UNIVERSITY OF CALIFORNIA
RIVERSIDE

Speculative Parallelization on Multicore Processors

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

Doctor of Philosophy

in

Computer Science

by

Chen Tian

June 2010

Dissertation Committee:

Dr. Rajiv Gupta, Chairperson
Dr. Laxmi Bhuyan
Dr. Iulian Neamtiu

The Dissertation of Chen Tian is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I own my gratitude to all those people who have made this dissertation possible and because of whom my graduation experience has been one that I will cherish forever.

My deepest gratitude is to Dr. Rajiv Gupta, who leads me to today's achievements. I have been amazingly fortunate to have an advisor who gave me the inspiration to solve different research problems and at the same time the guidance to recover when my steps faltered. He believed in me and encouraged me during times when I did not believed in myself. Without him, this dissertation never would have been done.

I would like to thank my other dissertation committee members, Dr. Laxmi Bhuyan and Dr. Iulian Neamtiu for taking their time to help me improve this dissertation.

I would like to express my gratitude to all the members of my research group including Min Feng, Dennis Jeffrey, Sriraman Tallam, Vijay Nagarajan, Xiangyu Zhang, Bengu Li, Changhui Lin for helping me in many ways during these years.

I would also like to thank all my teachers I have had throughout my life. I am where I am today because of them.

Finally, I would like to thank my family, particularly my father Quanlai Tian, my mother Qingfang Li and my wife Yejuan Long, for their unconditional support during these years.

This dissertation is dedicated to my family.

ABSTRACT OF THE DISSERTATION

Speculative Parallelization on Multicore Processors

by

Chen Tian

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2010
Dr. Rajiv Gupta, Chairperson

With the advent of multicore processors, extracting thread level parallelism from a sequential program has become crucial for improving performance. However, many sequential programs cannot be easily parallelized due to the presence of dependences. To solve this problem this dissertation presents a thread-based execution model, called *Copy-or-Discard* (CorD), that supports speculative parallelization. In CorD, the state of speculative threads is maintained separately from the non-speculative computation state. If speculation is successful, the results of the speculative computation are committed by copying them into the non-speculative state. If a misspeculation is detected, no costly recovery mechanisms are needed as the speculative state can be simply discarded.

To illustrate the applicability of CorD, this dissertation first shows how to apply it to streaming applications. Optimizations are proposed to reduce data copying overhead. A lightweight scheme based on version comparison is also presented to detect misspeculations. It is observed that when misspeculation rate becomes high, the benefits of parallelism are usually nullified. To address this problem two techniques, *Multiple Speculations* and *Incremental Recovery*, are proposed. The first technique creates multiple versions of

speculatively-executed code using different value predictions. If any one of these versions is found to be correct, the speculation is successful. The second technique focuses on reducing misspeculation cost. Instead of discarding all results, it allows saving and reuse of the results that are not affected by the variables that cause the misspeculation.

Finally, this dissertation shows the applicability of CorD in the presence of dynamic data structures. Such data structures pose many new challenges. The copying of data structures from non-speculative to speculative state is expensive due to the large sizes of data structures. The copying of updated data structures from speculative state to non-speculative state are complex due to the changes in the shape of dynamic data structures. In addition, translating pointers internal to dynamic data structures between their non-speculative and speculative memory addresses has to be addressed. This dissertation proposes an augmented design for the representation of dynamic data structures such that all of the above operations are performed efficiently.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Extracting thread level parallelism from sequential programs is very important for improving their performance on widely available multicore processors. Unfortunately, manual code parallelization by programmers is a time-consuming and error-prone process. Consequently, compiler based automatic parallelization techniques have drawn much attention of researchers. Many earlier works on DOALL parallelism [35, 42] focus on identifying loops without cross-iteration dependences. However, the main loop in most sequential programs cannot be easily parallelized by a compiler due to the presence of cross-iteration dependences.

To alleviate this problem the speculative parallelization technique has been proposed. The key idea of this technique is to make the assumption that there is no dependence between the two sequential regions of code (e.g., loop iterations) and execute them in parallel. This is based on the observation that in many programs the dependences that prevent parallelization may not frequently occur at runtime. Consequently, there are three steps to speculatively parallelizing a sequential application. The first step is to identify a

loop that has infrequent cross-iteration dependences. This process involves static analysis and dependence profiling. Programmers can also explicitly and optimistically identify loop level parallelism by annotating the program. The second step is to create multiple parallel threads or processes to execute the speculatively parallelizable loop. This work is usually carried out by the compiler. The last step is to develop a thread-level speculation (TLS) runtime system which can detect misspeculations (i.e., violations of these assumptions by detecting if dependences that were assumed to be absent manifest at runtime) and handle the results of speculative computations. Among these three steps, developing an efficient and reliable TLS system is the most important and complex.

Although considerable research work has been carried out on developing TLS systems [17, 72, 49, 27, 24, 36, 74, 60], most of the work is hardware based and not ready for use on the multicore systems available today. This is due to the architectural redesigns [17, 72, 49] requiring non-trivial hardware changes for detecting misspeculations and handling speculatively computed results (e.g., special buffers [27, 60], versioning cache [24], or versioning memory [23]), which have not been incorporated in commercial multicore processors. Therefore an attractive alternative avenue of optimistically extracting parallelism from programs is based upon a purely software realization of TLS. This is the approach that is developed in this dissertation.

While developing an efficient software implementation of TLS is challenging, the benefits of this approach are clear as it can be applied to existing widely available multicore systems. Recently, software based TLS techniques have been proposed [15, 34, 38, 40, 41]. While some of them [38, 40, 41] require the programmer to provide recovery code, others [15, 34] are based upon realization of state separation with no programmer help.

Ding *et al.* [15, 34] achieve state separation by creating separate processes for a non-speculative and speculative computations – since each process has its own address space, state separation is achieved. However, the communication among processes is too expansive due to the involvement of the operation system. Moreover, their approach cannot handle applications that make intensive use of dynamic data structures because such data structures are allocated dynamically and communicating a large amount of data between processes at runtime is expensive.

1.1 Dissertation Overview

This dissertation presents a software realization of TLS which overcomes the drawbacks of prior techniques and is applicable to a wide range of applications. The key issues addressed in this work are briefly described next.

1.1.1 Speculative Execution Model and its Software Implementation

To parallelize a sequential program, multiple processes or threads need to be created. Processes have their own independent address spaces and communication among them is expensive due to the involvement of the operating system; this is not desirable for a parallelized sequential program. In contrast, all threads that belong to the same process share same memory space and thus they can communicate with each other efficiently. Therefore, the model proposed in this dissertation is based upon threads.

To support speculation, an execution model is developed such that the misspeculation can be detected accurately and speculatively computed results can be handled appropriately and efficiently. State separation is achieved by this model by providing each

speculative thread with its own space to store speculatively computed results. The results are either copied back to non-speculative state or discarded depending upon the outcome of misspeculation detection step. Due to this feature, the model is named *Copy or Discard* (CorD). The software implementation of CorD is developed. In particular, thread creation, task assignment, state separation, misspeculation detection, and recovery mechanisms are all developed.

1.1.2 Profile-Guided Speculative Parallelization

It has been observed that optimistic parallelism exists in many sequential applications [6]. Specifically, cross-iteration dependences in sequential loops may not frequently manifest at runtime if they are caused by code appearing along a cold path. This is particularly true for some streaming applications which usually use a loop to handle the input data piece by piece. In such applications, updates that result in cross-iteration dependences are infrequent. Hence, the loop iterations can be executed in parallel as long as such updates do not manifest themselves. Table 1.1 presents the frequency of cross-iteration dependences in computations of sequential loops for a subset of SPEC2000 programs [31] during their execution on training inputs. One can see that the frequency is extremely small. This reveals that the iterations of the main loop in these programs can be executed in parallel most of the time.

Program	Frequency of cross-iteration dependences
`197.parser`	0.1%
`130.li`	0.5%
`256.bzip2`	0.9%
`255.vortex`	0.2%

Table 1.1: Frequency Of Cross-iteration Dependences.

Unfortunately, such optimistic parallelism cannot be exploited by a compiler that performs parallelization using static dependence analysis. The conservative nature of static analysis precludes parallelization. As a result, the opportunity of exploiting parallelism is missed for such applications.

To address this problem, this dissertation presents a profile-guided speculative parallelization algorithm. In particular, dynamic dependence profiling is performed when an application is executed with a small input. The result is provided to the compiler so that streaming applications can be speculatively parallelized. During the code transformation, CorD model is employed to enable the presence of ignored dependences to be appropriately detected and dealt with.

1.1.3 Multiple Speculations for Handling High Misspeculation Rate

A misspeculation occurs if a cross-iteration dependence arises at runtime. Speculative parallelization takes advantage of the fact that such situations rarely occur. However, on certain types of inputs an application may not exhibit this property. Consequently, high misspeculation rate wipes out the benefits of speculative parallelization.

