goes down with larger number of threads. This is caused by the dynamic memory allocation for output buffering. For comparison, Figure 5.12(b) shows the speedup of these parallelized applications without hybrid loop parallelization. The speedup of `SpaceTyrant` is always 1 since its backup thread cannot be parallelized without hybrid loop parallelization. The speedup of these applications without hybrid loop paralleliza-



(a) With Hybrid Loop Parallelization



(b) Without Hybrid Loop Parallelization

Figure 5.12: Absolute parallelized application speedup over sequential programs.

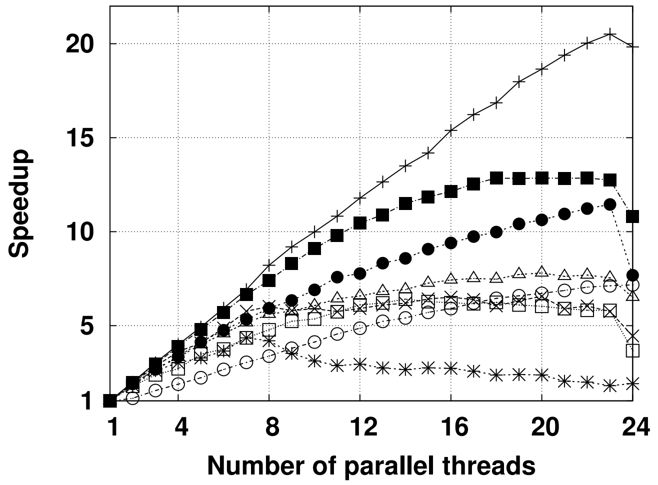
tion is between 2.3x–8.8x which is significantly lower than the speedups with hybrid loop parallelization.

Figure 5.13(a) shows the relative speedup of hybrid loops with parallelization vs. without parallelization. For seven applications, hybrid loop parallelization improves the hybrid loop performance by factors greater than 5x. On average, hybrid loop parallelization improves the loop performance by a factor of 7.54x. Figure 5.13(b) shows the relative parallelized full application speedups with hybrid loop parallelization vs. without hybrid loop parallelization. On average, hybrid loop parallelization improves the application performance by 68%.

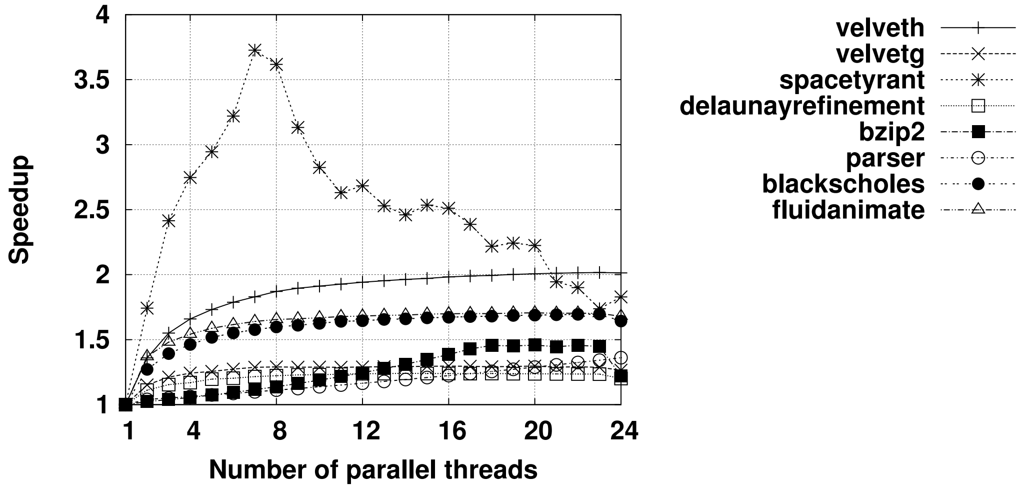
5.3.3 Impact of Helper Threading

Figure 5.14 shows the impact of I/O helper threading. On average, I/O helper threading improves the performance of parallelized hybrid loops by 11.9%. I/O helper threading usually provides more benefit with larger number of threads except 24 parallel threads where the I/O parallel thread competes with the parallel threads for processing resources. Figure 5.15 shows the impact of the buffer size of the helper thread in two applications—`velveth` and `DelaunayRefinement`. I/O helper threading achieves higher speedup with larger buffer sizes.

The buffer size is a critical factor that determines whether a helper thread can efficiently load data for multiple threads. The following example is used to show how the proper buffer size is set for a helper thread. Figure 5.16 compares the computation time with the data load time for different numbers of iterations in *DelaunayRefinement*. The



(a) Parallelized hybrid loops speedup



(b) Parallelized full application speedup

Figure 5.13: Relative speedup: with hybrid loop parallelization vs. without hybrid loop parallelization.

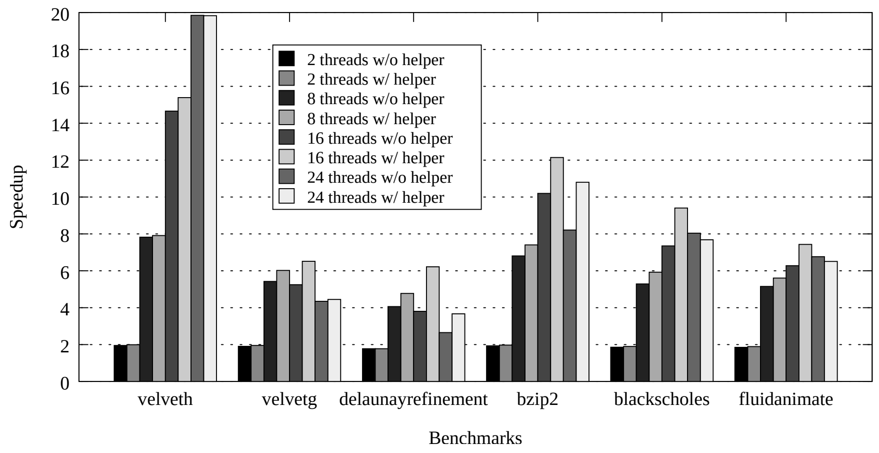


Figure 5.14: Impact of helper threading.

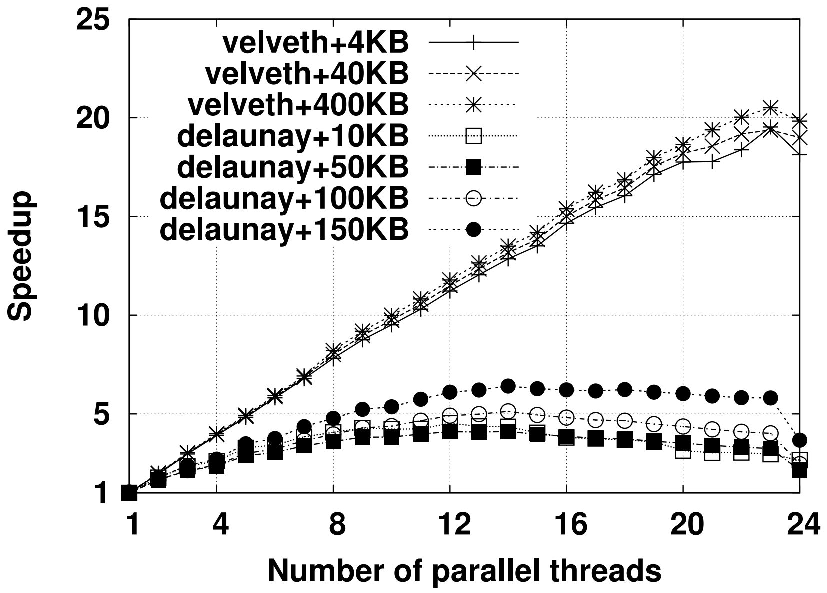


Figure 5.15: Speedup by varying buffer size.

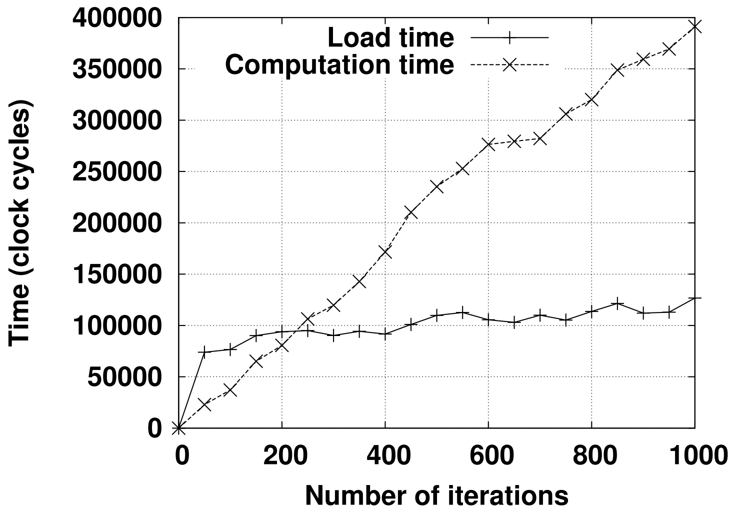


Figure 5.16: Computation time vs. data load time of benchmark *DelaunayRefinement*.

trend of the curves is similar for hybrid loops in other applications. Let us define tc_k as the computation time of k iterations and tl_k as the time of loading data for k iterations. From the figure, tc_k increases much more quickly than tl_k with the increase of k. For a helper thread that loads data for p parallel threads, its buffer for each parallel thread should be able to hold data for n iterations where $tc_n > p * tl_n$ since loading data should be finished before the data in the buffer is used up. Since tc_k increases much more quickly than tl_k, there always exists n satisfying $tc_n > p * tl_n$.

5.3.4 Overhead

Figure 5.17 shows the breakdown of the hybrid loop execution time for the parallelized applications. The time is divided into four categories: computation, I/O, speculation, and synchronization. For 5 out of 8 applications, most time is spent on the computation. `SpaceTyrant` and `Blackscholes` spend most time on I/O operations since their hybrid loops

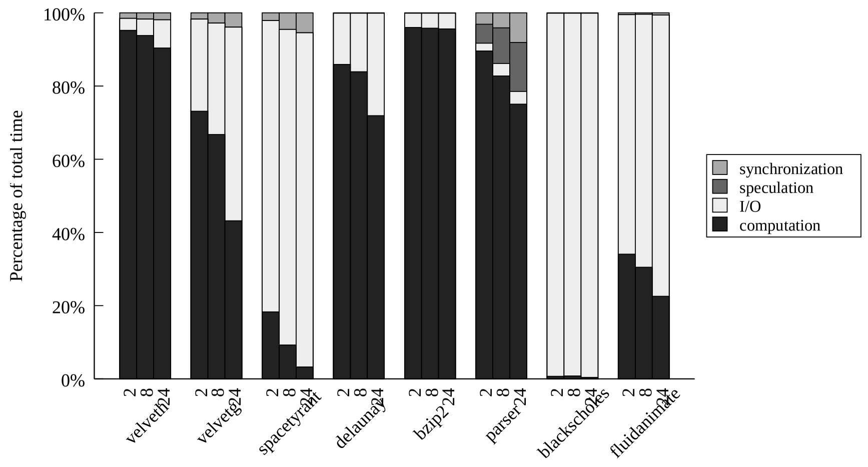


Figure 5.17: Breakdown of hybrid loop execution time.

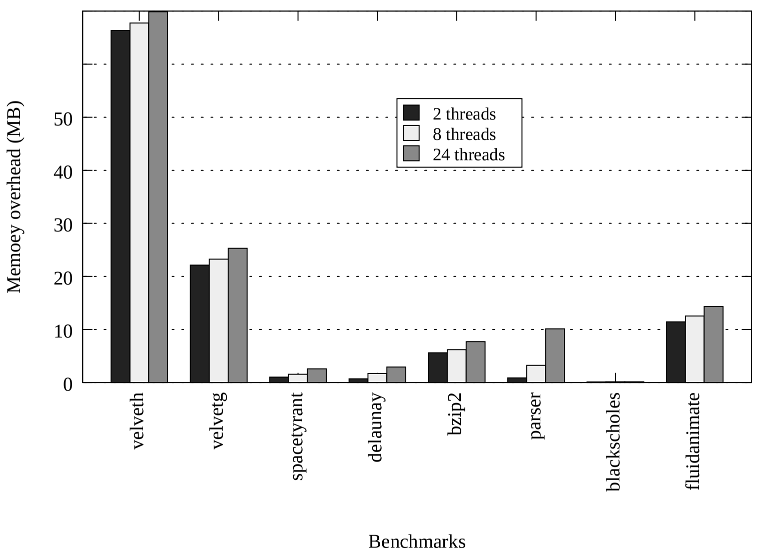


Figure 5.18: Memory overhead.

contain very little computation. Since most loops are parallelized using DOALL parallelism, very little synchronization overhead was introduced. `Parser` has the highest synchronization overhead since it is parallelized speculatively. I/O operations take a higher percentage of execution time with larger number of parallel threads due to I/O bus contention. Figure 5.18 shows the memory overhead incurred by hybrid loop parallelization. For 5 out of 8 benchmarks, the memory overhead is smaller than 10MB. `Velveth` and `velvetg` have high memory overhead since their output, which needs to be buffered in the memory during loop execution, is large.

5.4 Summary

In this chapter, the opportunity to parallelize hybrid loops, i.e., loops with computation and I/O operations, was identified. Several techniques were presented for efficiently parallelizing hybrid loops. The SpiceC programming model was extended for exploiting parallelism in hybrid loops. Parallelizing hybrid loops using the SpiceC extension requires few modifications to the code. The chapter also described the compiler and runtime support for the new SpiceC programming constructs.

Chapter 6

SpiceC on Heterogeneous Multicores with GPUs

GPUs provide an inexpensive, highly-parallel system to perform computations that are traditionally handled by CPUs. Increasingly, GPUs are being incorporated in multicore systems where they are used as accelerators. While the parallel application executes on the CPUs of the multicore system, kernals from the application that contain fine-grained data parallelism are executed on GPUs to further enhance performance. SpiceC can be easily extended to support GPUs since its copying and commit computational model (as described in Chapter 2) can naturally handle the host/device memory hierarchy in heterogeneous systems with GPUs. The host memory can be seen as the shared space while the device memory can be used as the private spaces. The data transfers between the shared and private spaces are then controlled by the SpiceC runtime. This chapter demonstrates how the SpiceC system described in preceding chapters is extended to take advantage of a GPU.

```
for (i=0; i<n; i++) {
   ... = A[P[i]];
   A[Q[i]] = ...;
}
```
(a) Irregular memory access

```
for (i=0; i<n; i++) {
   ... = A[i];
   if (A[i]) A[i+1] = ...;
}
```
(b) Irregular control flow

Figure 6.1: Examples of dynamic irregularities that cause cross-iteration dependences.

There is another aspect of this work that makes it novel. For the first time, it demonstrates how data parallel loops whose execution requires the use of speculation can be accelerated using a GPU. While recently a number of research works [76, 10, 14] have explored the use of GPUs, they exploit loop-level data parallelism in the absence of cross-iteration dependence. However, data parallel loops have been observed to contain dynamic irregularities that cause cross-iteration dependences at runtime, preventing existing techniques from parallelizing the loops on GPUs. In particular, two types of dynamic irregularities may cause cross-iteration dependences to arise at runtime as described below.

I. Dynamic irregular memory accesses refer to memory accesses whose memory access patterns are unknown at compile time. Figure 6.1(a) shows an example, where each iteration of the loop reads $A[P[i]]$ and writes to $A[Q[i]]$. The memory access patterns of $A[P[i]]$ and $A[Q[i]]$ are determined by the runtime values of the elements in arrays P and Q. It is possible that an element in array A is read in one iteration and written in another at runtime, which results in a dynamic cross-iteration dependence.

II. Irregular control flows are introduced by conditional statements, which may cause execution of paths that give rise to cross-iteration dependences at runtime, as illustrated in

Figure 6.1(b). In the example, there is a conditional branch that guards a write to $A[i+1]$, which is to be read in the next iteration. If the condition is true according to the runtime value of $A[i]$, a cross-iteration dependence occurs.

The remainder of this chapter presents a speculative execution framework for GPU computing. It is designed to parallelize loops that may contain cross-iteration dependences caused by the above dynamic irregularities. The SpiceC programming model is extended for writing speculative parallel loops for GPUs.

6.1 Execution Framework

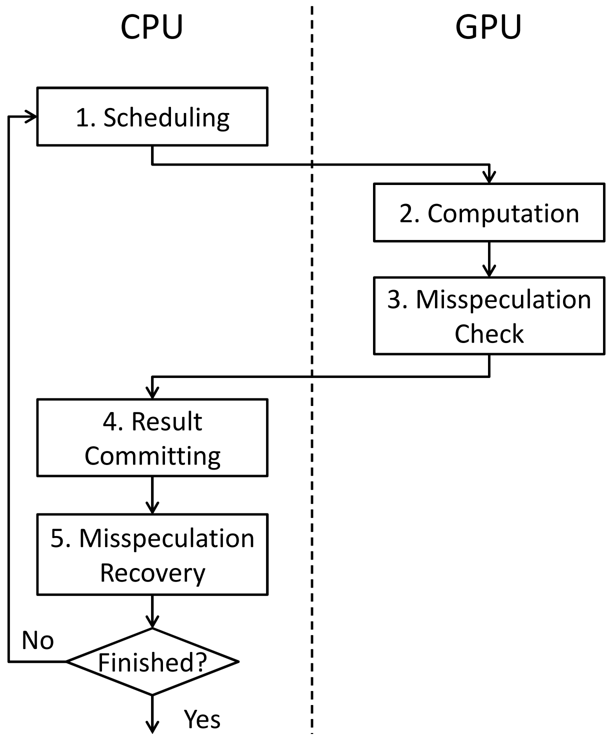


Figure 6.2: Execution framework of a speculative parallel loop with GPUs.

The overview of the developed framework for executing a speculatively parallelized loop using a GPU is given in Figure 6.2. The procedure consists of five phases: *scheduling*, *computation*, *misspeculation check*, *result committing*, and *misspeculation recovery*, among which *computation* and *misspeculation check* are performed on the GPU. The five phases are repeated until the entire loop is finished. This section briefly describes the five phases as follows.

I. Scheduling. Upon entering a speculatively parallelized loop, the CPU needs to determine the proper number of iterations that will be executed on the GPU in the next phase. Assigning large number of iterations to the GPU may cause excessive misspeculations while assigning small number of iterations may limit performance while leaving the GPU under-utilized.

Computation After scheduling, the GPU executes the iterations in parallel by speculating on the absence of cross-iteration dependence. To enable speculative execution, it is necessary to track the irregular memory accesses and control flows during the computation.

II. Misspeculation check. Misspeculation check consists of two steps: detection and localization. Misspeculation detection is used to determine whether the iterations have been executed correctly. If misspeculation is detected, the misspeculation localization step is used to identify the iterations that were executed incorrectly. In addition, for speculative execution on GPUs, it is necessary to identify the correct part of the results, which must be copied back to the CPU memory. To make misspeculation checks efficient, they are performed in parallel on the GPU. Since there is data parallelism in misspeculation checks, executing them on the GPU can lead to better performance.

III. Result committing. After misspeculation checks, the results should be copied from the GPU memory to the CPU memory. For better performance, it is necessary to optimize result committing to minimize copying overhead.

IV. Misspeculation recovery. The iterations where misspeculation occurs should be re-executed for correctness. It is better to re-execute on the CPU as few iterations as possible to minimize the recovery overhead. Executing more iterations on the GPU will get us better performance.

The following sections first illustrate the GPU part of the execution model, i.e., the second and third phases. Then the fourth and fifth phases, which are performed on the CPU, are elaborated. Finally, the scheduling policy is described.

6.2 Speculative Execution on GPUs

This section describes how to execute a loop speculatively in parallel on GPUs. The infrequent cross-iteration dependences in a speculative parallel loop are usually caused by two types of dynamic irregularities – irregular memory accesses and irregular control flows. The strategies for dealing with them are described separately.

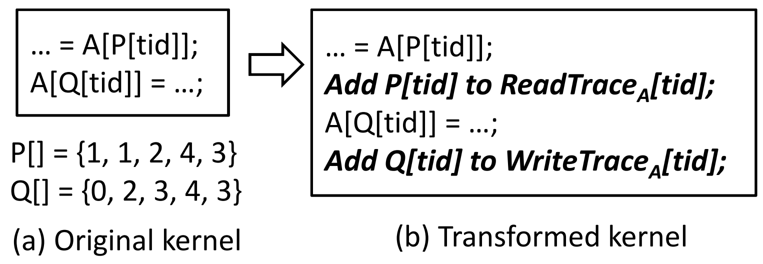


Figure 6.3: Code transformation of a loop with irregular memory accesses.

6.2.1 Irregular memory accesses

A loop with dynamic irregular memory accesses may have cross-iteration dependences that cannot be identified at compile time. Figure 6.3(a) shows the kernel of the loop example given in Figure 6.1(a). The conversion from loops to GPU kernels has been studied in [76, 10]. In the example, *tid* is the GPU thread ID. Each GPU thread executes one iteration of the loop. The memory access patterns of array A in the kernel is determined by the runtime values of the elements in arrays P and Q. Two iterations of the loop may read and write the same element of array A, causing cross-iteration dependence between them at runtime. An example of the runtime values of array P and Q is given below the kernel in Figure 6.3(a). Because iteration 1 (starting from 0) writes $A[2]$ and iteration 2 reads $A[2]$, there exists a RAW dependence between the two iterations. Similarly, since both iteration 2 and 4 write $A[3]$, there exists a WAW dependence between them. Because of these dependences, the results of iteration 2 and 4 will be incorrect if they are executed in parallel. The kernel can only be executed in parallel by speculating on the absence of cross-iteration dependence. To verify the correctness of the parallel execution of the kernel, it is necessary to identify the iterations whose results are incorrect due to occurrences of cross-iteration dependences at runtime. Our speculative execution on GPUs consists of three phases: execution with memory access tracking, misspeculation detection, and misspeculation localization. These phases are described next.

I. Memory access tracking. To detect the cross-iteration dependences, it is required to track which elements of the arrays with irregular access patterns are accessed in each iteration. This is done by inserting a tracking operation after each irregular memory access in the kernel. For low tracking overhead on the GPU, two static arrays of a predefined size are used for each iteration to store the indices of the read and written elements. The maximum number of elements that will be accessed in each iteration is assumed to be known. This is often true, including for all benchmarks used in our experiments. Figure 6.3(b) shows the transformed kernel with a tracking operation after each irregular access to array A.

II. Misspeculation detection. After the parallel execution of the kernel is finished, the existence of cross-iteration dependence should be checked. Unlike recent speculative parallelization techniques [126, 43] for CPUs, which only need to detect RAW dependences, speculative parallelization on GPUs requires detecting all kinds of dependences, including RAW, WAR, and WAW. Speculative parallelization on CPUs resolves WAR and WAW dependences by committing the results of the iterations in a sequential order. However, these techniques cannot be efficiently implemented on GPUs because they require complicated synchronizations. Since all computations are performed simultaneously on GPUs, it is necessary to detect all kinds of dependences.

```
foreach i in $ReadTrace_A$[tid] do
    if ( i not in $WriteTrace_A$[tid] )
        $Read_A$[i] = 1;
foreach i in $WriteTrace_A$[tid] do {
    $Write_A$[i] = 1;
    $WC_A$[tid] ++; }
Parallel sum reduction on
$Write_A$[] and $WC_A$[];
if ( tid < sizeof(A) &&
        $Read_A$[tid] $\wedge$ $Write_A$[tid] == 1)
        misspeculation  = true;
if ( tid == 0 &&
        $\sum_i Write_A[i] < \sum_i WC_A[i]$ )
        misspeculation  = true;
```

(a) Pseudocode for misspeculation detection

	Array Index				
$Read_A$	0	1	1	0	0
$Write_A$	1	0	1	1	1

$\sum_i Write_A[i]=4$
$\sum_i WC_A[i]= 5$

(b) Example of runtime values

Figure 6.4: Misspeculation detection.

The traditional shadow memory-based misspeculation detection method [107] is simplified and adapted for GPU computing. The misspeculation detection is performed on the GPU to efficiently explore the data parallelism in the detection procedure. The misspeculation detection method is lightweight. It can only detect the existence of cross-iteration dependences. If there is no such dependence, the lightweight method can decide that the results are all correct and it is safe to continue to the next computation. If there is any such dependence, the method cannot locate them. It is necessary to perform the misspeculation localization phase to determine in which iterations misspeculation has occurred.

Figure 6.4(a) shows the pseudocode for misspeculation detection for the kernel example given in Figure 6.3. The detection procedure is described in detail as follows.

1. $Read_A$ is calculated in parallel (line 1–3). $Read_A[i]$ is set when $A[i]$ is read but not written in an iteration. It records the elements of array A which are read-only in some iteration(s).

2. $Write_A$ and WC_A is then calculated in parallel (line 4–6). $Write_A[i]$ is set when $A[i]$ is written in an iteration. It records the elements of array A that have been written. WC_A stores the number of elements written in each iteration.

3. The sums of $Write_A$ and WC_A are calculated in parallel (line 7–8). The sum of $Write_A$ is the number of elements that have been written, where multiple writes to the same element in an iteration count as 1. The sum of WC_A is the number of writes to array A.

4. The intersection of $Read_A$ and $Write_A$ is calculated in parallel (line 9–11). If $\exists i, Read_A[i] \wedge Write_A[i] = 1$, then $A[i]$ is read-only in some iteration(s) and written in some other iteration(s). In this case, misspeculation occurs due to a RAW or WAR dependence. In some cases, an element may be read and written in an iteration(s) and also written in another iteration(s). This kind of dependences is treated as WAW dependences, which will be detected in the next step.

5. The sums of $Write_A$ and WC_A are compared. If $\sum_i Write_A[i] < \sum_i WC_A[i]$, then there must exist multiple iterations that write the same element. This indicates the existence of WAW dependences since some elements of array A are written in more than one iterations.

Figure 6.4(b) shows the calculated values of array $Read_A$ and $Write_A$ for the P

and Q given in Figure 6.3(a). The values indicate that there exist both RAW/WAR and WAW dependences in the kernel execution. $Read_A[2]$ and $Write_A[2]$ are both equal to 1 since iteration 1 writes $A[2]$ and iteration 2 reads $A[2]$. $\sum Write_A[i] = 4$ is smaller than $\sum WC_A[i] = 5$ since $A[3]$ is written in two iterations. The read and write of $A[3]$ happens in the same iteration. Thus, $A[3]$ is not recorded in $Read_A$. Therefore, there is no dependence detected on $A[3]$.

```
for (i=$BK_A$[tid]; i<$BK_A$[tid+1]; i++)
  $RW_A$[i]=ReadA[i] & WriteA[i];
if (tid < N)
  for(i=$BK_T$ [tid]; i<$BK_T$[tid+1]; i++)
     foreach j in $WriteTrace_A$[i] do
        $WW_A$[tid][j] ++;
Combine $WW_A$[1][] … $WW_A$[N][];
foreach i in $ReadTrace_A$[tid] do
  if ( $RW_A$[i] == 1)
     Misspec[tid] = 1;
foreach i in $WriteTrace_A$[tid] do
  if ($WW_A$[i] > 1)
     Misspec[tid] = 1;
if (Misspec[tid] == 1)
  foreach i in $WriteTrace_A$[tid] do
     $Wrong_A$[i] = 1;
Parallel reduction on $Wrong_A$[]
and Misspec[];
```

(a) Pseudocode for misspeculation localization

	Array Index				
RW_A	0	0	1	0	0
WW_A	1	0	1	2	1
$Wrong_A$	0	0	0	1	0
	Thread ID				
Misspec	0	0	1	0	1

$Wrong_{A\ reduced}[]=\{3\}$
$Misspec_{reduced}[]=\{2, 4\}$

(b) Example of runtime values

Figure 6.5: Misspeculation localization.

III. Misspeculation localization. Our misspeculation localization method identifies not only the misspeculated iterations but also the incorrect elements of arrays with irregular access patterns. With the information of incorrect elements, it is possible to optimize the copying of results from the GPU to the CPU. The localization procedure is parallelized on the GPU for better performance. Figure 6.5(a) shows the misspeculation localization for the kernel example. The details of the localization procedure are described below.

1. RW_A is calculated in parallel (line 1–2). RW_A is the intersection of $Read_A$ and $Write_A$, which indicates the elements that are read and written in different iterations. To calculate RW_A in parallel, array A is divided into blocks. Block boundaries are stored in BK_A. Each thread calculates RW_A for one block of array A.

2. WW_A is calculated in parallel (line 3–7). WW_A stores the number of iterations that write each element. $WW_A[i]$ is larger than 1 if $A[i]$ is written in multiple iterations. To calculate WW_A in parallel, $WriteTrace_A$ is divided into N blocks. Block boundaries are stored in BK_T. Each of the first N thread calculates WW_A for one block of $WriteTrace_A$.

3. RW_A is checked in each thread (line 8–10). If $RW_A[i]$ is set and $A[i]$ is read in the current thread, then the iteration performed by the current thread reads an element that is written in another iteration. The iteration is misspeculated due to a RAW/WAR dependence. Array $Misspec$ stores the misspeculated iterations.

4. WW_A is checked in each thread (line 11–13). If $WW_A[i]$ is larger than 1 and $A[i]$ is written in the current thread, then the iteration performed by the current thread writes an element that is written in some other iteration(s). The iteration is involved in misspeculation due to a WAW dependence. $Misspec[tid]$ is set when the iteration calculated by the current thread is involved in a misspeculation.

5. Next, $Wrong_A$ is calculated in parallel (line 14–16), which indicates the incorrect elements of array A. An element is incorrect only when it is written by at least one misspeculated iteration.

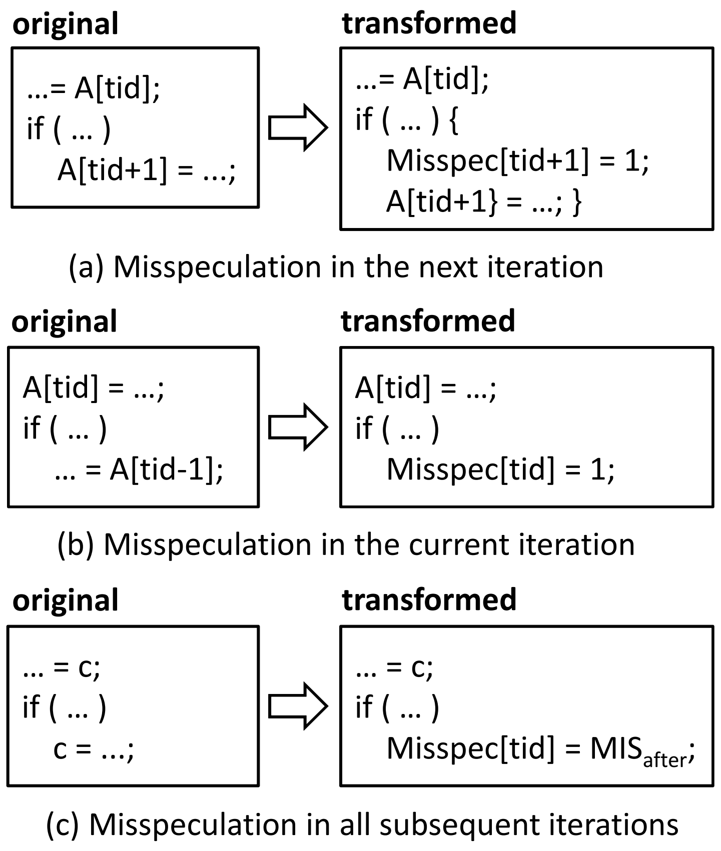
6. Finally, parallel reductions are performed on $Wrong_A$ and $Misspec$ to store the incorrect elements and misspeculated iterations in lists. The CPU needs to know the incorrect elements and misspeculated iterations to perform commit and recovery. Since both arrays, $Wrong_A$ and $Misspec$, are sparse, it is very inefficient for CPU to scan them to get the information. Storing them in lists reduces the commit and recovery overhead on CPU.