To address this problem, the frequent cross-iteration dependences must be resolved. A typical solution is to use value prediction. Specifically, the values of live-in variables (i.e., variables involved in cross-iteration dependences) can be predicted in a later iteration before they are produced by an earlier iteration. If the prediction is correct, these two iterations can still be executed in parallel. To increase the accuracy of predictions, the predictions can be performed in a path sensitive manner, i.e. the values of live-ins can be predicted in different ways for different paths. In this dissertation, by assigning different

prediction methods to different control flow paths, high prediction accuracy is achieved.

Program	One Path	Two Paths	Three Paths
`dry`	80%	87%	90%
`fldry`	81%	87%	89%
`llu`	71%	83%	92%
`mechcall`	66%	90%	100%
`objinst`	60%	85%	100%

Table 1.2: Correct-Value Coverage On Different Number Of Paths.

Table 1.2 shows how the overall prediction accuracy increases as increasing number of paths are considered during value prediction by the path sensitive approach developed in this dissertation. The data presented is for programs taken from the LLVM benchmark test suite [30]. From this table, it can be seen that predicting values on three different paths is very likely to cover the correct values of live-in variables. For `mechcall` and `objinst`, only three paths are taken at runtime and the prediction method on each path is 100% accurate. Thus, the correct values of the live-in variables can be fully covered. Note that the live-in variables in these loops cause definite cross-iteration dependences, and hence the loops cannot be speculatively parallelized.

This dissertation presents a technique called *Multiple Speculations* that exploits the above-mentioned observation to deal with the high misspeculation rate caused by frequent cross-iteration dependences. In particular, this technique creates multiple speculative versions of the same loop iteration by using different predictions along different paths. It allows the parallelism between consecutive iterations to be exploited even by the presence of frequent cross-iteration dependences.

1.1.4 Incremental Recovery for Reducing the Cost of Misspeculation

If a misspeculation occurs, results produced in the speculative computation are considered incorrect and have to be recomputed. However, a misspeculation may result by reading a wrong value of only a single *live-in* variable. This wrong value may not necessarily impact the entire computation. In other words, discarding all of the speculatively stored results is wasteful. Table 1.3 shows the fraction of the speculatively computed results that are actually correct even though a misspeculation has occurred.

Program	Fraction of correct results
`197.parser`	95%
`130.li`	55%
`256.bzip2`	52%
`255.vortex`	65%

Table 1.3: Fraction Of Correct Results Upon a Misspeculation.

As one can see, more than 50% of store instructions on an average store the correct values when a misspeculation occurs in these programs, because these values are not computed by using misspeculated live-in variables. Recomputing them is completely unnecessary during the misspeculation recovery. Therefore, this dissertation proposes a technique called *Incremental Recovery* that enables the correct speculatively computed results to be reused when a misspeculation occurs. In particular, it first identifies the live-in variables that have caused a misspeculation. Then it identifies the results that were computed using these live-in variables during speculative execution so that only these results are computed during recovery. This technique greatly reduces the cost of misspeculation recovery and can yield speedups from speculatively parallelizing a sequential program even when the misspeculation rate is high.

1.1.5 Speculative Parallelization in the Presence of Dynamic Data Structures

Speculative parallelization technique relies on the runtime bookkeeping of memory accesses. The bookkeeping process is the main source of the overhead during a parallelized program's execution. In most streaming applications, the parallelism benefits outweigh such overhead. However, for applications that make intensive use of linked dynamic data structures, this is not the case. The reason is that the sizes of the dynamic data structures are much larger than the sizes of data structures used in streaming applications. Table 1.4 shows this data for 5 applications from LLVM benchmarks [30] that use dynamic data structures.

Programs	Node Number	Memory Consumption
`MST`	0.1 Million	28MB
`Power`	18.2 Thousand	6.8MB
`Patricia`	18.8 Thousand	4.5MB
`Treesort`	4.9 Million	59 MB
`Hash`	3.5 Million	45MB

Table 1.4: Sizes Of Dynamic Data Structures.

From this table, one can see that each of the program creates thousands or even millions of nodes at runtime. The overhead of bookkeeping for accesses to these nodes easily nullifies the parallelism benefits. To address this problem, this dissertation proposes three techniques, *Copy-on-write Scheme*, *Heap Prefix*, and *Double Pointers*. They effectively reduce the overhead and address the challenges posed by the presence of dynamic data structures. As a result, significant speedups can still be obtained by applying speculative parallelization.

1.2 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 describes the CorD model and its software implementation. Chapter 3 presents the parallelization algorithm and code transformation for streaming applications. Chapter 4 describes the multiple speculations technique. Chapter 5 presents the incremental recovery technique. In chapter 6, three new techniques are presented to address the challenges of applying CorD in the presence of dynamic data structures. Related work is given in Chapter 7 and the conclusions of the dissertation are summarized in Chapter 8.

Chapter 2

Copy Or Discard Execution Model

Speculative parallelization allows the parallelism in a sequential program to be aggressively exploited. The idea of this technique can be explained as follows. Let P denote a sequential program that performs a computation which can be a process of data processing or an algorithm implementation. Consider a pair of subcomputations C and C' in P such that the execution of C precedes the execution of C' during sequential execution (e.g., C and C' may represent consecutive iterations of a loop). Thus, the results computed during the execution of C are available during the execution of C'. The goal of *speculative execution* is to relax the strict ordering imposed on the execution of C and C' by speculatively executing C' while C is still executing. During the speculative execution of C', if a data value is read prematurely (i.e., it is read before it has been computed by C), then *misspeculation* occurs and thus the results computed during speculative execution of C' must be discarded and C' must be executed again. On the other hand, if misspeculation does not occur, the execution time of the program is reduced due to parallel execution of C and C'. Of course, speculative execution is only beneficial if the misspeculation occurs *infrequently*.

Opportunities for speculative execution arise because the dependences from C to C' may arise from infrequently executed code or even if they do arise, they may be deemed harmless (e.g., dependences may arise due to silent stores). The above approach naturally extends to multiple levels of speculation. Given a series of dependent computations $C_1 \rightarrow C_2 \rightarrow \ldots C_n$, while C_1 executes non-speculatively, C_2 through C_n can be speculatively execute in parallel with C_1 on additional cores.

This chapter proposes an execution model, *Copy or discard* (CorD), that supports speculative execution as illustrated above. The overview of this model is described in Section 2.1 followed by the implementation details which are presented in Section 2.2.

2.1 Overview

2.1.1 Thread Execution Model

CorD is a thread based model. It has one main thread and multiple speculative parallel threads. The advantage of using threads is that the entire memory space is accessible for all threads within the process. Thus, the synchronizations and communications in CorD are easier and more light-weight than those in process based model.

Given a sequential computation that can be divided into non-parallelizable regions and parallelizable regions as shown in Figure 2.1(a), the main thread and multiple parallel threads execute these regions in the following way. The main thread executes all non-parallelizable regions. When it reaches a parallelizable region, it spawns a number of parallel threads and assigns a piece of work taken from the parallelizable region to each of them. After task assignments, it simply waits for the parallel threads to finish their assigned tasks.

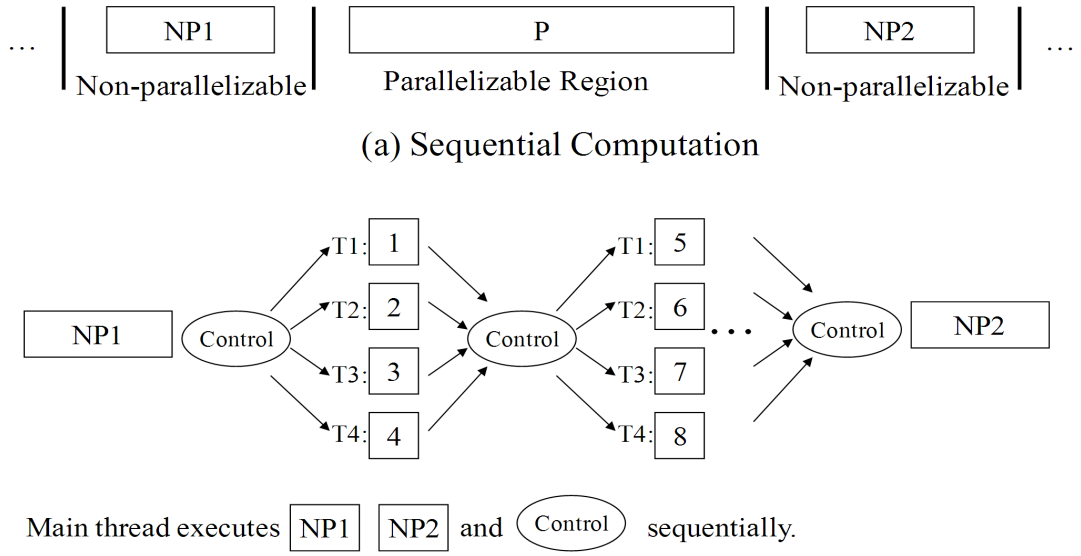


Figure 2.1: Sequential Execution And Thread Based Parallel Execution.

When a parallel thread finishes a task, it notifies the main thread, so that the main thread can detect misspeculations and redo the same task or assign a new task to it. This pattern continues until the entire non-parallelizable region is executed. After that, if the next region is non-parallelizable, the main thread executes it. Otherwise, all existing parallel threads are reused to perform the computation in parallel like before. These parallel threads will exit when the last non-parallelizable region has finished executing. Figure 2.1(b) illustrates this execution pattern. Control denotes the activity of the main thread during the speculative execution. The activity includes spawning new speculative threads, assigning tasks, detecting misspeculations and committing results.

2.1.2 State Separation

One of the key characteristics of CorD is *state separation* according to which the non-speculative state of the program is maintained separately from the speculative state of the computation. While the main thread maintains the non-speculative state of the computation, each parallel thread has to have a speculative state to store its own computation results because the computation is performed using speculatively-read operand values from non-speculative state. State separation guarantees thread isolation, that is, the execution of each thread, main or parallel, is isolated from execution of all other threads. Thus, if execution of one thread results in misspeculation, it does not necessarily force the reexecution of other threads.

In CorD, the entire memory space of a speculatively parallel execution is divided into three disjoint partitions < D, P, C> such that each partition contains a distinct type of program state (see Figure 2.2).

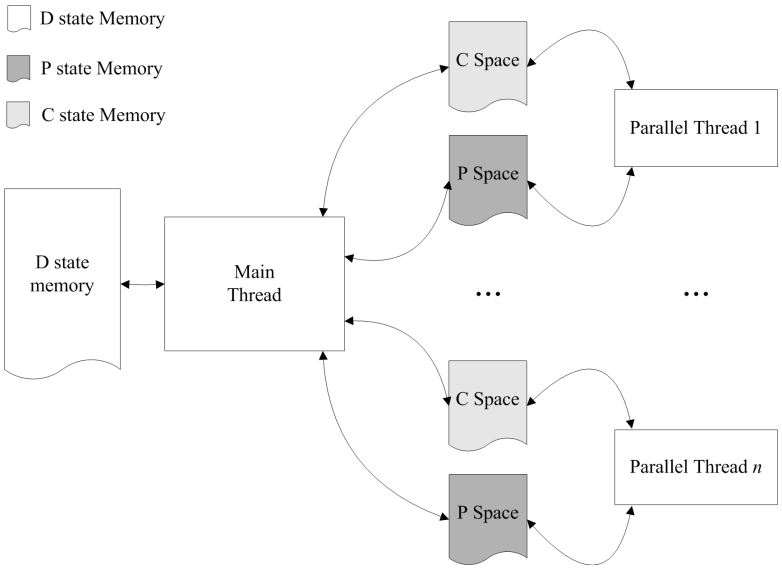


Figure 2.2: Maintaining Memory State.

Non-speculative State. D memory is the part of the address space that reflects the non-speculative state of the computation. Only the main computation thread Mt performs updates of D. If the program is executed sequentially, Mt performs the entire computation using D. If parallel threads are used, then Mt is responsible for updating D according to the results produced by the parallel threads.

Parallel or Speculative State. P memory is the part of the address space that reflects the parallel computation state, that is, the state of the parallel threads Ts created by Mt to boost performance. Since parallel threads perform speculative computations, speculative state that exists is at all times contained in P memory. The results produced by the parallel threads are communicated to Mt that then performs updates of D. Note that P memory is further divided into a number of disjointed spaces. Each parallel thread uses one of these spaces as its own P Space. To avoid execution interference, a parallel thread is not allowed to access any other parallel thread's P Space.

Coordinating State. C memory is the part of the address space that contains the coordinating state of the computation. Since the execution of Mt is isolated from the execution of parallel threads Ts, mechanisms are needed via which the threads can coordinate their actions. The coordinating state provides memory where all state needed for coordinating actions (e.g., actions in *control* component) is maintained. Similar to P memory, C memory is further divided into a number of C spaces, each corresponding to one parallel thread.

2.2 Implementation Of CorD

There are three key aspects of CorD model, namely *Thread Interactions*, *Memory State Separation*, and *Misspeculation Check and Recovery*. This section describes their implementations one by one.

2.2.1 Thread Interactions

Spawning Speculative Threads. The main thread executes non-speculative regions. When it reaches a parallelizable region for the first time, it needs to spawn the speculative parallel threads. This can be implemented by calling thread creation functions such as *pthread_create()*. A task is then assigned and performed speculatively by each parallel thread. In particular, the computational statements in a task are encapsulated into a function, whose pointer is passed to the thread creation function as a parameter. The input data is provided by the main thread. If the main thread reaches another parallel region, spawning new threads is not necessary because the speculative threads created earlier can be reused. Only when the entire execution reaches the end, all speculative threads exit.

Copying Operations. When a parallel thread starts a speculative computation, it requires the input data and the values of context variables (i.e., variables that are used before being defined) to be present in its P space. When a parallel thread finishes a speculative computation and the results are validated, all produced data need to be move back to D space. Therefore, a mechanism is needed to transfer data between D space and P space. This mechanism is implemented through copying operation in CorD. In particular, *copy-in* refers to the operation that copies data values from D space to a P space and *copy-out* refers

to the operation that copies data values from a P space to D space.

When copying operations are performed to transfer data, certain information has to be maintained for consistency. In particular, when the data value is copied-in, both its source and destination address have to be remembered. Thus, when this value is changed by a parallel thread, copy-out operation can update the correct memory location in D. To achieve this, the mapping table is required for copying operation. It is allocated by the main thread for every parallel thread. As shown below, an entry in the mapping table contains five fields.

D_Addr	P_Addr	Size	Version	WriteFlag

The D_Addr and P_Addr fields provide the corresponding addresses of a variable in the D state and P state memory while Size is the size of the variable. Version is the version number of the variable when the value is copied from D state to P state memory. It is used during misspeculation detection which will be described later. The WriteFlag is initialized to *false* when the value is initially copied from D state to P state memory. However, if the parallel thread modifies the value contained in P_Addr, the WriteFlag is set to *true* by the parallel thread. During the result-committing stage, this flag is examined to determine which variables need to be copied out.

2.2.2 Implementing State Separation

Since the D and P memories are used by all threads, they must support **stack**, **global**, and **heap** sections and each of these sections must provide state separation.

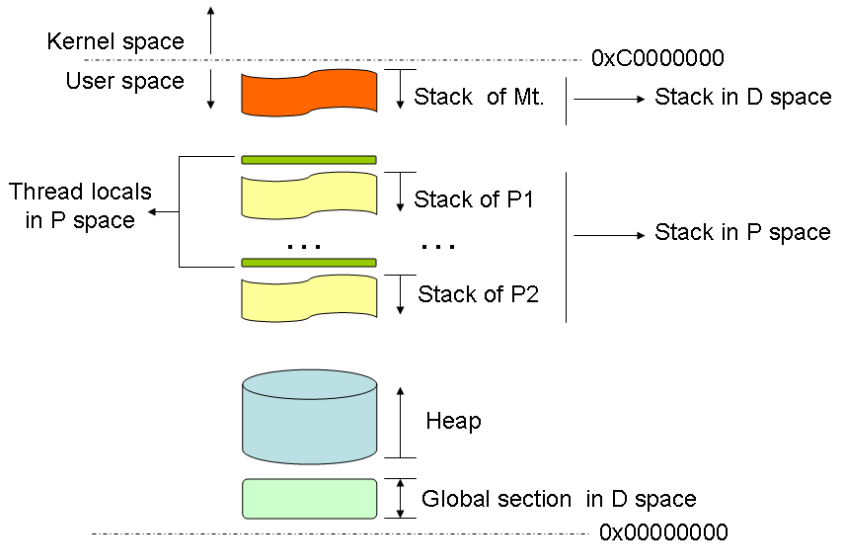


Figure 2.3: Separation Of Stack And Global Section.

Stack Separation. For a sequential program, all local variables are allocated on the stack and accessed through a stack pointer. When multiple threads co-exist, each of them has its own stack and stack pointer as shown in Figure 2.3. If POSIX threads on Linux are used, this is automatically achieved by the pthread library and OS. The default size of the stack allocated for each thread is 10M.

To avoid stack overflow, a safety check needs to be performed when a stack grows. This is also automatically achieved by the OS. In particular, when the size of a stack exceeds its limit, the OS sends a signal to the running process. In our execution model, this signal is captured by the process. Within the signal handler, the size of each parallel thread's stack can be reset (e.g., by calling the functions *pthread_attr_setstacksize()*).

Global Section Separation. Variables stored in the global section can only be used by the main thread. Parallel threads need to maintain speculative copies of these variables

to achieve state separation. However, creating a copy of the entire global section for each parallel thread is wasteful because parallel threads do not run any more after they complete the speculative computations. Therefore, only one global section is maintained which is used by the main thread. For every global variable used by a parallel thread, a local copy is created in the thread local storage above the parallel thread's stack as shown in Figure 2.3. As a result, the overflow check is not needed on the global section because its size is not increased at runtime.