Figure 6.5(b) shows the values of RW_A, WW_A, $Wrong_A$, and $Misspec$ for the P and Q given in Figure 6.3(a). Since iteration 2 reads $A[2]$ which is written by iteration 1, iteration 2 is involved in misspeculation. Since iteration 4 writes $A[3]$ which is written by multiple iterations, it is misspeculated. As $A[3]$ is written by misspeculated iterations 2 and 4, its value is incorrect.

After locating the misspeculated iterations and incorrect array elements, the correct array elements are stored and recovery needs to be done for the misspeculated iterations. Section 6.3 presents details of copying and recovery.

6.2.2 Irregular Control Flows

Cross-iteration dependences may also be caused by irregular control flows in a loop. Figure 6.6 shows three types of cross-iteration dependences that are caused by irregular control flows. In Figure 6.6(a), the true branch condition causes a write to an element


original

```
...= A[tid];
if ( ... )
   A[tid+1] = ...;
```

transformed

```
...= A[tid];
if ( ... ) {
   Misspec[tid+1] = 1;
   A[tid+1} = ...; }
```

(a) Misspeculation in the next iteration

original

```
A[tid] = ...;
if ( ... )
   ... = A[tid-1];
```

transformed

```
A[tid] = ...;
if ( ... )
   Misspec[tid] = 1;
```

(b) Misspeculation in the current iteration

original

```
... = c;
if ( ... )
   c = ...;
```

transformed

```
... = c;
if ( ... )
   Misspec[tid] = MIS_after;
```

(c) Misspeculation in all subsequent iterations

Figure 6.6: Loops with irregular control flows.

that is read in the next iteration and thus makes the next iteration misspeculated. In Figure 6.6(b), the true branch condition reads an element that is written in the previous iteration and thus causes the current iteration to be misspeculated. In Figure 6.6(c), the true branch condition writes a scalar variable that is read in all iterations and therefore executing the branch makes them all wrong. These loops can be parallelized by speculating the branch will not be executed. To verify the correctness of the parallel execution, the execution of these branches needs to be monitored. Once these branches are executed, it is required to detect the misspeculation and identify the misspeculated iterations.

The cross-iteration dependences in the branches can be either marked by the programmer or detected by the static data race detection techniques, such as Locksmith [102]. Static data race detection techniques identify dependences in a conservative way. Therefore, they may cause false misspeculations. Programmers can better identify the branches using their knowledge of the application. The SpiceC programming model is extended for marking the branches that cause cross-iteration dependences (Section 6.5).

Once the cross-iteration dependences in the branches have been identified, the branches need to be transformed for speculative execution. The transformation consists of two steps.

1. In the branches that cause cross-iteration dependences, an operation is inserted for recording the misspeculated iterations. The same array $Misspec$ is used to store the misspeculated iterations as shown in the previous section.

2. The statements in a branch are removed from the GPU kernels if the branch execution will cause previous or current iterations to misspeculate.

The rationale behind this transformation is explained using examples. Figure 6.6 gives the transformed code for the branches. The example given in Figure 6.6(a) uses an operation that marks the next iteration as misspeculated. The statements in the branch are kept since they will not pollute previous iterations. The example shown in Figure 6.6(b) uses an operation that marks the current iteration as misspeculated. Since the current iteration is misspeculated, executing the statements in the branch is meaningless. Therefore, the statements are removed from the branch. In Figure 6.6(c), the operation inserted in the branch sets a special flag in $Misspec$. The flag indicates that all subsequent iterations including the current iteration are misspeculated. Since executing the statements in the branch also make previous iterations wrong, the statements are removed from the branch so that the results of previous iterations will be correct. In this branch, the current iteration is included in the misspeculated iterations because the statements in the branch need to be re-executed during recovery.

Having identified which iterations are misspeculated, the incorrect elements in the output array (i.e., incorrect results) should be identified next. Since the memory accesses are regular, polyhedral tools such as the Omega Library [64] can be used to capture the mapping between the iterations and array elements. Once the mapping is known, the elements that are written in the misspeculated iterations can be easily found. These elements are incorrect and should be stored in array $Wrong$ as shown in the previous section.

Similar to the previous section, the GPU is used to perform parallel reductions on $Wrong_A$ and $Misspec$ to store the incorrect elements and misspeculated iterations in lists. This reduces the commit and recovery overhead on the CPU.

Cross-iteration dependences caused by irregular control flows can also be detected by the method for irregular memory accesses presented in the previous section. However, in comparison to the misspeculation check for irregular memory accesses, the misspeculation check for irregular control flows is much cheaper. Therefore, the misspeculation check for irregular control flows should be used whenever possible.

6.3 Commit and Recovery

```
copyFromGPUToCPU(Misspec);
copyFromGPUToCPU(Wrong$_A$);
if ( sizeof(Wrong$_A$) == 0 )
  copyFromGPUToCPU(A);
else {    // copy only correct part of array A
  prepend(-1, Wrong$_A$);
  append(size(A), Wrong$_A$);
  for (i=0; i<size(Wrong$_A$); i++)
    copyFromGPUToCPU(A[Wrong$_A$[i]+1 ... Wrong$_A$[i+1]-1]);
}
for (i=0; i<size(Misspec); i++)
  reexecute(Misspec[i]);
```

Figure 6.7: The pseudocode of commit and misspeculation recovery for the example given in Figure 6.3.

After the speculative execution on the GPU is finished, the correct results need to be committed and misspeculation recovery should be performed. The commit and misspeculation recovery are performed on the CPU. Figure 6.7 shows the pseudocode of commit and misspeculation recovery for the example given in Figure 6.3. The procedure is described next in detail.

1. The reduced arrays *Misspec* and *Wrong* are first copied from the GPU to to CPU. These arrays are required for the commit and misspeculation recovery. This step has very low overhead since the arrays are usually very small.

2. Data are committed back to the CPU. For an array, if all elements are correct, the whole array is directly copied from the GPU to the CPU and the original array is overwritten. If misspeculation is detected, it is necessary to scan array *Wrong* and only copy the correct elements between the wrong elements stored in array *Wrong*.

3. Finally, the misspeculation recovery is performed. In this step, the misspeculated iterations are re-executed on the CPU. For loops with irregular memory accesses, array *Misspec* is scanned before redoing every iteration inside. For the loops with irregular control flows, recovery is performed depending on the misspeculation type. For the first two cases in Figure 6.6, where only one iteration is misspeculated with the execution of the branch, every misspeculated iteration in array *Misspec* is re-executed on the CPU. For the third case in Figure 6.6, where all subsequent iterations are misspeculated, only the first iteration (i.e., the one that executes the branch and causes the misspeculation) is re-executed on the CPU. All remaining misspeculated

iterations are assigned to the GPU in the next scheduling assignment. This improves the performance due to the reduced number of iterations executed sequentially on the CPU.

Array $Wrong$ can also be used to reduce the copy-in (copy from the CPU to GPU) overhead. For loops with irregular memory accesses, it is unknown which array elements that will be accessed in an assignment of iterations. Therefore, the whole array needs to be kept in the GPU memory. After the recovery procedure, all elements that are re-calculated on the CPU are stored in array $Wrong$. In the next assignment, only the elements stored in array $Wrong$ needs to be copied from the CPU to GPU. All other elements in the GPU memory are already up-to-date.

6.4 Scheduling

For speculative parallel loops, scheduling more iterations in one assignment may not always give better performance. This is because larger number of iterations in one assignment may cause excessive misspeculations degrading the performance. Therefore, when scheduling iterations in a speculative parallel loop, it is necessary to consider the misspeculation rate. The scheduling policies for different types of loops are described as follows.

I. Loops with irregular memory accesses. For speculatively parallelized loops with irregular memory accesses, scheduling more iterations in one assignment will increase the chance of dependences between iterations. It is not possible to determine the optimal assignment size since the cross-iteration dependences are unknown at compile time. A runtime scheme that dynamically determines the assignment size is proposed.

In the first assignment, n/m iterations are scheduled to the GPU, where m is the number of elements written in each iteration and n is the number of elements in the array. If more than n/m iterations are scheduled in one assignment, there must exist two iterations that writes the same element. From the second assignment, the assignment size is adjusted based on the misspeculation rate in the previous scheduling. If the misspeculation rate is higher than a predefined threshold, the assignment size is halved to reduce the misspeculation rate in the next round of scheduling. If the misspeculation rate stays zero for a number of consecutive iterations, the assignment size is doubled for better utilizing the large number of stream processors on the GPU. It is unwise to increase the assignment size beyond n/m since misspeculation will definitely happen when the assignment size is large than n/m.

II. Loops with irregular control flows. For the first two cases in Figure 6.6, where only one iteration is misspeculated by the execution of the branch, the runtime schedules as many iterations as possible in one assignment. This is because the number of misspeculated iterations are almost solely determined by the number of iterations that execute the branches. Therefore the misspeculation rate can only be changed if the iterations that execute the branches are scheduled as the first or last iteration in an assignment, which is very unlikely.

For the third example in Figure 6.6, where all subsequent iterations are marked misspeculated if the branch is executed, the runtime measures the average interval between two iterations that executes the branch at runtime and uses the interval as the assignment size when performing scheduling. This is because once an iteration executes the branch, all subsequent iterations are misspeculated. Therefore, to reduce the number of misspeculated iterations, the iterations that execute the branch is better to appear near the end of the assignments. Sometimes, there may be more than two iterations executing the branch in one assignment. Then the runtime can use the interval between them as the size for the next assignment so that the second iteration that executes the branch will appear at the end of the next scheduling.

6.5 Programming Speculative Parallel Loops on GPUs

This section shows how speculative parallel loops are programmed on GPUs. The techniques described to enable speculative parallelization on GPUs have been incorporated into SpiceC.

To enable loop execution on GPUs, the `target` clause is introduced in the SpiceC `parallel` construct.

```
#pragma SpiceC parallel doall target(device)
```

The intent of the `target` clause is to specify the device on which a given computation will be executed. The valid device specified by the `target` clause can be cuda, cell, and etc. This chapter uses `target(cuda)` for loop parallelization on GPUs. Only DOALL loops can be executed on GPUs since other forms of parallelism require synchronizations which are extremely expensive on GPUs.

6.5.1 Irregular Memory Accesses

To enable speculative parallelization of loops with irregular memory accesses, the `speculate` clause is introduced:

```
#pragma SpiceC parallel doall speculate(array)
```

A loop parallelized with the `speculate` clause will be executed speculatively under the previously described framework. Programmers can specify which arrays may cause cross-iteration dependences in the `speculate` clause. The memory accesses to these arrays will be monitored at runtime for misspeculation check. Although the compiler can identify the arrays that have irregular access patterns [140], not all of them will cause cross-iteration dependence at runtime. Programmers can better identify which arrays need to be monitored using their knowledge.

```
#pragma SpiceC parallel doall target(cuda) speculate(data)
for (mm=1; mm<=mtrn; mm++) {
  jrev2s = (mm-1)*mskip + jrev2*nskip;
  js = (mm-1)*mskip + (j+1)*nskip;
  h = data(jrev2s+1);
  data(jrev2s+1) = data(js+1);
  data(js+1) = H;
}
```

Figure 6.8: A speculatively parallel loop from benchmark `ocean`.

Figure 6.8 shows a loop example extracted from benchmark `ocean`, which is a Boussinesq fluid layer solver [35]. The loop accesses a vector with runtime determined strides and offsets. Each iteration of the loop first calculates the indices of two elements of array *data* based on the loop index *mm* and a few other variables. It then reads and writes the two elements to swap them. It is impossible to know which elements will be swapped in each iteration at compile time since their indices are determined by runtime values. There may exist cross-iteration dependences at runtime since two iterations could potentially access the same element. Although the cross-iteration dependences usually do not manifest themselves in real runs, it is possible that performing the iterations in parallel produces wrong results. Therefore, speculative parallelization is required for the loop. In the example, the SpiceC `parallel` construct with the `speculate` clause is added before the loop to speculatively parallelize it. Array *data* is specified to be monitored at runtime since its memory accesses may dynamically cause cross-iteration dependences. From the example, we can see that it is very easy to speculatively parallelize a loop with the `speculate` clause since programmers just need to insert the clause before the loop without modifying the loop.

6.5.2 Irregular Control Flows

To enable speculative parallelization of loops with irregular control flows, the `branch` construct is introduced:

```
#pragma SpiceC branch misspeculate(iterations)
```

The `branch` construct is designed to be inserted at the beginning of a branch that will cause cross-iteration dependences once its branch condition is true. The `misspeculate` clause is used to specify the misspeculated iterations if the branch is executed. A loop that contains the `branch` construct will be executed speculatively using the scheme described in Section 6.2.2.

The `iterations` expression in the `misspeculate` clause is designed to allow the following forms: absolute iterations, relative iterations, and iteration ranges. Absolute iterations can be expressed as `(i)`, where i is the iteration index. For example, `(10)` denotes the 10^{th} iteration. Relative iterations can be expressed as `(+i/-i)`, where i is the relative iteration index. For example, `(+1)` denotes the next iteration. Iteration ranges can be expressed as `(i:j)`, where i and j can be either absolute iteration index or relative iteration index. For example, `(+0:+4)` denotes the current iteration and next four iterations. Multiple iterations can be separated by comma in the expression. For example, `(-1,+1)` denotes the previous and next iterations.

Figure 6.9 shows a loop example extracted from benchmark `mdg`, which dynamically calculates water molecules in the liquid state at room temperature and pressure [77]. The example is simplified for better illustration. The example contains two nested loops. The outer loop is parallelized. In each iteration of the outer loop, the inner loop calculates array rs and counts how many elements are larger than a predefined threshold. After the inner loop, there is a branch that is executed only if there exists elements in array rs, which are smaller than the threshold. The execution of the branch will cause cross-iteration dependences since it reads and writes a shared variable vir. Therefore, parallelizing the loop requires speculation. In the example, the `for` construct is inserted before the loop to

```
#pragma SpiceC parallel doall target(cuda)
for (j = i + 1; j <= nmol; j++) {
  ...
  kc = 0;
  for (k = 1; k <= 9; k++) {
    rs[k - 1] = ...;
    if (rs[k - 1] > mdvar_1.cut2)
      kc++;
  }
  if (kc < 9) {
    #pragma SpiceC branch misspeculate(+1:nmol)
    ...
    vir = ...;
    ...
  }
}
```

Figure 6.9: A loop with irregular control flow from benchmark `mdg`.

parallelize it. The `branch` construct is inserted at the beginning of the branch for monitoring the control flow. The `iterations` is set to `(+1:nmol)` since the execution of the branch can make subsequent iterations misspeculated. From the example, we can see that programmers can use the `branch` construct to speculatively parallelize loops with irregular control flows without modifying the loop.

6.6 Evaluation

To generate CUDA code, the backend of the SpiceC compiler is extended using OpenMPC [76], which is an OpenMP-to-CUDA compiler. The programmers use pragmas to annotate the variables or control flows that may cause cross-iteration dependences. A runtime library is developed for scheduling, misspeculation check, result committing, and misspeculation recovery.

CPU	4-core Intel Xeon E5540 Processor 24GB RAM
GPU	nVidia Tesla C1060 240 processor cores (1.296GHz) 4GB SDRAM CUDA 3.0 installed
OS	Linux kernel version 2.6.18

Table 6.1: Machine details.

An nVidia Tesla C1060 was used as the experimental platform. The device includes a single chip with 240 cores organized as 30 streaming multiprocessors. The device is connected to a host system consisting of Intel(R) Xeon(R) E5540 processors. The machine has CUDA 3.0 installed. The machine details are summarized in Figure 6.1.

6.6.1 Benchmarks

The framework was evaluated using the benchmarks shown in Table 6.2. They were selected because they contain dynamic irregularities that may cause cross-iteration dependences at runtime and thus they cannot be parallelized without speculation. Three of the benchmarks have irregular memory accesses and four have irregular control flows. It is observed that only one or two pragmas are required to parallelize each loop. The details of the benchmarks are described as follows.