Heap Separation. The heap is used to support dynamic memory allocation in a sequential program. The allocation is performed through a memory allocator such as *malloc* library call. In the thread based execution model, however, a separate heap does not have to be maintained for each thread. This is because it is very hard to predict how much of heap space a thread will use at runtime. Thus, only one heap is used by the whole process. Logical separation can be achieved as follows. When a heap chunk is allocated to the main thread, it is considered as D space heap. If it is allocated to a parallel thread, it is considered as P space heap. The safety check of heap access is simply done by checking the return value of the memory allocator. Specifically, if a *malloc* call fails in a parallel thread's execution, the parallel thread will free all the memory resources and are terminated.

Besides the D state and P state that support the execution of different threads, a buffer is also allocated for each parallel thread to coordinate the execution of the parallel thread and the main thread. This buffer is essentially the C space for a parallel thread. If a variable (stack, global or heap) maintained by the main thread in D space is used by a parallel thread, a local copy of the variable is created in the corresponding P space. The

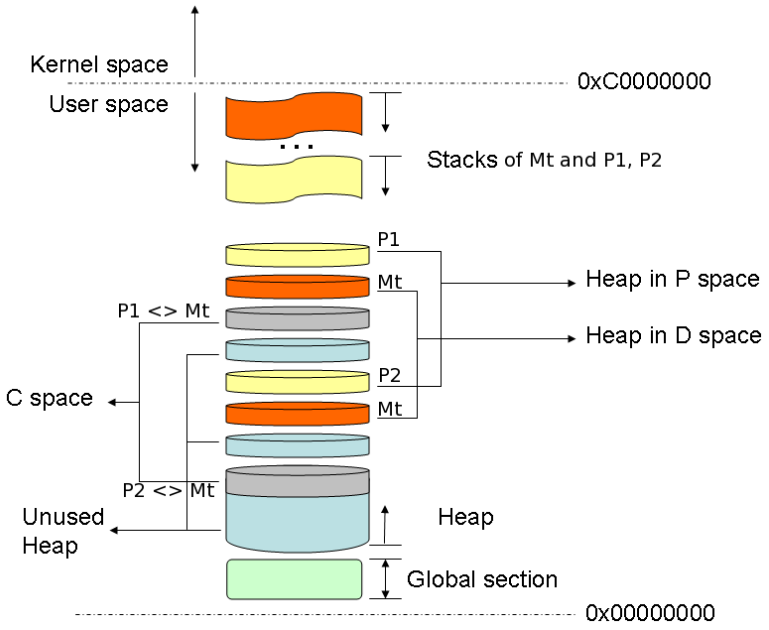


Figure 2.4: Separation Of Heap.

mapping information of this variable (i.e., its D space address and its P space address) needs to be stored in the buffer. C state is essentially the collection of all such buffers. In the implementation, the main thread allocates them by calling the *malloc* function. Thus, they are also on the heap at runtime. Each of them is deallocated when the corresponding parallel thread finishes its execution. Note that although message buffers can be implemented through the `pipe` function call, and thus allocated by the OS in the kernel space (not shown in the figure), they still conceptually belong to the C space in the execution model.

Figure 2.4 illustrates how the heap of a process is used in our execution model. The figure shows that six heap chunks have been allocated. Two are allocated by the main thread, and thus considered in D state. Two chunks requested by parallel threads P1 and P2 respectively are considered in P state. Essentially, they are the duplicate copies of some D state heap chunk in the speculative state. Another two are allocated for coordinating

the main thread and parallel threads, so they are logically in C state. The rest of heap is unused. The location of each memory chunk in the heap is decided by the memory allocator at runtime.

2.2.3 Misspeculation Detection And Recovery

A misspeculation occurs when a speculative computation reads a data value before it is produced by an earlier computation. The misspeculation check is performed by the main thread in sequential order. In particular, the main thread performs the misspeculation check for a parallel thread performed the earliest task among all tasks being currently performed by all the parallel threads.

Misspeculations detection algorithm considers each data value speculatively read by the committing parallel thread. The main thread examines if any such value is updated during the execution of an earlier committed task to detect misspeculation. Thus, a *global version* of each variable in D state memory that is potentially read and written by parallel threads needs to be maintained. This version is incremented every time the value of the variable in D state memory is modified during the committing of results produced by parallel threads. For each variable in D state memory, if it is copied into a parallel thread's P space, its global version is also copied into the *version* field of the corresponding entry in the mapping table.

When a parallel thread t informs the main thread that it has completed a speculative task, the main thread performs the misspeculation check by consulting the *mapping table* and accordingly taking the actions shown in Figure 2.5. The main thread compares the current version numbers of variables with the version numbers of the variables in the

```
1:  foreach variable v that has an entry e in the mapping table {
2:    if (e.version != v.global_version){
3:      discard all results and ask t to reperform the task;
4:      return SpeculationFail;
5:    }
6:  }
7:  foreach variable v that has an entry e in the mapping table {
8:    if (e.WriteFlag){
9:      copy-out v;
10:     v.global_version++;
11:   }
12: }
13: return SpeculationSuccess;
```

Figure 2.5: Misspeculation Detection - Algorithm 1.

mapping table. If a version number does not match, then the main thread concludes that misspeculation has occurred (line 1-5). It discards the results and ask the parallel thread to reperform the task. If all version numbers match, then speculation is successful. Thus, the main thread commits the results by *copying* the values of variables for which the WriteFlag is true from P state memory to D state memory. The global version of the copied variable is also incremented by one (line 6-12). Note that if the WriteFlag is not true, then there is no need to copy back the result as the variable's value is unchanged.

Note that the results produced by speculative threads are also committed in sequential order. If a misspeculation occurs for a thread, the main thread waits for this thread to finish the non-speculative reexecution before committing the correct results to D space. Next, the main thread moves to the misspeculation check for the next speculative thread. This is important for ensuring that the results of the parallelized program are consistent with the results of the sequential program.

2.3 Summary

This chapter presented a runtime execution model, CorD, to support speculative parallelization. The model requires one main thread that executes non-parallelizable regions and multiple parallel threads that execute parallelizable regions. The entire memory space is divided into three disjoint partitions such that the execution of each thread is isolated. The implementation of the key operations of CorD, namely thread interactions, state separation, and misspeculation detection were also described in this chapter.

Chapter 3

Speculative Parallelization Of Streaming Applications

A streaming application refers to an application that performs various computations such as parsing and compression on an input to produce the output. Since the input is usually very large, such applications employ a loop to process the input piece by piece. Figure 3.1 shows the typical pattern of streaming applications.

```
Repeat:
1:    Read some input into a buffer;
2:    Process the data in the buffer;
3:    Generate output;
```

Figure 3.1: Pattern of Streaming Applications.

Processing of data during different iterations is usually independent and cross-iteration dependences arise infrequently. Therefore, step 2 forms the body of the loop during speculative parallelization. This chapter presents parallelization transformations that enable the above parallelism to be exploited using CorD. This chapter is organized as

follows. Section 3.1 presents a profile-guided algorithm for speculative parallelization, which effectively identifies the speculative body of a streaming application. Section 3.2 proposes a scheme to optimize the copying operations for CorD. Section 3.3 shows the transformation that is performed by the compiler. Section 3.4 discusses two variants of loop parallelization which allows a better performance to be obtained in certain situations. Section 3.5 shows the experimental results and section 3.6 gives the summary.

3.1 Speculative Parallelization Algorithm

Based upon the analysis of several streaming applications, it was determined that a loop iteration can be partitioned into the prologue, speculative body, and the epilogue. The algorithm for performing the partitioning first constructs the prologue, then the epilogue, and finally everything that is not included in the prologue or the epilogue is placed in the speculative body. Profiling is used to distinguish frequently arising cross-iteration dependences from infrequently arising cross-iteration dependences. Below is the construction of the prologue and the epilogue:

Prologue. The prologue is constructed such that it contains all the input statements that read from files (e.g., *fgets()*). This is because such input statements should not be executed speculatively. In addition, an input statement within a loop is typically dependent *only* upon its execution in the previous iteration – this loop carried dependence is needed to preserve the order in which the inputs are read from a file. Therefore input statements for multiple consecutive loop iterations can be executed by the main thread before the speculative bodies of these iterations are assigned to parallel threads for execution. Loop index

update statements (e.g., $i++$) are also included into the prologue, as the index variables can be considered as the input of each iteration and hence should be executed non-speculatively.

Epilogue. The epilogue is made up of two types of statements. First, the output statements are included in the epilogue because output statements cannot be executed speculatively. If an output statement is encountered in the middle of the loop iteration or it is executed multiple times, then the code is transformed so that the results are stored in a memory buffer and the output statements that write the buffer contents to files are placed in the epilogue which is later executed non-speculatively by the main thread. Second, a statement that *may frequently depend* upon another statement in the preceding iteration is placed in the epilogue if the probability of this dependence manifesting itself is above a threshold. This is obtained through the analysis of profiling results. Any statements that are control or data dependent upon statements already in the epilogue via an intra-iteration dependence are also placed in the epilogue.

Figure 3.2 illustrates the partitioning of a loop body. In the *for* loop shown on the left, the first statement is a typical input statement that reads data from a file and stores it into a buffer. Hence, it is placed into the prologue. Then the epilogue of this loop is constructed. First, all output statements (lines 5 and 12) are added in the epilogue. Since the profiling information can reveal that a cross-iteration dependence at line 10 is exercised very often, this statement is also added to the epilogue. Thus, the epilogue of this loop has three statements, as shown by the code segment to the right in Figure 3.2. Note that in this example, all three statements appear in the middle of the loop. Thus, a buffer is used to store the information of epilogue statements such as the PC of statements and values of

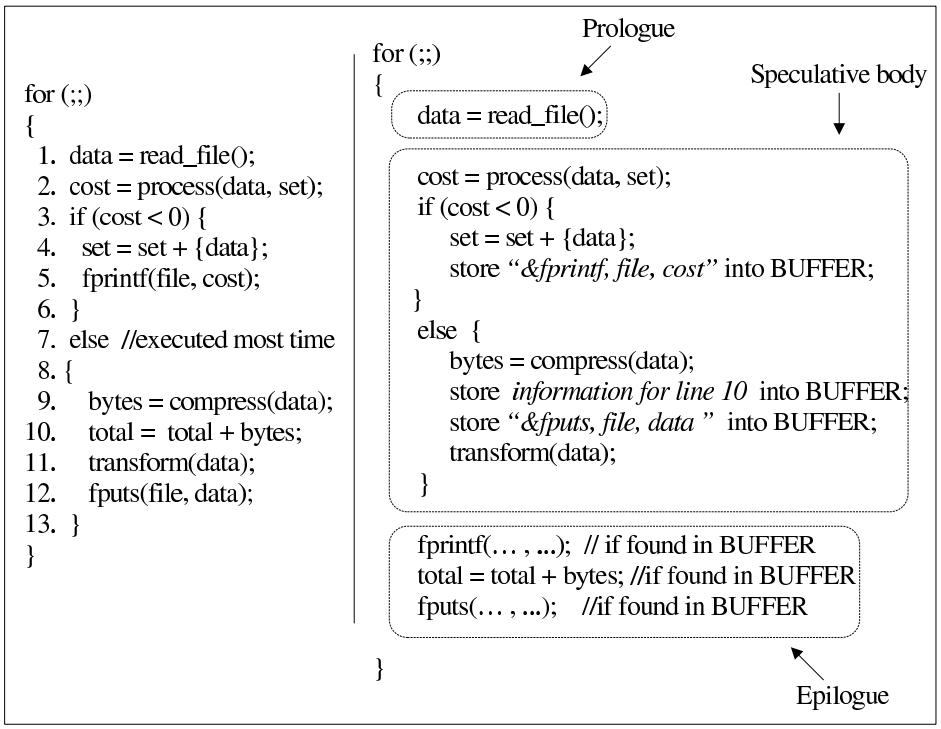


Figure 3.2: Partitioning A Loop Into Prologue, Speculative Body, And Epilogue.

the arguments. When the epilogue is executed by the main thread, the information stored in this buffer is referenced.

After the prologue and epilogue of a loop are identified, the rest of the code is considered as the speculative body as shown in Figure 3.2. Note that line 4 may introduce cross-iteration dependence because of the accesses to variable *set* which is also referred to as a *live-in* variable. However, if the profiling results show that this dependence seldom manifests itself, a speculation is performed on this variable.

Once a loop is successfully partitioned, optimistic parallelism in the speculative loop body has been extracted and ready to be exploited. Figure 3.3 shows how threads in CorD actually exploit this parallelism. The left figure shows the static version of a loop is divided into three sections: the prologue, the speculative body, and the epilogue. The

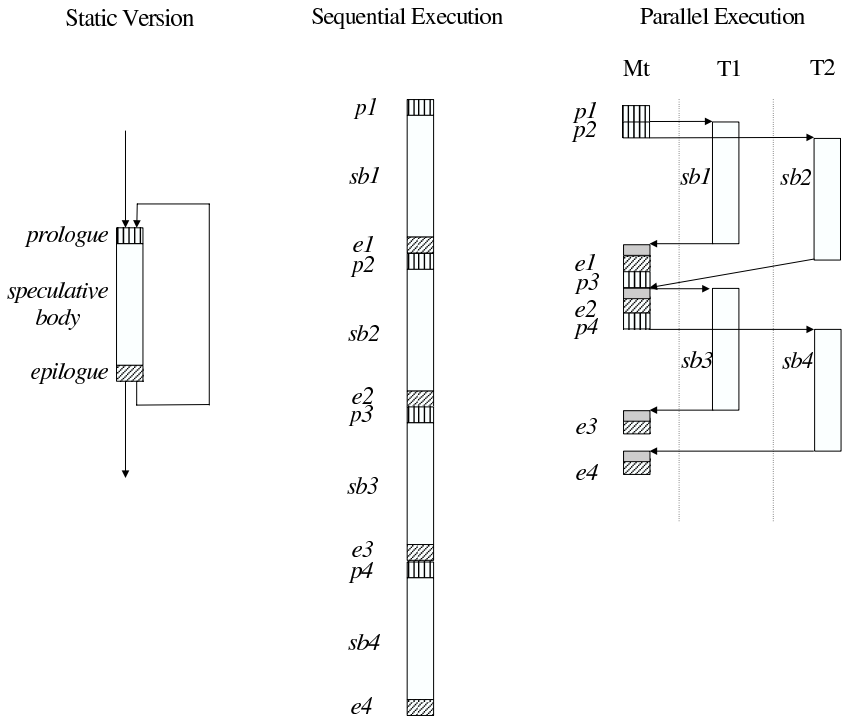


Figure 3.3: Thread Execution Model.

sequential execution of four iterations of the loop is shown in the middle. The corresponding speculative parallel execution is shown on the right. The main thread (Mt) *non-speculatively* executes the prologues and epilogues, while the parallel threads (T1 and T2 in the example) are created to *speculatively* execute the bodies of the iterations on separate cores. Speculative execution entails optimistically reading operand values from non-speculative state and using them in the execution of speculative bodies. If the speculations are successful, then the total execution time is reduced as pairs of consecutive iterations are executed in parallel. If more parallel threads are used, the execution time can possibly be further reduced.

3.2 Optimizing Copying Operations For CorD

Since *state separation* of CorD model presents a clear and simple model for interaction between the non-speculative and speculative computation state, it is amenable for implementation through compiler and runtime support. However, this simplicity comes at the cost of *copying overhead*. The compiler can play an important role in minimizing copying overhead. Through compile time analysis the subset of non-speculative state that must be copied to the speculative state is identified. In particular, if the compiler identifies data items that are *not referenced* by the speculative computation, then they need not be copied. Moreover, shared data that is *only read* by the speculative computation, need not be copied as it can be directly read from the non-speculative state.

In the above discussion, it is assumed that all data locations that may be accessed by a parallel thread have been identified and thus code can be generated to copy the values of these variables from (to) D state to (from) P state at the start (end) of parallel thread speculative body execution. Let us refer to this set as the *Copy Set*. This section discusses how the *Copy Set* is determined. One approach is to use compile-time analysis to conservatively overestimate the *Copy Set*. While this approach guarantees that any variable ever needed by the parallel thread would have been allocated and appropriately initialized via copying, this may introduce excessive overhead due to wasteful copying. The main causes of *Copy Set* overestimation is that even when the accesses to global and local variables can be precisely disambiguated at compile-time, it is possible that these variables may not be accessed as the instructions that access them may not be executed.

Reducing Wasteful Copying. To avoid wasteful copying, a profile-guided approach is used. It identifies data that is highly likely to be accessed by the parallel thread and thus potentially underestimates the *Copy Set*. The code for the parallel thread is generated such that accesses to data items are guarded by *access checks* that determine whether or not a data item's value is available in the P state memory. If the data item's value is not available in P state memory, a *Communication Exception* mechanism is triggered that causes the parallel thread to interact with the main thread to transfer the desired value from D state memory to P state memory. Specifically, a one-bit tag is used for each variable to indicate if the variable has been initialized or not. Note that uninitialized variables, unlike initialized ones, do not have entries in the mapping table. The accesses (reads and writes) to these variables must be modified as follows. Upon a read, the variable's tag is checked and if the tag is not initialized, then the parallel thread performs actions associated with what is called a *Communication Exception*.

A request is sent to the main thread for the variable's value. Upon receiving the response, which also includes the version number, the variable in P state memory is initialized using the received value, and the variable's entry in the mapping table is updated. Upon a write, the WriteFlag in the *mapping table* is set and if there is no entry for the variable in the mapping table, an entry is first created.

Optimizing Communication Exception Checks. Even for the variables which are created in P state memory at the start of a parallel thread's execution, some of the actions can be optimized. First, not all of these variables require copying in and copying out from D state memory to P state memory. Second, all the actions associated with loads and stores

of these global and local variables during the execution of a parallel thread may not be required for all of the variables, that is, some of the actions can be optimized away. As shown in Table 3.1, the variables are classified according to their *observed dynamic behavior* which allows the corresponding optimizations.