I. Benchmarks with irregular memory accesses. The `ocean` benchmark [35] solves the dynamical equations of a two-dimensional Boussinesq fluid layer for studying the chaotic behavior of free-slip Rayleigh-Bénard convection. The source code of the parallelized loop is given in Figure 6.8. Each loop iteration swaps two elements whose indices are calculated based on some runtime values. Since two iterations may access the same element, parallelization requires speculation. The memory accesses to a data array are tracked for misspeculation check. The `trfd` [136] benchmark simulates a two-electron integral transformation, which

Benchmark	Irregularities	% of time	# of pragmas
ocean	irregular memory accesses	45%	1
trfd	irregular memory accesses	6%	1
fftbench	irregular memory accesses	20%	1
mdg	irregular control flows	94%	2
strcat	irregular control flows	99%	2
gothic	irregular control flows	99%	2
alvinn	irregular control flows	97%	8

Table 6.2: From left to right: benchmark name, type of irregularities that cause cross-iteration dependences, percentage of total execution time taken by the loop, and number of pragmas inserted for parallelization.

is used in computing correlated wavefunctions and determinations of molecular electronic structure. The loop parallelized in `trfd` has irregular memory access pattern. Each iteration of the loop calculates the runtime value of variable val and assigns it to two elements of array X. The indices of the two elements are calculated based on the runtime values of array IA. Speculative parallelization is required as two iterations may write to the same element. The memory accesses to array X are tracked. The program, `fftbench`, is from the Coyote library which comes with LLVM [75]. In the loop that was parallelized, each iteration reads and writes two elements of array $result$. The indices of the elements are calculated dynamically. The loop requires speculative parallelization as the compiler cannot identify elements accessed in each iteration.

II. Benchmarks with irregular control flows. The `mdg` [77] benchmark performs a molecular dynamics calculation of water molecules in the liquid state at room temperature and pressure. It can be used to predict a wide variety of static and dynamic properties of liquid water. The simplified version of the parallelized loop is shown in Figure 6.9. The code contains two nested loops – the outer loop is parallelized. Each iteration of this loop calculates the runtime values of array rs and counts the number of values that are larger than a threshold. The branch that can cause cross-iteration dependences is executed if the count is smaller than array rs's size. Since whether or not the branch will be executed cannot be determined at compile time, parallelizing the loop requires speculation. The execution of the branch needs to be monitored. Program `strcat` is from the shootout benchmark suite. The main loop is parallelized. Each iteration of the loop appends a fixed number of characters to a large buffer. A branch in the loop checks the free buffer space and reallocates the buffer if there is not enough space. This branch causes cross-iteration dependences and cannot be executed on the GPU. Therefore, the loop is parallelized via monitoring the execution of the reallocation branch. Program `gothic` is a gothic printer. The kernel loop is parallelized. Each iterations of the loop process one character. If a special character is met, the program reverses the background, causing cross-iteration dependences. Our implementation tracks the branch that examines the occurrence of the special character. The `alvinn` benchmark trains a neural network using backpropagation. Four loops were parallelized in the program. Each iteration of these loops need to move the *weight* pointer, thus causing cross-iteration dependences. Although the increment in each iteration is input dependent, it often does not change. To parallelize the loops, the increment is assumed unchanged at runtime and a branch is created to check this condition. The execution of the branch is monitored to detect cross-iteration dependences.

6.6.2 Performance Overview

Figure 6.10 shows the speedups for the loops considered. The baseline is the sequential execution time of the loops on the host system. Bars higher (lower) than 1 indicate speedup (slowdown).

For each benchmark there are four bars – the first bar shows the performance of our technique with all optimizations. The rest of the bars show the performance with different optimizations individually omitted (for discussion of optimization results see Section 6.6.3). The speedups for the fully optimized version are between 3.62x and 13.76x, with five (out

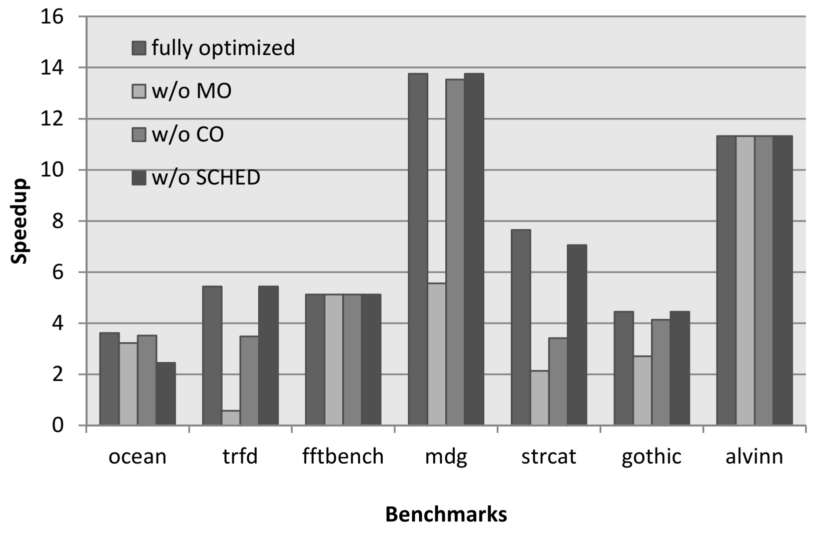


Figure 6.10: Loop speedups for different optimization.

of seven) benchmarks achieving over 5x. *The speedups demonstrate the effectiveness of our framework in using GPUs for irregular loops considered.*

6.6.3 Effectiveness of the Optimizations

Let us examine Figure 6.10 to study the effectiveness of optimizations. The second bar ("w/o MO" in Figure 6.10) gives the performance without misspeculation optimization (i.e., misspeculation detection without misspeculation localization and re-executing all iterations on the host system once misspeculation is detected). The third bar ("w/o CO" in Figure 6.10) shows the performance without copy optimization (i.e., copying all data between CPU and GPU for every assignment of iterations). The last bar ("w/o SCHED" in Figure 6.10) shows the performance without our scheduling policy (i.e., scheduling all iterations to the GPU in the first assignment). These three groups of bars are intended to show the importance of misspeculation localization, copy optimization, and our scheduling policy.

For the `ocean` benchmark, our scheduling policy improves the performance by around 32% over the one ("w/o SCHED") with minimum speedup. Our scheduling policy decreases the size of each assignment so that there is almost no misspeculation after the first few assignments. Misspeculation and copy optimizations do not improve the performance much since no misspeculation occurs in most assignments of iterations. Misspeculation optimization improves the performance of `trfd` greatly because there is always only one misspeculation in each execution of the loop. Without misspeculation optimization, all iterations will be re-executed on the CPU, which apparently will cause slowdown. Copy optimization improves its performance by 36%. The copy-in (i.e., copy from CPU to GPU) overhead is greatly reduced as only the elements that are re-computed on the CPU (for recovery) are copied to the GPU memory for every assignment. Our scheduling policy does not have much impact on the performance of `trfd` since there is only one misspeculation for each execution of the loop. The speedup of `fftbench` is partially offset by the number of memory access tracking. Since no cross-iteration dependences occur at runtime for the test input, misspeculation localization and recovery are never performed. Therefore, none of the optimizations has a performance impact. The speedup of `mdg` is high because the loop body has a lot of computation which can fully utilize the massively parallel architecture of the GPU. Also, only a few iterations execute the branch at runtime. Therefore, most computations are performed in parallel. With misspeculation optimization, only the first misspeculated iteration is re-executed on the CPU. The rest of the misspeculated iterations are assigned to the GPU in the next assignment of iterations. If all misspeculated iterations are re-executed on the CPU, the performance will be degraded by 60%. The speedup for `strcat` is good since the misspeculation rate is very low due to the rapid growth of buffer size. Misspeculation optimization improves the performance by 72% for the same reason as in `mdg`. Copy optimization is critical for performance of `strcat`. By avoiding copying the correct results back and forth between the CPU and GPU, the performance is improved by 55%. In `gothic`, a misspeculation makes all subsequent iterations incorrect. Thus, misspeculation optimization greatly reduces iterations executed on the CPU and improves the performance. The speedup of `alvinn` is high because no misspeculation happens in our experiments. The increment of the *weight* pointer does not change according to the input and thus the optimizations make no impact.

6.6.4 Overhead

Figure 6.11 shows the time overhead (divided into recovery and misspeculation check) as the percentage of the loop execution time. The time of computation and copy is a necessity for all GPU computations since they are required by the GPU computing model. The misspeculation check overhead for the `ocean` benchmark is the highest among all benchmarks because it requires memory access tracking, and its misspeculation check needs to detect both RAW/WAR and WAW dependences. The recovery overhead is high for `ocean` since the first few schedules of the loop cause many misspeculations. The misspeculation check overhead for `trfd` is lower than `ocean` since its misspeculation check only needs to

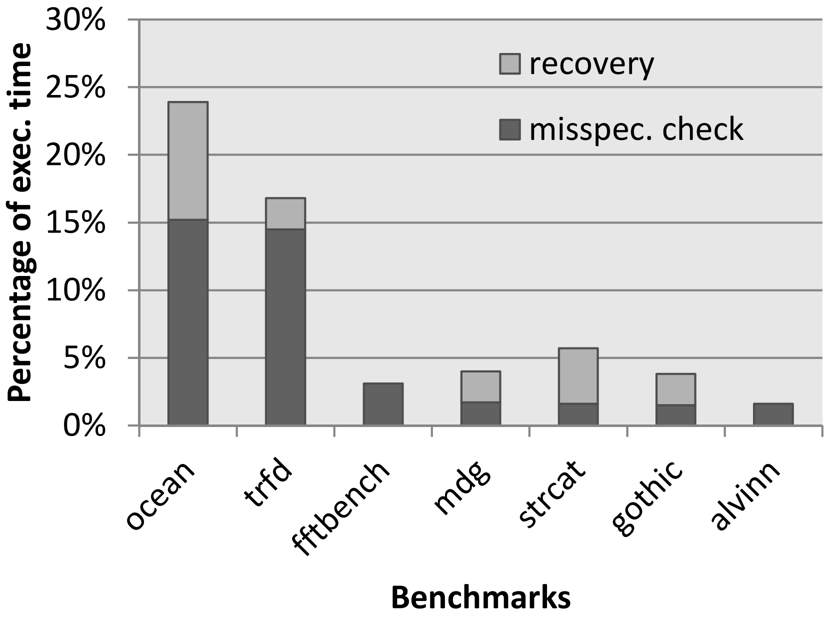


Figure 6.11: Time overhead.

detect WAW dependences. In `fftbench` and `alvinn`, since no cross-iteration dependence occurs for the test input, misspeculation localization and recovery are never performed. The recovery overhead for `mdg` and `gothic` is low since only the first misspeculated iteration is re-executed on the CPU. In `strcat`, the overhead for misspeculation check is low since the runtime only monitors the execution of the branch that reallocates the buffer.

Overall, our framework introduces very low runtime overhead for performing recovery. Recovery takes less than 10% of the loop execution time. Misspeculation check takes more time for loops with irregular memory accesses due to tracking of memory accesses. On average, misspeculation check takes around 11% of the execution time for loops with irregular memory accesses and 1.7% for the rest. *Our speculative parallelized loops spend*

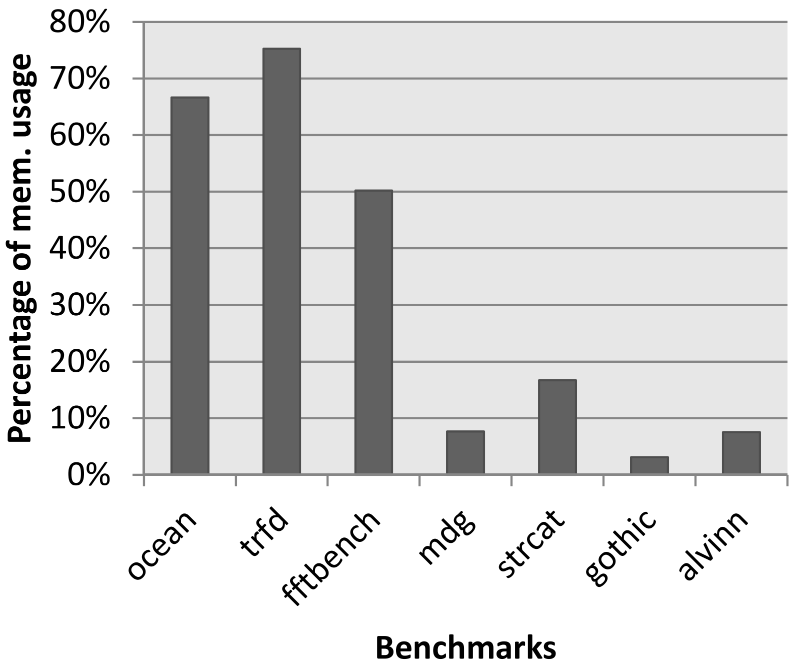


Figure 6.12: Memory overhead for misspeculation check.

more than 75% *of execution time on computation and copy, demonstrating its efficiency.*

Figure 6.12 shows the percentage of GPU memory usage for misspeculation check, including the memory space used for memory access tracking, misspeculation detection, and misspeculation localization. For benchmarks `ocean`, `trfd`, and `fftbench`, misspeculation check requires significant GPU memory space because they require checking cross-iteration dependences between memory accesses. Benchmark `fftbench` spends less memory on misspeculation check than `ocean` and `trfd` because no cross-iteration dependence occurs at runtime. The memory overhead for the rest of the benchmarks is low since they only require monitoring the execution of the branches.

6.7 Summary

This chapter has described a framework that extends SpiceC to allow the use of GPUs. Further, it shows how GPUs may be used to speculatively parallelize loops that may have cross-iteration dependences at runtime due to irregularities. Several optimizations were proposed to improve the performance, including parallelizing misspeculation check on the GPU, optimizing the procedure of result committing and misspeculation recovery, and adaptive scheduling policy for different types of cross-iteration dependences. An extension to the SpiceC programming model was presented for writing speculative parallel loop on GPUs. Parallelizing a loop with the SpiceC extension only requires inserting one or two pragmas into the loop.

Chapter 7

SpiceC on Distributed Memory Clusters

Modern applications from important domains (e.g., data mining, machine learning, and computational geometry) have high processing demands and a *very large memory footprint*. To meet the demands of such applications, it is natural to consider the use of clusters of multicore machines. Clusters can not only scale to a large number of processing cores, they also provide a large amount of memory via the scalable distributed memory architecture. SpiceC can be adapted to the distributed memory architecture since its copying and commit computational model (as described in Chapter 2) does not require shared memory and the data transfers between shared and private spaces are controlled by software. The challenge for porting SpiceC to clusters is to implement the mechanisms for data transfers between private and shared spaces so that they can work for distributed memory. Since the data transfers are handled by the SpiceC compiler and runtime, this modification is transparent to the user. Programs parallelized using SpiceC can be executed on distributed memory clusters without placing additional burden on the programmer. This chapter demonstrates the ability of SpiceC to exploit the scalability of a distributed-memory system, consisting of a cluster of multicore machines.

Unlike the SpiceC runtime on multicore processors, which stores all private and shared spaces in the same physical memory, the SpiceC runtime on clusters distributes the private and shared spaces across different machines. Therefore, to enable SpiceC on clusters, the SpiceC runtime needs to address the following issues:

- The SpiceC runtime should provide a shared-memory abstraction on top of distributed memory to hide explicit communication from users and give them an illusion of physically shared memory since SpiceC has a shared memory programming model.
- Since network communication is time-consuming, the overhead of network communication should be carefully handled for better program performance.

This chapter presents SpiceC-DSM (SpiceC Distributed Shared Memory), which is a runtime system designed to enable SpiceC on clusters. With the support of SpiceC-DSM, users can write shared memory parallel programs using the SpiceC programming model and execute them on clusters, which greatly facilitates parallel programming on clusters. SpiceC-DSM also move the overhead of network communication off the critical path of program execution to improve the performance on clusters.

7.1 Shared Memory Abstraction

To provide an efficient shared Memory abstraction, SpiceC-DSM creates a two-level virtual memory hierarchy on a cluster. Figure 7.1 shows the two-level virtual memory hierarchy. The second level *virtual shared memory* aggregates the physical memories from different machines to provide a shared memory abstraction, which corresponds to the shared space in the SpiceC computation model (as described in Chapter 2). The first level *virtual private cache* is created for each thread for temporary storage of data to exploit the temporal locality and reduce the network overhead, which is also used as the private space in the SpiceC model. According to the SpiceC computation model, each thread must first copy the required data from the virtual shared memory to its private cache before operating on them. After that, the thread can continue to use the data copies in its private cache until they are committed. The commit of modified data copies, in the private cache, to the virtual shared memory makes the results visible to other machines.

The remainder of this section describes how SpiceC-DSM provides users with the second level `virtual shared memory`. It first illustrates what data structures to be shared across the cluster and then describes how they are distributed and accesses.

7.1.1 Definition of Shared Data

In SpiceC-DSM, all shared data structures must be dynamically allocated and deallocated. In other words, only dynamic data structures will be shared across the cluster. This avoids the overhead of sharing unnecessary data across the cluster and works perfectly for applications with large data sets such as data mining applications since they usually

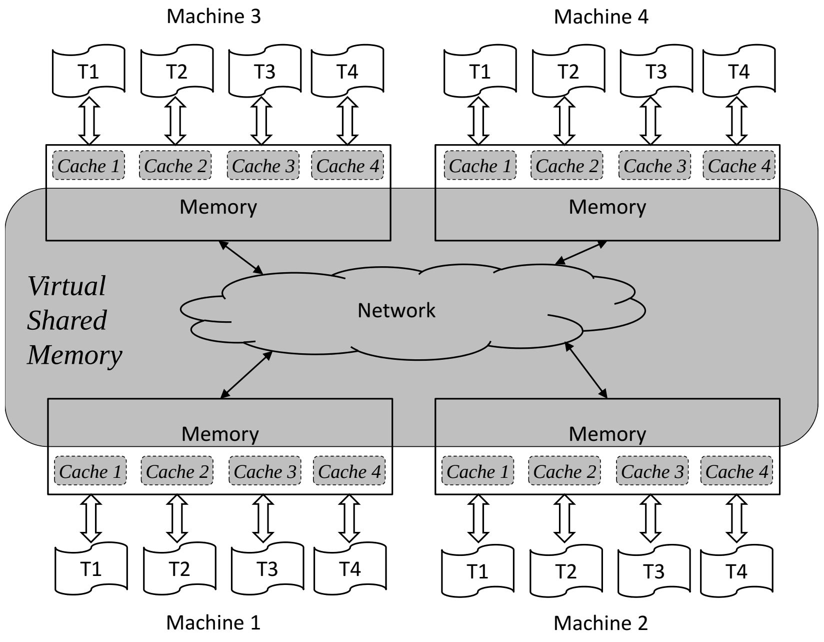


Figure 7.1: Memory hierarchy of SpiceC-DSM on a modern cluster.

use dynamic data structures to hold the data sets. Arrays and scalar variables can also be shared using SpiceC-DSM as long as they are dynamically allocated.

Similar to the SpiceC support for dynamic data structures presented in Chapter 4, SpiceC-DSM must be able to identify the shared data structures. To achieve this, SpiceC-DSM requires that the data structures to be shared across the cluster inherit a base class, called `BaseNode`. To support proactive communication described in Section 7.2, SpiceC-DSM also needs to learn the topology of the shared data structures. The `BaseNode` class defines two interfaces (i.e., virtual functions in C++) — `GetNeighborNum` and `GetNeighbor` that the developers must implement. The first interface returns the total number of neighbors and the second returns a specified neighbor. The SpiceC-DSM can acquire the topology by calling these two functions.

7.1.2 Data Distribution

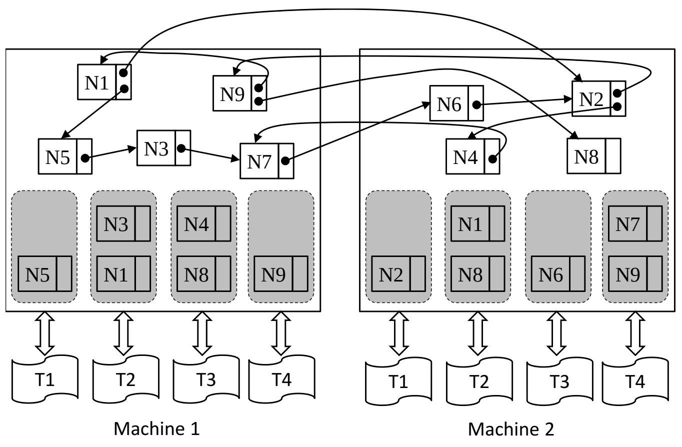


Figure 7.2: An example of dynamic data structure stored in SpiceC-DSM.

In SpiceC-DSM, dynamic data structures are distributed across multiple machines. Figure 7.2 shows an example of a graph data structures stored in SpiceC-DSM. The nodes in the graph are distributed among two machines.

In SpiceC-DSM, data distribution is done automatically at the allocation sites as all shared data structures are dynamically allocated. When an element is allocated at runtime, it is automatically assigned to one of the machines in the cluster using a hash function. This machine is called the *owner* of the element and maintains a master copy of the element. When an element is deallocated, it will be removed from its owner.

7.1.3 Data Reference

Since all shared data structures are dynamically allocated, they can only be accessed via pointers. As shown in Figure 7.2, an edge in the graph may connect two nodes on the same machine or on different machines. Therefore, a thread should be able to access a data element stored on a remote machine via a pointer.

In SpiceC-DSM, a pointer stores the ID of a shared data element. Each shared data element is assigned with an ID when it is allocated in SpiceC-DSM. The ID is a string unique to the element. It is used to access the element, acting as the virtual address of the element. When the element is distributed, its *owner* is selected by hashing the ID. Therefore, when a pointer is dereferenced, the owner of the data element can be easily located by hashing the ID stored in the pointer. According to the SpiceC computation model, the corresponding data element will be copied into the local virtual cache. All references to the data element will access the local virtual cache, which is much faster than remote accesses. If the data element is updated, it will be copied back to its owner when the computation is committed.

7.1.4 Sequential Consistency

To ensure that the programmer can easily reason about a program, SpiceC-DSM employs a form of *sequential consistency* as follows. As shown in Chapter 2, each parallel loop written in SpiceC is divided into regions by the `region` directives. With SpiceC-DSM, each region is executed with atomicity and in isolation since the data are copied into the local virtual cache before being accessed and copied back to its owner at the commit site of the region. Sequential consistency is then enforced by SpiceC-DSM at the coarse grain level of regions. For performance, the compiler can freely reorder instructions within a region without the risk of violating coarse grain sequential consistency.

7.2 Proactive Data Communication

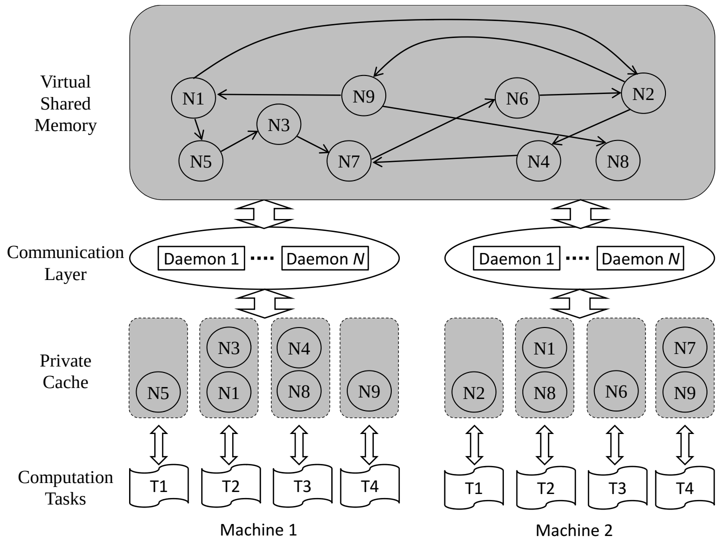


Figure 7.3: Data communication in SpiceC-DSM.

SpiceC-DSM introduces a communication layer to perform data communication proactively between the virtual private caches and the virtual shared memory, as shown in Figure 7.3. This layer is composed of a set of daemons that handle data communication in parallel with the actual computation. They proactively fetch data from remote machines for the local computation tasks and forward the updates from the local computation tasks to the owners. The daemons ensure that the data required by the computation tasks are *mostly* available locally when needed. In this way they hide the network latency from the computation tasks. As a result, the computation tasks only need to operate on data in the local memory, which leads to better performance. Proactive data communication layer hides *read latency* by proactively fetching data needed by computation tasks and it hides *write latency* by proactively committing updates to the owners. Detailed discussion of both these mechanisms follows.

I. Proactive data fetching. When a computation task wants to access a data element, it first checks whether a replicated copy already resides in the local virtual cache. If not, the computation task will have to wait for a remote access for fetching the data element. To avoid this case, the daemons in the data communication layer proactively fetch data for the computation tasks.

To perform proactive data fetching, when a data element is accessed by a computation task for the first time, a communication daemon also predicts what elements are likely to be accessed next. If the predicted elements have not previously been copied into the local virtual cache, then the daemon sends data requests to their owners. In applications that use dynamic data structures such as graphs and trees, each computation often accesses a set of connected data elements. Therefore, when a data element is accessed, its neighbors are predicted to be accessed next and proactively fetched by the communication daemon. This is done using the two interfaces – `GetNeighborNum` and `GetNeighbor` which developers implement for the shared data structures.

A computation task will still be blocked when the data elements it requests are not in the local virtual cache. In this case, the access latency for these data elements is on the critical path of the computation.

II. Proactive data committing. At the end of each computation, the computed results in the virtual private cache need to be committed to their owners. The data commit is performed by one of the daemons in the proactive communication layer to keep the overhead off the critical path. Atomicity check must be performed before data commit to ensure sequential consistency. During the atomicity check, the communication daemon communicates with the owners to see if the elements in the virtual private cache have been updated by other computation tasks. If so, the data in the virtual private cache must be discarded and the computation must be redone. To ensure sequential consistency, multiple commits from the same computation task should be performed one by one.

Proactive data communication does not only hide the network latency from the computation tasks, but also makes communication asynchronous. With asynchronous communication, multiple communication requests can be processed simultaneously to further hide the network latency.

7.3 SpiceC Programming on Clusters

SpiceC programming on clusters is similar to that on shared memory systems (as described in Chapter 2). The only difference is that only dynamic data structures are shared on clusters.

Figure 7.4 shows an example of parallel graph coloring written in SpiceC. The first half of the code reads input from a file and constructs the graph sequentially. Since it is sequential region, it will only be executed on the main thread. The first loop reads the node information and allocates the nodes. As the nodes are dynamically allocated, they will be shared in SpiceC-DSM. The second loop reads the edge information and connects the nodes accordingly. The objects shared in SpiceC-DSM are just used as normal objects. All objects accessed in the sequential region will be cached in the private space of the main thread. They will be copied back to the shared space before entering the parallel region so

```
// sequential region: read input and construct graph
for(i=0; i<nodes; i++) {     // read nodes
   read_node(file, &v);
   node_ptrs[id] = new Node;
   node_ptrs[id]→value = v;
}
for(i=0; i<edges; i++) {     // read edges
   read_edge(file, &id1, &id2);
   node_ptrs[id1]→neighbors.add(node_ptrs[id2]);
}
// parallel region: perform graph coloring
#pragma SpiceC parallel doall
for(int i=0; i<Node.size(); i++) {
   #pragma SpiceC region R1 {
      coloring(Node[i]);
      #pragma SpiceC commit atomicity_check
   }
}
```

Figure 7.4: Pseudocode for graph coloring.

that other threads can see the results from the sequential region.

The second half of the code example performs graph coloring in parallel. It is just like writing a regular DOALL loop using SpiceC. Since the `Node` structure inherits the base class `BaseNode` (as described in Section 7.1.1), member function `size()` returns the total number of `Node` objects and each `Node` object can be referred to by using an index with the `Node` type. In the DOALL loop, they are used to enumerate all objects with the `Node` type. Since multiple iterations of the parallel loop may access the same node, `atomicity_check` is specified at the commit site to enable speculation.

7.4 Evaluation

This section evaluates SpiceC-DSM. The SpiceC-DSM runtime is implemented using *Memcached*, which is an open-source distributed memory object caching system. In the experiments, the proactive communication layer on each machine consists of 32 daemons. The number is limited by the size of Memcached connection pool. When no data needs to be transferred, the daemons will be blocked, thus consuming no CPU resource.

The experiments were conducted on a cluster consisting of five eight-core DELL PowerEdge T605 machines. Table 7.1 lists the details of each machine. These machines ran *Ubuntu 11.04*. In the experiments, SpiceC-DSM uses the memory of all five machines no matter how many cores are used for computation.

Processors	2×4-core AMD Opteron processors (2.0GHz)
L1 cache	Private, 64KB for each core
L2 cache	Private, 512KB for each core
L3 cache	Shared among 4 cores, 2048KB
Memory	8GB RAM
Network	1GB Ethernet

Table 7.1: Machine details.

7.4.1 Benchmarks

To evaluate SpiceC-DSM, the experiments were performed using 6 applications, including Delaunay Refinement, Graph Coloring, K-means Clustering, PageRank, BlackScholes, and Betweenness Centrality. Since SpiceC-DSM is built for applications with large data sets, these applications were chosen as they are all data-intensive. Most of the applications are widely-used data mining programs designed to process large volume of data.

Benchmark	Data Struct.	Speculation?
Delaunay Refinement	Graph	Yes
Graph Coloring	Graph	Yes
Betweenness Centrality	Graph	Yes
K-means Clustering	Dynamic Array	Yes
PageRank	Graph	No
BlackScholes	Dynamic Array	No

Table 7.2: Benchmark summary.

Table 7.2 summarizes the applications. `Delaunay Refinement` is an implementation of a mesh generation algorithm. It transforms a planar straight-line graph to a Delaunay triangulation of only quality triangles. The data set is stored in a pointer-based graph structure. Each iteration of the main loop updates a cavity which is a subset of connected nodes. Speculation is used since multiple threads can update the same nodes. `Graph`

`Coloring` is an implementation of the scalable graph coloring algorithm proposed in [21]. Each computation task colors one node using the information of its neighbors. `K-means Clustering` [39] is a parallel implementation of a popular cluster analysis algorithm, which aims to partition n points into k clusters where each point belongs to the cluster with the nearest mean. All points are stored in a dynamic array in the implementation. Each iteration of the parallel loop assigns a node to the nearest cluster and updates the centroid of the cluster. Since multiple iterations may update the same cluster, the program needs speculation. `PageRank` is a link analysis algorithm used by Google. It iteratively computes weight for every node of a linked webgraph. In the implementation, a dynamic data structure is used to hold the webgraph. `BlackScholes` is a benchmark from the PARSEC benchmark suite [16]. It uses the Block-Scholes partial differential equation to calculate the prices of European-style options. All data are stored in dynamic arrays in the implementation. `Betweenness Centrality` is a popular graph algorithm used in many areas. It computes the shortest paths between all pairs of nodes. For a given node, it determines what fraction of shortest paths contain that node. Speculation is used to atomically update the betweenness centrality of every node.

7.4.2 Scalability

Figure 7.5 shows the application speedups achieved on SpiceC-DSM using different numbers of cores for computation. In the experiment, every group of eight cores are on the same machine. The baseline is the sequential versions of the same applications. For all sequentail and parallel runs, Spcice-DSM uses the memory of all five machines. We can

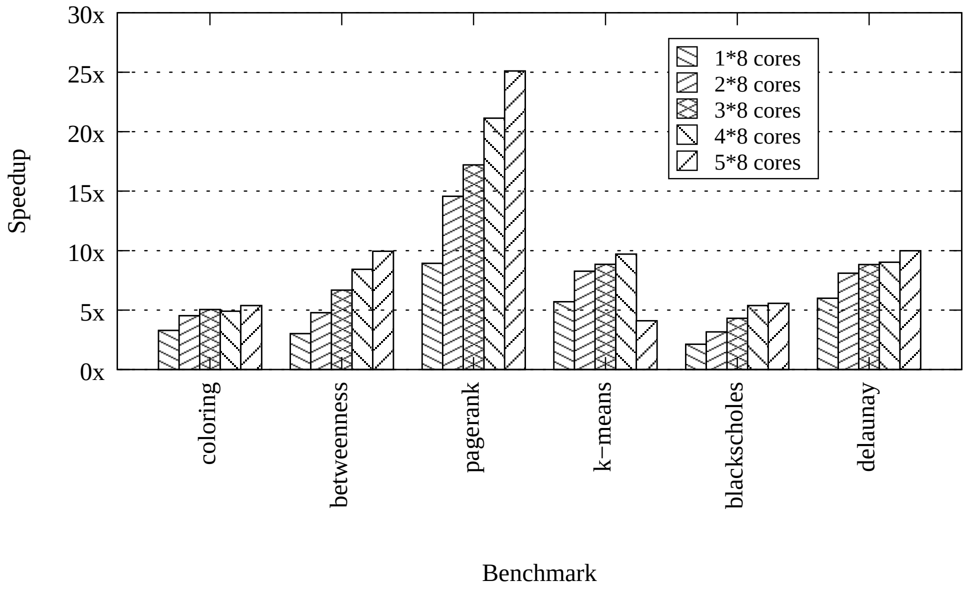


Figure 7.5: Application speedups.

see that the performance of most applications scales well with the number of cores used for computation. The parallel versions are always faster than the sequential version with speedups up to 25x. The performance of `k-means` drops at 40 cores due to very high misspeculation rate, which is around 40%. The average speedup at 40 cores is 7x, which demonstrates the SpiceC-DSM's ability to handle parallel applications.