Type of Variable	Copying Needed	Actions Needed
Copy In	Copy In = *YES*; Copy Out = *MAYBE*	Put Actions at Stores
Copy Out	Copy In = *MAYBE*; Copy Out = *YES*	Put Actions at Loads
Thread Local	Copy In = *NO*; Copy Out = *NO*	No Actions
Copy In and Out	Copy In = *YES*; Copy Out = *YES*	No Actions
Unknown	Copy In = *MAYBE*; Copy Out = *MAYBE*	All Actions

Table 3.1: Variable Types in Parallel Threads.

A *Copy In* variable is one that is observed to be *only read* by the parallel thread during profiling. Therefore its value is definitely copied in and no actions are performed at loads. However, actions are performed at stores to update the WriteFlag in the mapping table so that the value can be copied out if a store is executed. A *Copy Out* variable is one that is observed to be written during its first access by the parallel thread while profiling, and thus it is not copied in but requires copying out. However, actions are needed at loads to cause a communication exception if the value is read by the parallel thread before it has been written by it. *Thread Local* variables are ones that definitely do not require either copy in or copy out, and *Copy In and Out* are variables that are always copied in and copied out. Thus, no checks are required for variables of these types. Finally, all other variables that are observed not to be accessed during profiling are classified as *Unknown*. If these

are accessed at runtime by a parallel thread, the accesses are handled via communication exceptions and thus no optimizations are possible for these variables.

3.3 Transforming Partitioned Loop

This section shows the detailed form of the main thread and the parallel thread created by the speculative parallelization transformation. The interactions between the main thread and the parallel threads are first discussed. Then the details of the work carried out by each of the threads are presented.

3.3.1 Thread Interactions

The main thread and a parallel thread need to communicate with each other to appropriately respond to certain events. This communication is achieved via messages. Four types of messages exist. When the main thread assigns an iteration to a parallel thread, it sends a Start message to indicate to the parallel thread that it should start execution. When a parallel thread finishes its assigned work, it sends a Finish message to the main thread. When a parallel thread tries to use a variable that does not exist in the P space, a communication exception occurs which causes the parallel thread to send an Exception message to the main thread. The main thread services this exception by sending a Reply message.

The main thread allocates a *message buffer* for each parallel thread it creates. This message buffer is used to pass messages back and forth between the main thread and the parallel thread. When a parallel thread is free, it waits for a Start message to be deposited in its message buffer. After sending an Exception message, a parallel thread waits for the main

thread to deposit a Reply message in its message buffer. After sending Start messages to the parallel threads, the main thread waits for a message to be deposited in any message buffer by its parallel thread (i.e., for a Finish or Exception message). Whenever the main thread encounters an Exception message in a buffer, it processes the message and responds to it with a Reply message. If a message present in the message buffer of some parallel thread is a Finish message, then the main thread may or may not process this message right away. This is because the results of the parallel threads must be *committed in order*. If the Finish message is from a parallel thread that is next in line to have its results committed, then the Finish message is processed by the main thread; otherwise the processing of this message is postponed until a later time. When the main thread processes a Finish message, it first checks for misspeculation. If misspeculation has not occurred, the results are committed and new work is assigned to the parallel thread, and a Start message is sent to it. However, if misspeculation is detected, the main thread prepares the parallel thread for reexecution of the assigned work and sends a Start message to the parallel thread. The above interactions continue as long as the parallel threads continue to speculatively execute iterations.

3.3.2 Main Thread

The code corresponding to the main thread is shown in Figure 3.4(a). In addition to executing the prologue and epilogue code, the main thread performs the following actions. It calls `init_version_table()` (see Figure 3.4(b)) to initialize the version table that it must maintain. Next it creates parallel threads one at a time and initializes them by calling `init_thread` (see Figure 3.4(c)). During the first call to the initialization routine corresponding to a parallel thread, the main thread allocates the C space for the mapping

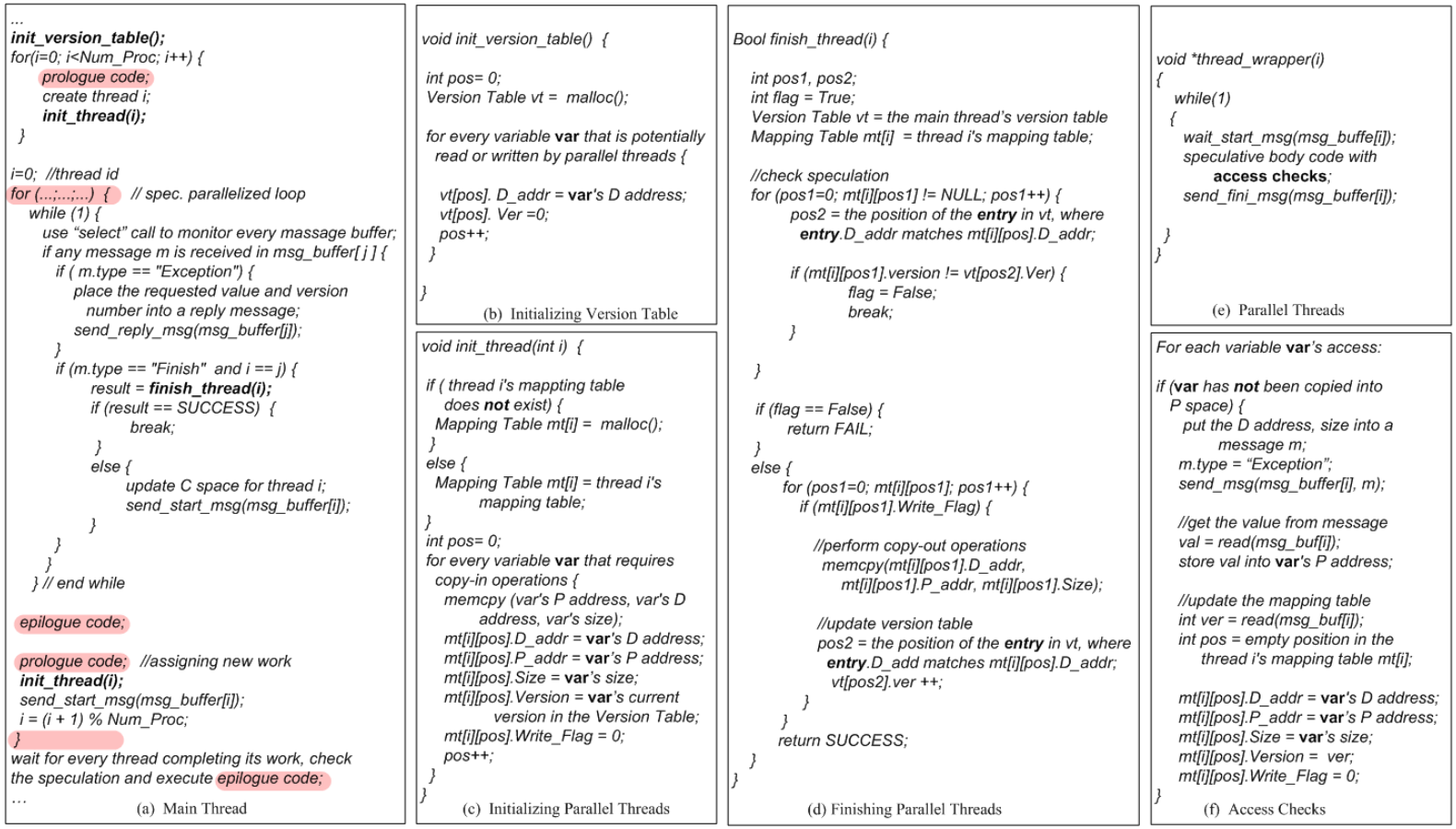


Figure 3.4: Code Transformation.

table and initializes the mapping table. In subsequent calls to the initialization routine for the same thread, the mapping table is only initialized since it has already been allocated. Copy operations are also performed by the main thread for the variables that are marked as `Copy In` and `Copy In and Out`. Note that the P space into which values are copied is allocated when a parallel thread is created. For `unknown` variables, no record in the mapping table is setup. Instead, these variables' one-bit tags are initialized to false. The main thread also gets a new iteration by executing *prologue code* and then assigns it to the newly created thread. After the initialization work, the main thread enters a main loop where it first waits for messages from parallel threads and responds accordingly. It processes Exception requests from all parallel threads until finally a Finish message is received from the parallel thread that executed the earliest speculative iteration currently assigned to the parallel threads (this is thread `i` in the code). Upon receiving this message, it calls `finish_thread` routine (see Figure 3.4(d)). This routine first performs checks to detect *misspeculation* by examining the version numbers. If speculation is successful, it commits the results and returns SUCCESS. Otherwise it returns FAIL and as a result the main thread prepares the parallel thread for reexecuting the assigned iteration. Committing the result is essentially implemented by performing copy-out operations by consulting the mapping table. Once the results have been committed, the *epilogue code* is executed. Next, the main thread executes the *prologue code* for the next available iteration and prepares the idle parallel thread to execute this iteration. Finally, the value of `i` is updated to identify the next parallel thread whose results will be committed by the main thread.

3.3.3 Parallel Threads

After its creation, each parallel thread executes function `thread_wrapper` shown in Figure 3.4(e). After entering the `while` loop, a parallel thread `i` waits for the `start` message from the main thread. Upon receiving this message, it executes the *speculative body code* and then sends the `Finish` message to the main thread. Note that the speculative body code is also obtained by transforming its corresponding sequential code in couple of ways. First, code is introduced to perform updates of the `WriteFlag` in the mapping table. The above code is put immediately following every store access to variable `p_var` which can be of any type but `thread local`. This information is used by the main thread when the copy-out operations are performed. Second, code for access checks must be introduced. When a `unknown` variable is read, its one-bit tag has to be checked. If it is not true, that means this variable has not been copied into the current thread's P space, so we copy it on the fly. Note that the above code is inserted immediately before every read access to variable `p_var`. Although copying on-the-fly seems to be complex and may potentially slow down the execution due to the use of message passing, it does not cause much overhead at runtime because of two reasons. First, these `unknown` variables are not accessed in the profiling run and hence, are very unlikely to be accessed in the normal run. Therefore, the likelihood of executing the corresponding access checks is very small. Second, the copy operation for each `unknown` variable only needs to be performed once. For the subsequent accesses, no request to the main thread has to be sent as the value has been copied into the P space of the current thread.

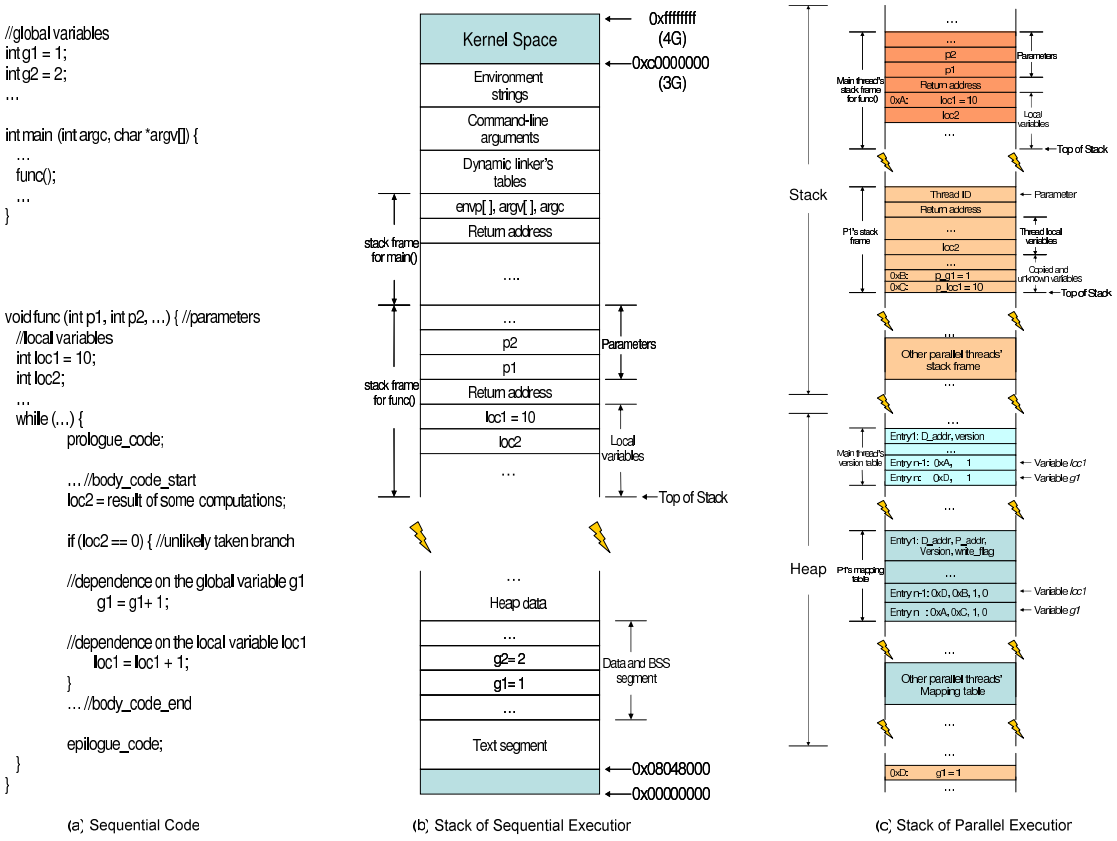


Figure 3.5: Runtime Memory Layout For Sequential Version and Parallel Version.

3.3.4 Runtime Memory Layout

The previous section described how to transform the code from sequential version to the parallel version. This section shows the execution difference between these two versions via the comparison of the virtual memory layout under the 32-bit linux OS. Figure 3.5(a) shows an example of the sequential code with variables needing different types of treatment under the copying scheme. In particular, `g1` and `g2` are global variables. The `main` function calls `func` which has `p1` and `p2` as its parameters. Before the main loop `while`, there are two local variables `loc1` and `loc2` declared in the `func`. The `while` loop has already been partitioned into three parts. Note that in the *body code*, `loc2` is classified as a `thread local` variable as it is always defined before used. An unlikely taken branch is introduced in which `g1` and `loc1` carry out a cross-iteration respectively. In other words, speculations can be made on these two live-in variables since the two increment statements are unlikely to be executed.

Figure 3.5(b) shows the virtual memory layout when the `func` is being executed sequentially. The text segment starts at the bottom (0x8048000). The data and BSS segments, where `g1` and `g2` are allocated, are next to it. Above these two segments, heap starts growing towards higher virtual addresses. On the other side, the address that can be used as the stack is from 0xc0000000 (3G) to lower addresses. All local variables, parameters and the return address of the `func` are stored on the stack. In particular, right after the stack frame for `main`, the parameters of `func`(e.g., `p1` and `p2`) are stored. All local variables of `func` (e.g., `loc1` and `loc2`) are stored after the `return address` location. As the execution continues, the stack grows and shrinks accordingly.

Figure 3.5(c) shows the virtual memory layout when the `func` is being executed

under the CorD execution model. First, the stack frame for `func` is split into several frames, one for each thread. The very top frame is for the main thread which is identical to the sequential version. The one next to it is the frame of the first parallel thread `P1`. This frame contains one parameter (`ThreadID`), the return address, thread local variables (e.g., `loc2`), copied variables (e.g., `p_g1` which corresponds to `g1`, `p_loc1` which correspond to `loc1`) and unknown variables. Note that the copied variables have correct values at the start of the parallel thread's execution because of the copy-in operations performed by the main thread. Other threads also have a similar stack frame which essentially is the P space in the CorD.

The C space in the CorD, as described earlier, is implemented through the `malloc` function call. In other words, the mapping table for each parallel thread and the version table for the main thread are allocated in the heap. From the figure, each entry contains an address and its current version in the version table. For example, the global variable `g1`'s current version is 1. The mapping table (P1's mapping table is shown in the figure) contains the mapping information of each copied variable. For instance, the address 0xD (corresponding to `g1`) is mapped to the address 0xB (corresponding to `p_g1` in P1's stack).

3.4 Loop Parallelization Variants

The loop parallelization algorithm developed in this section assigns individual iterations of the loop to separate cores. Once an iteration is completed, its results are committed to non-speculative state. However, there are situations in which it is more appropriate to commit results of executing multiple iterations or results of executing a part of a single iteration to non-speculative state. These variants of speculative loop parallelization are discussed in this section.

Committing Results of Multiple Iterations. The performance of the parallel version may be hindered by *thread idling*. If a parallel thread that is assigned work earlier, finishes its work before some later threads are assigned work by the main thread, it has to wait for the main thread to check its result. However, it may take a long time for the main thread to finish assigning the later iterations to other threads. So during this period, this parallel thread cannot do any other work but simply idle. This causes substantial performance loss. This situation arises when relative to the number of cores being used, the work being assigned to a single core in one step is quite small. Thus, to avoid thread idling, the work assigned to each parallel thread can be increased in a single step, that is, the main thread can assign two or more iterations of work to a parallel thread in a single step. This ensures that every parallel thread stays busy while the main thread is still assigning the later iterations to other parallel threads.

Committing Results of a Part of an Iteration. During the partitioning of the loop body the algorithm constructed the speculative body in such a manner that it did not contain any statements that are involved in frequently occurring (or definitely occurring) loop dependencies. This was done because frequent dependences give rise to frequent misspeculations; hence making the performance of the parallel execution no better than the sequential execution. Currently statements involving these dependences must be included in the prologue of the epilogue. However, it is observed that in some programs the inclusion of statements involved in frequent or definite dependences into the prologue or epilogue yielded a very small speculative body. In other words most of the statements are part of the prologue or epilogue and relatively few statements are actually contained in the speculative

body. Thus, significant performance enhancement in such situations cannot be expected.

It is observed that to address the above problem, and effectively exploit the parallelism available in loops, a loop iteration must be partitioned into multiple distinct speculative code segments. These speculative code segments are interrupted by statements involving frequent or definite cross-iteration dependences and thus their execution across parallel threads must be serialized. This modified approach yields a parallelized loop in which misspeculations are once again an infrequent occurrence. Another consequence of allowing multiple speculative code segments is that after one such segment is executed its result must be committed before the execution of a parallel thread can proceed further. In other words, misspeculation checks followed by commit or reexecute must be performed multiple times for different speculative segments of the loop body.