7.4.3 Impact of Copying and Commit Model

SpiceC-DSM uses the copying and commit model to reduce the number of remote accesses occurring at each threads. Private spaces are used to hold temporary data copies for consecutive accesses. Without private spaces, every data accesses will go to the virtual shared memory, greatly increasing the number of remote accesses. Figure 7.6 shows the execution time of `blackscholes` with and without private spaces. All time data are

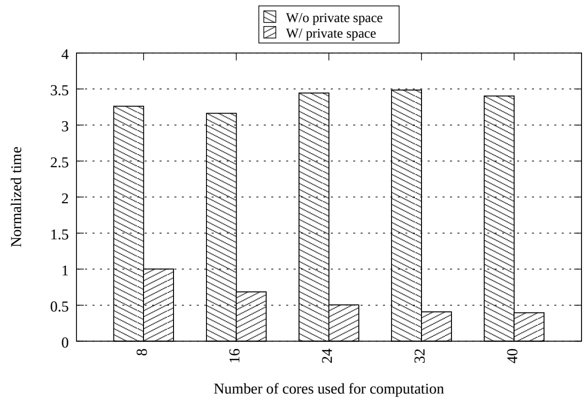


Figure 7.6: Execution time of `blackscholes` with and without private spaces.

normalized to the execution time with private spaces using 8 cores. We can see that the performance with private spaces is always better than without private spaces, which shows that the copying and commit model can efficiently improve the program performanc on distributed memory systems.

7.4.4 Impact of Proactive Communication

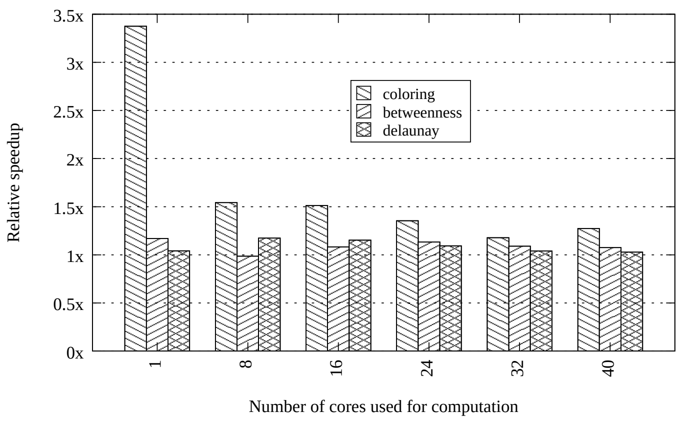


Figure 7.7: Relative speedup: with proactive communication vs. without proactive communication.

Figure 7.7 shows the performance improvement achieved by proactive communication for three applications. Sequential programs are used for measuring the performance on 1 core. We can see that proactive communication can improve the program performance by up to 3.4x. It improve the performance of both sequential and parallel programs.

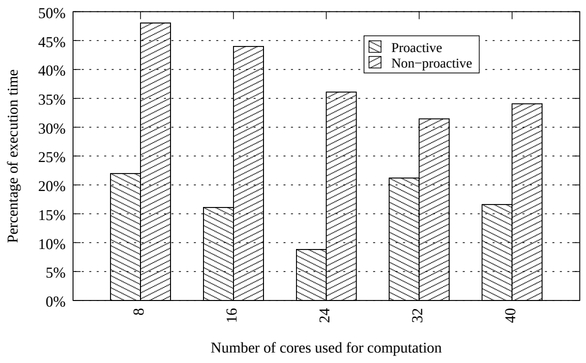


Figure 7.8: Speculation overhead in the execution of `graph coloring` with and without proactive communication.

Figure 7.8 shows the speculation overhead in the execution of `graph coloring` with and without proactive communication. The speculation overhead with proactive communication is always lower than that without proactive communication.On average, proactive communication reduces the speculation overhead by 60% for `graph coloring`, which demonstrates its ability to hide speculation overhead.

7.4.5 Impact of Data Contention

Figure 7.9 compares the speedups of `k-means` with high and low data contention. Input data sets are calibrated to produce different data contention. We can see that SpiceC-DSM achieves better performance when data contention is low. Even when data contention is high, SpiceC-DSM can still achieve speedups for `k-means`, which shows that SpiceC-DSM

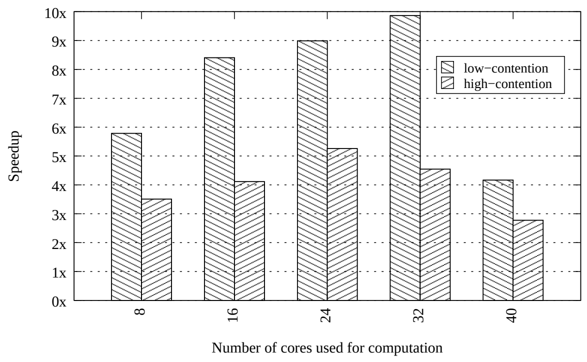


Figure 7.9: Speedups of `k-means` with high data contention and low data contention.

can efficiently handle data contention.

7.4.6 Bandwidth Consumption

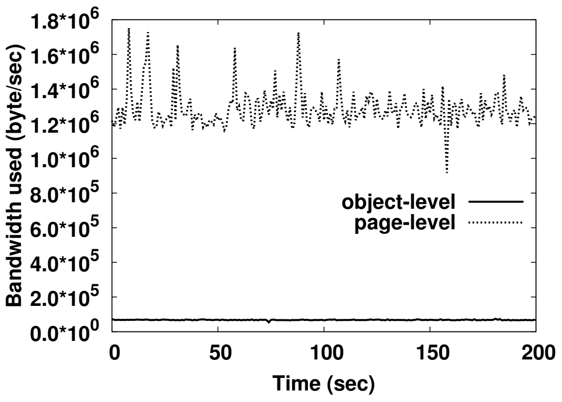


Figure 7.10: Comparison of page-level and object-level data transfer.

Previous cache-coherence DSM systems [103] typically perform data transfers at page level when maintaining memory coherence. However, a cluster of nodes accessed by a code segment (e.g., a node and its neighbors in a graph), are not necessarily allocated consecutively in the memory space. Thus, little benefit can be achieved from the high communication overhead paid for page level data transfer. SpiceC-DSM transfers data between shared and private spaces at object level. Figure 7.10 compares the bandwidth consumed by page level and object level data transfer using the parallel *graph coloring*. As we can see, data transfers at page level cause far greater level of communication than object level transfers. Thus, it is desirable to use SpiceC-DSM for applications that rely on the use of dynamic data structures.

7.5 Summary

This chapter presented SpiceC-DSM, which is designed to enable SpiceC on clusters. SpiceC-DSM aggregates the memory from different machines to form a shared memory abstraction, which meets the demand of applications that have large data sets. SpiceC-DSM introduces proactive data communication, which uses the on-chip parallelism provided by multicore machines to hide the network latency from the computation tasks. SpiceC-DSM also offers sequential consistency through the atomic execution of code regions.

Chapter 8

Related Work

This chapter discusses the related research in the two main areas addressed by this dissertation. First, the state of the art in parallel programming is discussed. Second, since speculative execution is a key component of all the solutions developed in this work, related work in this area is presented. The works on parallel programming and speculative execution are presented for all types of architectures, i.e., multicore CPUs, heterogeneous multicores with GPUs, and distributed memory clusters. The final section discusses some miscellaneous works not covered in the sections on parallel programming and speculative execution.

8.1 Parallel Programming

The related work on parallel programming is discussed next. The discussion is divided according to the parallel computing platforms that it is aimed at.

8.1.1 Shared Memory Systems

Many programming models have been proposed to write parallel programs for multiprocessors. OpenMP [36] is probably the most widely used programming model for parallelizing sequential programs on shared-memory systems. Similar to SpiceC, OpenMP also uses compiler directives to express shared-memory parallelism. OpenMP gives developers a simple interface to parallelize their programs. A few extensions to OpenMP have been proposed for speculative parallelization of C/C++ code. OpenTM [12] is an extension to OpenMP, which is a widely-used API for shared-memory parallel programming. The new compiler directives are designed to express loop-level speculative parallelization based on TMs. Milovanović et al. [90] proposed an extension to OpenMP that supports a multithreaded STM design with a dedicated thread for eager asynchronous conflict detection improving time efficiency.

Threading Building Blocks (TBB) [108] is another programming model used to parallelize programs for shared-memory systems. Instead of using compiler directives, TBB provides a set of threadsafe containers and algorithms to allow developers to easily write parallel programs. To use TBB, developers need to wrap their codes using the containers provided by TBB. Like OpenMP, TBB does not support speculative parallelism. To parallelize a loop with cross-iteration dependences, developers need to manually use low level primitives to handle the dependences.

Galois [72, 71, 70, 89, 99] provides a runtime library for exploiting the data parallelism in irregular applications. It supports speculative parallelism but developers must provide code to perform rollback once speculation fails. It has been extended to use data partitioning for better locality and lower misspeculation rate. However, partitioning is not done automatically and should be provided by developers.

Bamboo [146] is a data flow language designed to exploit parallelism on manycore processors. It is a data-oriented extension to Java. Bamboo programs are composed of a set of tasks that implement the programs operations. Developers focus on how data flows between tasks. Parallelism is achieved through the parallel execution of independent tasks.

Prabhu et al. [101] proposed two language constructs to enable developers to achieve parallelism via the use of value speculation. They required all parallel tasks to be defined in a producer and consumer model. Prediction functions provided by developers are used to predict values of data dependences between tasks for increasing parallelism. A static analysis was proposed to reduce the misspeculation overhead.

There are a few other parallel programming interfaces proposed to improve memory performance on shared memory systems. Sequoia [45] is a memory hierarchy aware parallel programming model. In Sequoia, developers explicitly control the movement and placement of data at all levels of the machine memory hierarchy to make parallel programs bandwidth-efficient. Hierarchically Tiled Array (HTA) [17] accelerates sequential OO languages by using an array to express multiple levels of tiling for locality and parallelism.

None of the above programming models are as versatile as SpiceC. They are all designed to exploit parallelism in a specific class of applications with the same form of parallelism. They are unable to parallelize real applications that use precompiled functions and system calls. While keeping simplicity, they do not support more scalable systems like clusters. Except OpenMP and TBB, these programming models are not as easy to use as SpiceC. They do not have a compiler for automatically generating code for speculative execution. Some programming models like Bamboo and Sequoia require the programs to be written in a disciplined style.

8.1.2 Distributed Memory Systems

Single Program Multiple Data (SPMD) is a programming model to achieve parallelism on distributed memory systems. Message passing interface (MPI) [52] is currently the de facto standard for SPMD. Using MPI, developers need to explicitly write parallel programs, including manually distributing workloads among threads and handling all communications and synchronizations between threads. Since developers can fully control their programs in MPI, MPI is good for optimizing performance and managing data locality. Partitioned global address space (PGAS) is a parallel programming model which tries to combine the performance advantage of MPI with the programmability of a shared-memory model. It includes Unified Parallel C [34], Co-Array Fortran [44], Titanium [57], Chapel [27] and X10 [28]. Unified Parallel C, Co-Array Fortran, and Titanium use SPMD style and try to improve the programmability of SPMD. Chapel and X10 extend PGAS to allow each node to execute multiple tasks from a task pool and invoke work on other nodes. These PGAS programming models explore parallelism for array-based data-parallel programs using data partitioning [66, 110]. Since these works focus on array-based data parallel programs, they do not need to consider dynamic computation partitioning and data contention between threads. High Performance Fortran was extended for dynamic data distribution to parallelize array-based unstructured computations [92]. Several works [137, 51] have focused on data and task partitioning for stream programs.

Software DSM has been proposed to provide the easy of shared-memory programming on distributed-memory systems. Callahan and Kennedy [24] have proposed a software DSM mechanism that executes programs in SPMD mode. In SPMD mode, every CPU executes the same program but performs only the instructions that access data local to the CPU. The other series of works is Cache-coherent DSM [115, 116], which was mostly inspired by virutal memory and cache coherence protocol on shared-memory systems [103]. They allow data replication and migration so that most data accesses cab hit the local memory as long as the application exhibits high locality. The above DSM systems were not developed for modern clusters, where each node has many processing cores. They do not make use of the multiple cores on each node. In addition, these DSM systems work poorly for applications that use using dynamic features provided by modern programming languages. The performance of SPMD-based DSM is greatly degraded for modern applications as these dynamic features make the appropriate distribution of data impossible to determine at compile-time. In a cache-coherent DSM system, data need to be replicated and/or migrated on the fly as it is impossible to statically figure out the memory access patterns for programs using dynamic features. SpiceC overcomes these drawbacks. It allows automatic data distribution while keeping the communication overhead low using the copying and commit model. SpiceC also moves most communication overhead off the critical path using the multicore architecture on each machine.

Distributed caching is a simplified DSM mechansim for caching objects in distributed-memory systems. Memcached [50] is a popular implementation of distributed caching. It is designed to speed up web applications by alleviating database load. For the system's simplicity, users need to manually load and store data in the system. The system does not provide any guarantee for memory consistency. Besides, users must manually serialize data before puting them into the distributed cache since the system does not understand data structures, which further increases the programming burden. Unlike SpiceC, Memcached does not provide any support for keeping the communication overhead off the critical path.

Recently, software transactional memory [118] has been introduced to clusters as a parallel programming paradigm. Distributed Multiversioning (DMV) [84] is a page-level distributed memory coherence algorithm that allows transactions on different machines to manipulate shared data structures in an atomic and serializable manner. It is designed for parallel database applications. Cluster-STM [20] studied metadata distribution and communication aggregation for STM on clusters. It also introduced a low-level STM API for clusters. DiSTM [67, 68] is an object-level transactional memory for clusters. It supports multiple transactions to commit concurrently. Many software DSM libraries also provide certain levels of consistency to support transactional applications, such as snapshot isolation [79, 139], session consistency [37], and persistent sotre [123]. Compared to SpiceC, none of the above systems is designed to keep the communication and speculation overhead off the critical path by using the multicore architecture on each machine.

MapReduce [38] is a programming framework for processing large data sets on distributed-memory systems. The framework consists of two steps: *map* and *reduce*. In the *map* step, the master node divides the computation into smaller sub-computations and distributes them to the worker nodes. In the *reduce* step, a set of *reducer* nodes combine the results from the worker nodes to generate the final output. While MapReduce may be easy for developers to adopt for simple or one-time processing tasks [98], it is non-straightforward to port sophisticated tasks with complex dependences into MapReduce. It requires users to manually map the computation into the *map* and *reduce* steps. In comparison with MapReduce, SpiceC do not require significant modification to the original programs.

8.1.3 Heterogeneous Systems With GPUs

Several parallel programming models have been proposed for GPU computing. OpenCL [122] is a programming model for parallel programming of heterogeneous systems. The CUDA programming model [94] is a standard for NVIDIA's parallel computing architecture – CUDA. Both programming models provide APIs for writing kernels that execute on GPUs and defining/ controlling the GPUs. OmpSs [10] is a programming model that extends the OpenMP standard [36] for the GPU architectures. It provides pragmas for programmers to specify the data flow requirements for kernels, which can be used to optimize memory accesses. OmpSs also provides APIs for specifying architecture heterogeneity, which is used to improve kernel scheduling. Lee et al. [76] developed an OpenMP-to-CUDA compiler that allows programmers to write GPU programs using OpenMP directives. *None of these programming models support speculative parallelization for GPU computing.*

8.2 Speculation Execution

8.2.1 Automatic Speculative Parallelization

I. Multicore CPUs. Speculative parallelization has been widely studied to improve the program performance on CPUs. It was earlier proposed for automatically parallelizing loops on the Symmetric Multiprocessor (SMP) machines [107]. To parallelize a loop on SMPs, the loop is run in parallel and the existence of cross-iteration dependences is checked. If any cross-iteration dependence is found to occur, the sequential version of the loop is executed to obtain the correct results. Recovery for individual iterations is not supported.

Recently, thread level speculation (TLS) techniques have been proposed to explore more parallelization opportunities for sequential programs. Krishnan et al. [69] proposed an architecture for hardware-based TLS. Recently, many works have focused on software-based TLS. Behavior oriented parallelization (BOP) [43, 65] is a process-based speculation technique. Copy or Discard (CorD) [127, 129] is a thread-based speculation technique. All these TLS techniques are based on state separation. In other words, speculative computations are performed in a separate memory space. The results are not committed to the non-speculative space until the speculation succeeds. CorD was extended to support a set of compiler optimizations for copying dynamic data structures between spaces [129]. BOP [43, 65] improves the performance of data copying between spaces with the help of OS.

Transactional memory has been used to enable speculative parallelization. Mehrara et al. [86] customized a low-cost STM for facilitating profile-guided automatic loop parallelization. They used compiler analysis to eliminate a considerable amount of checking and locking overhead in conventional software transactional memory models

II. Heterogeneous multicores with GPUs. Speculative execution has been used to explore task-level parallelism on multi-GPU systems [40]. Usually, the runtime system must block the execution of a kernel until its predecessors in the control flow graph (CFG) have finished. On multi-GPU systems, the performance is limited by the runtime system's inability to execute more kernels in parallel. Diamos and Yalamanchili [40] alleviated this problem by speculating the control flows between kernels. They allow kernels to be executed speculatively in parallel before their predecessors in the CFG have been finished. Unlike their work, this paper explores thread-level speculative parallelism in a kernel. An exploratory study has been done for speculative execution on GPUs [80, 81]. The preliminary solution proposed in the study cannot handle any benchmark used in the experiments of Chapter 6. It assumes that programs have irregular reads but regular writes, which is often not true in real applications.

8.2.2 Transactional Memory Based Compilers

I. Unmanaged languages. The work closest to SpiceC on code generation for speculative execution is the Intel STM Prototype Compiler [3, 134, 93]. Both approaches provide high-level programming constructs for C/C++ in the form of an atomic block or transaction. However, the SpiceC compiler differs from the Intel STM Compiler in several major aspects. First, the Intel STM compiler requires programmers to annotate every function called in transactions while the SpiceC compiler does not. The experiments in Chapter 3 have shown that annotating functions called in transactions places a substantial burden on the programmer. Second, SpiceC provide programming and compiler support for precompiled library functions and irreversible functions. Programmers can use our programming constructs to annotate these functions to enable concurrent execution of the transactions containing them. In contrast, the Intel STM compiler serializes the execution of the transactions that contain them. Third, SpiceC provides constructs for programming synchronization between transaction commits while the Intel STM compiler does not. Fourth, the Intel STM compiler is designed to use an undo logging-based STM [114] while SpiceC adopts write buffering-based STMs. This makes SpiecC's code generation and optimization different from theirs. For example, the Intel STM compiler needs to handle aliasing of stack allocated variables for transaction aborts while the SpiceC compiler does not. The SpiceC compiler eliminates unnecessary searches in write buffers, which the Intel STM compiler does not need to perform. Finally, the SpiceC compiler is compared with the Intel STM compiler in the experiments. Code generated with the SpiceC compiler outperforms code generated with Intel's by 20.8% using 8 threads.

Some other works have also introduced high-level TM constructs into C or C++. Tanger [47] is another STM compiler for C/C++. Similar to the Intel STM compiler, it does not provide STM constructs for synchronization, precompiled library functions or irreversible functions. It relies on dynamic instrumentation to support precompiled library functions in transactions. Luchangco et al. [82] theoretically analyzed different design options for integrating transactional memory into the C++ programming language.

II. Managed languages. Many works have introduced TM constructs into managed languages. AtomCaml [109] introduced first-class constructs to support atomic execution of code written in Objective Caml, which is based on a uniprocessor execution model. Adl-Tabatabai et al. [7] presented compiler and runtime optimizations for TM constructs in JAVA. Their system supports composition of transactions and partial roll back. They use just-in-time (JIT) optimizations on STM operations. Hindman and Grossman [58] developed a source-to-source translator to support atomicity in JAVA. Their implementation is based on locks. Harris et al. [54] developed a STM compiler for Bartok, an optimizing ahead-of-time research compiler and runtime system for Common Intermediate Language (CIL) programs.

III. Binary rewriting-based systems. Dynamic binary instrumentation has been used to support atomic execution of code. JudoSTM [96] is a dynamic binary-rewriting approach that implements STM in C/C++ code. It uses value-based conflict detection and supports transactional execution of both library functions and irreversible functions. Tarifa [47] and LDBTOM [135] are two other binary rewriters that instrument binary code with calls to a STM library. They are designed to enable atomic execution of legacy library functions. Roy et al. [111] proposed to incorporate static and dynamic binary rewriting techniques to reduce the binary instrumentation overhead for STM systems.

8.3 Other Works on Enhancing Performance

8.3.1 Enhancing Performance on GPUs

This dissertation handles irregularities that cause cross-iteration dependences on GPUs. Irregularities may also severely limit the efficiency of GPU computing due to the warp organization and SIMD (Single Instruction Multiple Data) execution model of GPUs. Many software optimizations have been proposed to reduce the impact of irregularities. Zhang et al. [144] gave a few runtime optimizations with the support of a CPU-GPU pipeline scheme to remove thread divergences. They also proposed two code transformations [145], data reordering and job swapping, to remove irregularities in both memory accesses and control flows. Baskaran et al. [14] proposed a polyhedral compiler model to optimize affine memory accesses in regular loops. Lee et al. [76] developed an OpenMP-to-CUDA compiler that optimizes memory accesses within loops. Ryoo et al. [113] gave a few optimization principles for manually improving memory accesses. Yang et al. [140] presented an optimizing compiler for memory bandwidth enhancement, data reuse and parallelism management, and partition-camping elimination.

8.3.2 Enhancing I/O Performance

Many techniques have been proposed to improve the I/O performance in parallel programs.

I. Parallel I/O. Parallel I/O has been proposed to improve the performance of multiple I/O operations at the same time. It is mostly designed to deal with massive amounts of data on distributed systems. Research work in parallel I/O can be mainly divided into two different groups: parallel file systems and parallel I/O libraries. Parallel file systems [119] usually spread data over multiple servers for high performance. They allow shared accesses to files from multiple processes. Parallel I/O libraries such as ROMIO [124] are APIs designed to access parallel file systems. Collective I/O [125] has been proposed to optimize non-contiguous I/O requests from multiple processes; it coordinates accesses to files by a group of processes in which collective I/O functions are called.

SpiceC support for hybrid loops are orthogonal to parallel I/O. Parallel I/O provides lower-level programming constructs designed to improve the I/O throughput of large-scale systems. However, when using parallel I/O, programmers must be highly skilled in order to express, and make efficient use of, parallel I/O operations. SpiceC makes it easy for programmers to parallelize hybrid loops by providing a higher-level programming model. The SpiceC compiler is designed to optimize the performance of hybrid loops written in the SpiceC model.

More specifically, SpiceC differs in two ways. First, to parallelize a loop with operations using MPI I/O, programmers must write code to calculate the starting and ending offset for each thread and explicitly set the offset using the MPI I/O APIs. Programmers also need to take care of synchronization, scheduling, load balancing, etc. Using the SpiceC programming model, programmers just need to insert a few pragmas. Second, Parallel I/O, such as MPI I/O, does not provide any support for speculative parallelization, while SpiceC does.

II. I/O support for transactional memory. Unrestricted transactional memory [19] is a hardware transactional memory technique that has been proposed to support I/O calls in transactions. Transactions usually cannot contain I/O calls because these operations cannot easily be rolled back. Unrestricted transactional memory gives up some concurrency in exchange for gaining the ability to perform I/O calls within transactions by allowing only a single overflowed transaction per application.

III. I/O prefetching. Helper threading has been used in software-guided prefetching to hide I/O latency [120, 23]. To minimize I/O latency, these techniques require timely prefetching. They rely on the profiler or operating system to insert prefetching calls. The helper threading technique in SpiceC is designed to reduce contention on the I/O bus instead of hiding the I/O latency. Therefore, it only requires data residing in main memories (i.e., off-chip memories) instead of caches (i.e., on-chip memories) when they are read. Since main memories have very large capacity nowadays, SpiceC does not require very timely prefetching. Moreover, the helper threading in SpiceC is specially designed for loops with contiguous I/O accesses. Therefore, it does not require any support from a profiler or the operating system.

Chapter 9

Conclusions

9.1 Contributions

This dissertation makes contributions in the area of programming and compiler support for widely-available parallel computing systems. It presents SpiceC, which is a parallel programming interface that simplifies the task of parallel programming through a combination of SpiceC directives and an intuitive computation model. The programming interface provided by SpiecC consists of a set of high-level directives which allow programmers to easily express different forms of parallelism. The computation model of SpiceC offers is based upon software-managed memory isolation and data transfers between threads private spaces and shared space. This model achieves ease of portability of SpiceC across shared and distributed memory platforms. In particular, this dissertation makes the following contributions:

I. Satisfying the demand for the parallelization of various real world applications. Due to the diversity of real world application, they cannot all be parallelized in the same way. Therefore SpiceC programming model can express various forms of parallelism, including DOALL, DOACROSS, and pipelinning parallelism. SpiceC also provides support for dealing with real world applications with the following characteristics. Applications that cannot be statically parallelized but contain large amounts of dynamic parallelism can be handled easily and effectively via speculative parallelization features of SpiceC. For applications that operate on pointer-based dynamic data structures SpiceC provides support for partitioning data structures across threads and then distributing the computation in a partition sensitive fashion. Finally, applications with I/O operations interspersed with computation can be easily parallelized using SpiceC.

II. Meeting the challenges posed by the memory architectures of modern parallel systems. The memory architectures of parallel systems include two categories: shared memory and distributed memory. The SpiceC computation model can handle both kinds of architectures by simply adapting the manner in which data transfers between private spaces and shared space are implemented. The burden of these transfers is on the compiler and runtime; thus, performance portability is achieved without placing burden on the programmer. This thesis demonstrated the implementations of SpiceC for shared-memory multicores with and without cache coherence support, heterogeneous multicores with GPUs, and distributed memory systems.

III. Addressing the practical issues related to the implementation of software-based speculative parallelization. While exploiting dynamic parallelism is highly desirable, programming speculatively parallelized code in an unmanaged language, such as C/C++, is a demanding task for programmers. Although many implementations of transactional memory have been proposed, all of them require significant amount of programming effort, including inserting read/write barriers for each shared read/write, annotating functions called directly and indirectly in a transaction (i.e., a speculatively executed code region), and manually handling the precompiled functions and system calls. With the SpiceC programming model, programmers only need to mark the code regions that need to be executed speculatively. The compiler generates code for speculative execution by automatically inserting STM constructs. The compiler also enables precompiled library functions and system calls to be executed in parallel within transactions.

IV. Overcoming the obstacle to parallelization caused by the dynamic irregularities for GPU computing. This dissertation for the first time demonstrates how data parallel loops whose execution requires the use of speculation, can be accelerated using a GPU. Data parallel loops have been observed to contain dynamic irregularities that cause cross-iteration dependences at runtime, preventing existing techniques from parallelizing the loops on GPUs. A speculative execution framework was developed for GPU computing. It is designed to parallelize loops that may contain cross-iteration dependences caused by the dynamic irregularities. The SpiceC programming model is extended for writing speculative parallel loops for GPUs.

V. Handling the performance bottleneck raised by the network communication on distributed memory systems. The scalability of distributed memory systems can meet the demands of modern applications from important domains (e.g., data mining, machine learning, and computational geometry) that have high processing demands and a *very large memory footprint*. However, since the network communication could be many orders of magnitude slower than CPUs, it greatly limits the program performance on distributed memory systems. This dissertation presents a data communication technique, called proactive communication, which is designed to move the overhead of communication off the critical path. With proactive communication, data are proactively fetched from remote machines to ensure that the data required by the computation are *mostly* available locally when needed. Proactive communication allows local computations to make progress without having to wait due to network communication delays. Data is committed to remote owners in parallel with the computation to increase the throughput.

9.2 Future Direction

I. Extending SpiceC for efficiently utilizing the massive parallel architecture on future manycore processors. This dissertation demonstrates SpiceC on systems consisting of 8–56 processor cores. Researchers from Intel have announced that the architecture of their recent 48-core processor called Single-Chip Cloud Computer can scale to 1,000 cores. Therefore, SpiceC should be able to meet the challenges posed by the massive parallel architectures on future manycore processors. This problem can be tackled from three angles. First, the SpiceC programming model should be extended for extracting parallelism from real applications for manycore processors. Extracting parallelism for this kind of extreme-scale processors is a challenging task, requiring new understanding of parallelism and program behavior. Second, many applications lack sufficient parallelism to keep all the cores busy. An alternative approach is to use the idle cores to execute helper threads that assist the work on the busy cores. The possible role of helper threads (e.g., data prefetching) in SpiceC should be examined on 1000-core processors. Finally, with the massive parallel architecture, it is necessary to to revisit the current implementations of some parallelisms in SpiceC. For example, the current implementations of speculative parallelism require synchronization between commits if the original order of the sequential loop needs to be preserved, which may prevent performance scaling. Besides, the current implementations of speculative parallelism requires some shared data structures or locks for misspeculation detection, which may become performance bottlenecks on 1000-core processors.

II. Optimizing SpiceC for new hardware designs presented on future manycore processors. First, to integrate more cores on a chip, cores in a lot of manycore processors are tightly coupled, sharing a lot of computing resources. For example, in AMD's Bulldozer architecture, two neighboring cores share the instruction fetcher, the instruction decoder, the FPUs, and the L2 cache. In the future, the SpiceC runtime should be designed to reduce the contention on shared resources. Second, 3D integrated circuits have been seen as a promising way to design manycore processors. However, extensive localized heating can occur due to the stacking technology. This will cause memory failures and thus degrade the performance. Compiler and runtime optimizations should be developed for SpiceC programs to alleviate the heat problem.

III.Developing intelligent debugging tools for SpiceC. Using SpiceC, users develop parallel programs by instrumenting sequential programs with parallelization code. Currently there is no efficient tool for debugging this kind of parallel programs. Theoretically, debugging the parallelization code can be done by comparing the runtime behavior (e.g. dynamic data dependences) of sequential code and parallel code. It will have three advantages over traditional concurrent debugging techniques based on thread interleaving. First, it may expose bugs in the parallelization code even if the program outputs are correct. Second, such tool can tell if a bug is caused by parallelization or not. Finally, it reduces the time required by debugging by avoiding exploiting thread interleaving, which has been widely used in traditional concurrent debugging.

Bibliography

[1] Comeau C++ compiler. http://www.comeaucomputing.com/.

[2] DDBJ sequence read archive. http://trace.ddbj.nig.ac.jp/dra/index_e.shtml.

[3] Intel C++ STM compiler, prototype edition. http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/.

[4] Single-chip cloud computer. http://techresearch.intel.com/articles/Tera-Scale/1826.htm.

[5] Space tyrant. http://spacetyrant.com/st.c.

[6] UCI machine learning repository. http://archive.ics.uci.edu/ml/.

[7] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI*, pages 26–37, 2006.

[8] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the GPU. In *ACM SIGGRAPH Asia*, pages 154:1–154:9, 2009.

[9] Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *PLDI*, 1993.

[10] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An extension of the StarSs programming model for platforms with multiple GPUs. In *Euro-Par*, pages 851–862, 2009.

[11] M. Azimi, N. Cherukuri, D. N. Jayasimha, A. Kumar, P. Kundu, S. Park, I. Schoinas, and A. S. Vaidya. Integration challenges and tradeoffs for tera-scale architectures. *Intel Technology Journal*, 11(3):173–184, 2007.

[12] Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. The opentm transactional application programming interface. In *PACT*, pages 376–387, 2007.

[13] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(4):446–559, 1986.

[14] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *ICS*, pages 225–234, 2008.

[15] M Berry, D Chen, P Koss, D Kuck, S Lo, Y Pang, L Pointer, R Roloff, and A Sameh. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *Int. J. Supercomp.*, 3(3):5–40, 1989.

[16] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, pages 72–81, 2008.

[17] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarn, D. Padua, and C. Von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *PPoPP*, pages 48–57, 2006.

[18] Guy E. Blelloch and Phillip B. Gibbons. Efficiently sharing a cache among threads. In *SPAA*, 2004.

[19] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical Report TR-CIS-06-09, University of Pennsylvania, 2006.

[20] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP*, pages 247–258, 2008.

[21] Erik B. Boman, Doruk Bozdağ, Unit Catalyurek, Assefaw H. Gebremedhin, and Fredrik Manne. A scalable parallel graph coloring algorithm for distributed memory computers. In *EURO-PAR*, pages 241–251, 2005.

[22] K. D. Bosschere, W. Luk, X. Martorell, N. Navarro, M. OBoyle, D. Pnevmatikatos, A. Ramirez, P. Sainrat, A. Seznec, P. Stenstrom, and O. Temam. High-performance embedded architecture and compilation roadmap. In *Transactions on HiPEAC*, volume 1, pages 5–29, 2007.

[23] Angela Demke Brown, Todd C. Mowry, and Orran Krieger. Compiler-based I/O prefetching for out-of-core applications. *ACM Trans. Comput. Syst.*, 19:111–170, May 2001.

[24] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *J. Supercomputing*, 2(2):151–169, 1988.

[25] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in olden. In *PPoPP*, pages 29–38, 1995.

[26] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: bulk enforcement of sequential consistency. In *ISCA*, pages 278–289, 2007.

[27] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[28] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, 2005.

[29] Jeffrey S. Chase, Darrell C. Anderson, Andrew J. Gallatin, Alvin R. Lebeck, and Kenneth G. Yocum. Network i/o with trapeze. In *HOTI*, 1999.

[30] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloth, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *SPAA*, 2007.

[31] Andrew A. Chien and William J. Dally. Concurrent aggregates. In *PPoPP*, pages 187–196, 1990.

[32] Wu chun Feng and Shucai Xiao. To GPU synchronize or not GPU synchronize? In *ISCAS*, pages 3801–3804, 2010.

[33] J.W. Chung, H. Chafi, C.C. Minh, A. McDonald, B. Carlstrom, C. Kozyrakis, and K. Olukotun. The common case transactional behavior of multithreaded programs. In *HPCA*, pages 266–277, 2006.

[34] UPC Consortium. UPC language specifications, v1.2. *Lawrence Berkeley National Lab Tech Report LBNL-59208*, 2005.

[35] James H. Curry, Jackson R. Herring, Josip Loncaric, and Steven A. Orszag. Order and disorder in two- and three-dimensional bénard convection. *Journal of Fluid Mechanics*, 147:1–38, 1984.

[36] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE computational science & engineering*, 5(1):46–55, 1998.

[37] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with ordering guarantees. In *ICDE*, pages 424–435, 2004.

[38] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[39] Inderjit S. Dhillon and Dharmendra S. Modha. A data-clustering algorithm on distributed memory multiprocessors. In *Large-Scale Parallel Data Mining*, pages 245–260, 2000.

[40] G. Diamos and S. Yalamanchili. Speculative execution on multi-gpu systems. In *IPDPS*, pages 1–12, 2010.

[41] Dave Dice, Ori Shalev, and Nir Shavit. Transactional lock II. In *Distributed Computing*, pages 194–208, 2006.

[42] Edsger W. Dijkstra. The origin of concurrent programming. chapter Cooperating sequential processes, pages 65–138. 2002.

[43] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI*, pages 223–234, 2007.

[44] Yuri Dotsenko, Cristian Coarfa, and John Mellor-Crummey. A multi-platform co-array fortran compiler. In *PACT*, 2004.

[45] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *SC*, 2006.

[46] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, pages 237–246, 2008.

[47] Pascal Felber, Torvald Riegel, Christof Fetzer, Martin Süβkraut, Ulrich Müller, and Heiko Sturzrehm. Transactifying applications using an open compiler framework. In *TRANSACT*, 2007.

[48] Min Feng, Rajiv Gupta, and Yi Hu. SpiceC: scalable parallelism via implicit copying and explicit commit. In *PPoPP*, pages 69–80, 2011.

[49] Min Feng, Rajiv Gupta, and Iulian Neamtiu. Effective parallelization of loops in the presence of I/O operations. In *PLDI*, pages 487–498, 2012.

[50] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5, 2004.

[51] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS*, pages 151–162, 2006.

[52] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. The MIT Press, 1994.

[53] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA*, 2003.

[54] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI*, pages 14–25, 2006.

[55] John L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33:28–35, July 2000.

[56] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architecture support for lock-free data structures. In *ISCA*, 1993.

[57] P. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, B. Liblit, G. Pike, J. Su, and K. Yelick. Titanium language reference manual. *U.C. Berkeley Tech Report, UCB/EECS-2005-15*, 2005.

[58] Benjamin Hindman and Dan Grossman. Atomicity via source-to-source translation. In *MSPC*, pages 82–91, 2006.

[59] Benoît Hudson, Gary L. Miller, and Todd Phillips. Sparse parallel delaunay mesh refinement. In *SPAA*, pages 339–347, 2007.

[60] Wen-mei Hwu, Shane Ryoo, Sain-Zee Ueng, John H. Kelm, Isaac Gelado, Sam S. Stone, Robert E. Kidd, Sara S. Baghsorkhi, Aqeel A. Mahesri, Stephanie C. Tsao, Nacho Navarro, Steve S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Implicitly parallel programming models for thousand-core microprocessors. In *DAC*, pages 754–759, 2007.

[61] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *ICS*, 1991.

[62] Mahmut Kandemir, Taylan Yemliha, SaiPrashanth Muralidhara, Shekhar Srikantaiah, Mary Jane Irwin, and Yuanrui Zhnag. Cache topology aware computation mapping for multicores. In *PLDI*, 2010.

[63] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.

[64] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The omega library interface guide. Technical report, University of Maryland at College Park, 1995.

[65] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast track: A software system for speculative program optimization. In *CGO*, pages 157–168, 2009.

[66] Ken Kennedy and John Allen, editors. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann, 2001.

[67] Christos Kotselidis, Mohammad Ansari, Kimberly Jarvis, Mikel Lujan, Chris Kirkham, and Ian Watson. DiSTM: A software transactional memory framework for clusters. In *ICPP*, pages 51–58, 2008.

[68] Christos Kotselidis, Mohammad Ansari, Kimberly Jarvis, Mikel Lujan, Chris Kirkham, and Ian Watson. Investigating software transactional memory on clusters. In *IPDPS*, pages 1–6, 2008.

[69] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. Comput.*, 48(9):866–880, 1999.

[70] M. Kulkarni, Martin Burtscher, Rajasekhar Inkulu, Keshav Pingali, and Calin Cascaval. How much parallelism is there in irregular applications? In *PPoPP*, 2009.

[71] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. In *ASPLOS*, pages 233–243, 2008.

[72] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222, 2007.

[73] Milind Kulkarni, Martin Burtscher, Calin Cascaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS*, pages 65–76, 2009.

[74] George Kurian, Jason E. Miller, James Psota, Jonathan Eastep, Jifeng Liu, Jurgen Michel, Lionel C. Kimerling, and Anant Agarwal. Atac: a 1000-core cache-coherent processor with on-chip optical network. In *PACT*, pages 477–488, 2010.

[75] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, page 75, 2004.

[76] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP*, pages 101–110, 2009.

[77] G. C. Lie and E. Clementi. Molecular-dynamics simulation of liquid water with an ab initio flexible water-water interaction potential. *Phys. Rev. A*, 33(4):2679–2693, 1986.

[78] Fa-Hsuan Lin, Fu-Nien Wang, Seppo P. Ahlfors, Matti S. Hämäläinen, and John W. Belliveau. Parallel MRI reconstruction using variance partitioning regularization. *Magnetic Resonance in Medicine*, 58(4):735–744, 2007.

[79] Yi Lin, Bettina Kemme, Marta Patiño Martínez, and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD*, pages 419–430, 2005.

[80] Shaoshan Liu, Christine Eisenbeis, and Jean-Luc Gaudiot. Speculative execution on GPU: An exploratory study. In *ICPP*, pages 453–461, 2010.

[81] Shaoshan Liu, Christine Eisenbeis, and Jean-Luc Gaudiot. Value prediction and speculative execution on GPU. *IJPP*, 39(5):533–552, 2011.

[82] Victor Luchangco, Lawrence Crowl, Yossi Lev, Dan Nussbaum, and Mark Moir. Integrating transactional memory into C++. In *TRANSACT*, 2007.

[83] Mary E. Mace. *Memory storage patterns in parallel processing.* Kluwer Academic Publishers, 1986.

[84] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP*, pages 198–208, 2006.

[85] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. In *ISCA*, pages 53–65, 2006.

[86] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *PLDI*, pages 166–176, 2009.

[87] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *PLDI*, 2009.

[88] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[89] M. Mendez-Lojo, Donald Nguyen, Dimitrios Prountzos, Xin Sui, M. Amber Hassaan, Milind Kulkarni, Martin Burtscher, and Keshav Pingali. Structure-drive optimization for amorphous data-parallel programs. In *PPoPP*, 2010.

[90] Miloš Milovanović, Roger Ferrer, Vladimir Gajinov, Osman S. Unsal, Adrian Cristal, Eduard Ayguadé, and Mateo Valero. Multithreaded software transactional memory and OpenMP. In *MEDEA*, pages 81–88, 2007.

[91] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, pages 35–46, 2008.

[92] Andreas Müller and Roland Rühl. Extendings high performance fortran for the support of unstructured computations. In *ICS*, pages 127–136, 1995.

[93] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for c/c++. In *OOPSLA*, pages 195–212, 2008.

[94] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[95] Victor Olman, Fenglou Mao, Hongwei Wu, and Ying Xu. Parallel clustering algorithm for large data sets with applications in bioinformatics. *TCBB*, 6(2):344–352, 2009.

[96] Marek Olszewski, Jeremy Cutler, and J. Gregory Steffan. Judostm: A dynamic binary-rewriting approach to software transactional memory. In *PACT*, pages 365–375, 2007.

[97] David Patterson. The trouble with multi-core. *IEEE Spectrum*, 47(7):28–32, 2010.

[98] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178, 2009.

[99] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *PLDI*, pages 12–25, 2011.

[100] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput.*, 36:1425–1439, 1987.

[101] Prakash Prabhu, G. Ramalingam, and Kapil Vaswani. Safe programmable speculative parallelism. In *PLDI*, pages 50–61, 2010.

[102] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. LOCKSMITH: Practical static race detection for C. *TOPLAS*, 33(1):3:1–3:55, January 2011.

[103] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Concurrency*, 4(2):63–79, 1996.

[104] Dan Quinlan. Rose: Compiler support for object-oriented framework. In *CPC*, 2000.

[105] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative parallelization using software multi-threaded transactions. In *ASPLOS*, pages 65–76, 2010.

[106] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. Decoupled software pipelining with the synchronization array. In *PACT*, 2004.

[107] Lawrence Rauchwerger and David Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *PLDI*, pages 218–232, 1995.

[108] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor*. O'Reilly Media, 2007.

[109] Michael F. Ringenburg and Dan Grossman. Atomcaml: first-class atomicity via rollback. In *ICFP*, pages 92–104, 2005.

[110] Anne Rogers and Keshav Pingali. Process decomposition through locality of reference. In *PLDI*, pages 69–80, 1989.

[111] Amitabha Roy, Steven Hand, and Tim Harris. Hybrid binary rewriting for memory access instrumentation. In *VEE*, pages 227–238, 2011.

[112] Barbara G. Ryder and Ben Wiedermann. Language design and analyzability: a retrospective. *SP&E*, 42(1):3–18, 2012.

[113] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP*, pages 73–82, 2008.

[114] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP*, pages 187–197, 2006.

[115] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: a low overhead, software-only approach for supporting fine-grain shared memory. In *ASPLOS*, pages 174–185, 1996.

[116] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *ASPLOS*, pages 297–306, 1994.

[117] Michael Scott, Michael F. Spear, Luke Dalessandro, and Virendra J. Marathe. Delaunay triangulation with transactions and barriers. In *IISWC*, 2007.

[118] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[119] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 2008.

[120] Seung Woo Son, Sai Prashanth Muralidhara, Ozcan Ozturk, Mahmut Kandemir, Ibrahim Kolcu, and Mustafa Karakoy. Profiler and compiler assisted adaptive I/O prefetching for shared storage caches. In *PACT*, pages 112–121, 2008.

[121] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *PPoPP*, 2009.

[122] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, 2010.

[123] Chunqiang Tang, DeQing Chen, Sandhya Dwarkadas, and Michael L. Scott. Integrating remote invocation and distributed shared state. In *IPDPS*, 2004.

[124] R. Thakur, W. Gropp, and E. Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *FRONTIERS*, pages 180–187, 1996.

[125] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *FRONTIERS*, pages 182–191, 1999.

[126] Chen Tian, Min Feng, and Rajiv Gupta. Copy or discard execution model for speculative parallelization on multicores. In *MICRO*, pages 330–341, 2008.

[127] Chen Tian, Min Feng, and Rajiv Gupta. Copy or discard execution model for speculative parallelization on multicores. In *MICRO*, pages 330–341, 2008.

[128] Chen Tian, Min Feng, and Rajiv Gupta. Speculative parallelization using state separation and multiple value prediction. In *ISMM*, 2010.

[129] Chen Tian, Min Feng, and Rajiv Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *PLDI*, pages 62–73, 2010.

[130] Chen Tian, Changhui Lin, Min Feng, and Rajiv Gupta. Enhanced speculative parallelization via incremental recovery. In *PPoPP*, pages 189–200, 2011.

[131] Paul E. Utgoff, Neil C. Berkman, and Jeffery A. Clouse. Decision tree induction based on efficient tree restructuring. *Mach. Learn.*, 29(1):5–44, October 1997.

[132] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative decoupled software pipelining. In *PACT*, pages 49–59, 2007.

[133] L. Van Put, D. Chanet, B. De Bus, B. De Sutler, and K. De Bosschere. DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In *ISSPIT*, pages 7–12, 2006.

[134] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO*, pages 34–48, 2007.

[135] Cheng Wang, Victor Ying, and Youfeng Wu. Supporting legacy binary code in a software transaction compiler with dynamic binary translation and optimization. In *CC*, pages 291–306, 2008.

[136] J. D. Watts and M. Dupuis. Towards efficient parallel computation of correlated wave functions. Implementation of two-electron integral transformation on the LCAP parallel supercomputer. *IBM Technical Report KGN-100*, 1987.

[137] Shih wei Liao, Zhaohui Du, Gansha Wu, and Guei-Yuan Lueh. Data and computation transformations for brook streaming applications on multiprocessors. In *CGO*, 2006.

[138] Tien-Hsiung Weng, Ruey-Kuen Perng, and Barbara Chapman. Openmp implementation of SPICE3 circuit simulator. *Int. J. Parallel Program.*, 35(5):493–505, 2007.

[139] Shuqing Wu and Bettina Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *ICDE*, pages 422–433, 2005.

[140] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A GPGPU compiler for memory optimization and parallelism management. In *PLDI*, pages 86–97, 2010.

[141] Mitsuo Yokokawa, Fumiyoshi Shoji, Atsuya Uno, Motoyoshi Kurokawa, and Tadashi Watanabe. The K computer: Japanese next-generation supercomputer development project. In *ISLPED*, pages 371–372, 2011.

[142] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *PADS*, 1998.

[143] Daniel R. Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, 18:821–829, 2008.

[144] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, and Xipeng Shen. Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In *ICS*, pages 115–126, 2010.

[145] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In *ASPLOS*, pages 369–380, 2011.

[146] Jin Zhou and Brian Demsky. Bamboo: a data-centric, object-oriented approach to many-core software. In *PLDI*, 2010.