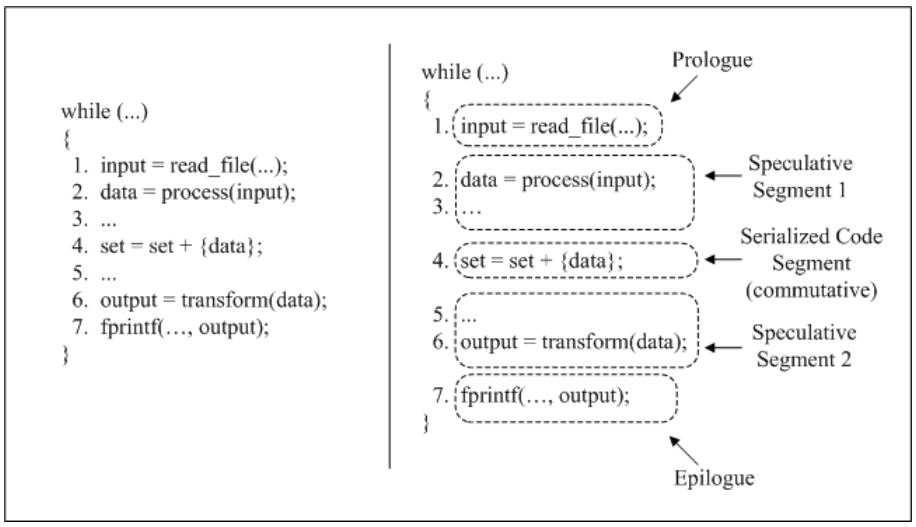


Figure 3.6: Partitioning A Speculative Body Into Several Parts.

Figure 3.6 shows a loop requiring the partitioning of the loop body such that it contains two distinct speculative code segments. On the left, a *while* loop contains typical input and output statements which can be put into the prologue and epilogue respectively.

However, the rest of the statements cannot be put into a single speculative body because of statement 4, which introduces a dependence across all loop iterations. Therefore, statements 2 through 6 are divided into three parts as shown on the right: the first speculative code segment, the serialized segment, and the second speculative code segment. During execution, when each speculative code segment is started, the main thread performs copy-in operations; and when each speculative code segment is finished, the main thread performs the misspeculation checks. If speculation succeeds, copy-out operations are performed; otherwise, only the failed part is reexecuted. The execution of statement 4 that intervenes the execution of two speculative code segments never causes misspeculation as its execution is serialized. It is further observed that the executions of statement 4 by different loop iterations being executed in parallel can be carried out in any order, that is, they are *commutative* [6, 38]. Thus, the updates to variable *set* can be expressed and implemented as critical sections. Another thing that needs to be noted is that segment 1 in a later iteration cannot be dependent on segment 2 in an earlier iteration. Otherwise, committing segment 1 may cause the results to be incorrect. This property can be checked through static analysis.

3.5 Experiments

3.5.1 Experimental Setup

The speculative loop parallelization approach described in this chapter has been implemented and the experiments for a set of programs have been conducted. To speculatively parallelize loops, a profiler implemented using the Pin [48] instrumentation framework is used to first profile the loop code to gather information such as the dependence graph and

dynamic access patterns of variables. Based upon the gathered profile data, the compiler performs code transformation. The compiler infrastructure LLVM [44] is used as it allows us to perform static analysis and customized transformation. All the experiments were conducted under CentOS 4 OS running on a dual quad-core (i.e., total of 8 cores) 3.0 GHz Xeon machine with 16GB memory.

The benchmarks used in the experiments are divided into two groups. The first group contains five programs taken from *SPEC* [31] and *MiBench* [26] suites in which the program transformation that creates a single speculative code segment is used, i.e., one loop iteration or multiple loop iterations are assigned to a single parallel thread and after all of the assigned work is finished the results are committed to non-speculative state. The programs in this group include: `197.parser`, `130.li`, `256.bzip2`, `255.vortex`, and `CRC32`. The second group contains the sequential version of five programs from the *STAMP* [55] suite of programs and they include `bayes`, `kmeans`, `labyrinth`, `vacation`, and `yada`. The main loops in all these programs contain definite cross-iteration dependences caused by commutative statements. Thus, parallelization in these programs requires creation of multiple speculative code segments from a single loop iteration which are intervened by statements involving definite cross-iteration dependences.

Table 3.2 describes the characteristics of the programs used in the experiments. In this table, the first column gives the program name. The profiling input is shown by column *Prof. Input* and the column *Exp. Input* gives the input used in the experimental evaluation. The next two columns show the contents of the prologue and epilogue of the parallelized loop. The profiling results are analyzed to identify different communication types of each variable used in the speculative body. The last four columns show the distribution of

Program	Prof. Input	Exp. Input	Prologue	Epilogue	Vars. in Body			
					I	O	L	IO
`197.parser`	1K file	36K file	fgets	printf	49	6	12	2
`130.li`	6 scripts	72 scripts	i++	printf, var++	30	0	3	6
`256.bzip2`	200K file	7.5M file	fgetc	fputc	12	8	11	1
`255.vortex`	test/input	train/input	++i	fprintf	76	5	4	6
`CRC32`	one 4M-file	80 4M-files	−−argc	printf	1	0	2	1
`bayes`	simulator input	non-simulator input	get a task	update TMLIST	4	0	4	1
`kmeans`	n2048-d16-c16	n65536-d32-c16	loop++	update new center	5	0	3	2
`labyrinth`	x32-y32-z3-n96	x512-y512-z7-n512	get a task	update PVECTOR	3	0	2	3
`vacation`	simulator input	non-simulator input	i++	None	6	0	5	2
`yada`	633.2	ttimeu-1000000.2	get a task	update region var++	2	1	2	4

I-*Copy In* O-*Copy Out* L-*Thread Local* IO-*Copy In and Out*

Table 3.2: Characteristics of Benchmarks.

variables across these categories.

3.5.2 Performance

This section presents the results of execution speedups obtained using the parallelization algorithms. In this experiment, the baseline which is the sequential execution time of the loop that was parallelized is first measured. Then the time of executing this loop in the model with different numbers of parallel threads is measured. Figure 3.7 shows the speedup for the the first group of programs (*SPEC* and MiBench programs). Figure 3.7 shows when the number of parallel threads increases, the speedup for all benchmarks goes up linearly. The highest speedup achieved ranges from 4.1 to 7.8 across the benchmarks when 8 parallel threads are used. It is also noticed that the performance of the model with

one parallel thread is slightly worse than with the sequential version for some benchmarks. That is due to the copying, tracking and checking overhead. In fact, if only one core is available, the main thread is forced to perform all computations rather than spawning parallel threads.

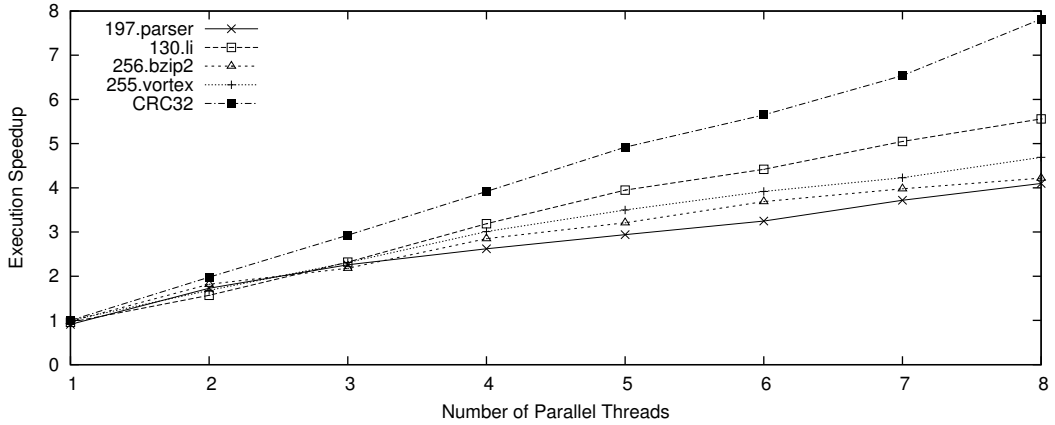


Figure 3.7: Execution Speedups For SPEC and MiBench Programs.

The data shown in Figure 3.7 was obtained by selecting loop parallelization that reduced the effect of thread idling. In particular, with the exception of `CRC32`, thread idling was observed in the four other programs whenever more than five parallel threads were created. In fact it is observed that when a single iteration of work was assigned to each parallel thread, thread idling caused performance to remain unchanged for `197.parser`, `130.li`, `256.bzip2`, and `255.vortex` when the number of threads was increased beyond five threads. However, when multiple iterations were assigned to each parallel thread in a single step, thread idling was eliminated and improvements in speedups was observed. Figure 3.8(a)-(d) shows the speedups of these benchmarks with and without elimination of thread idling.

Figure 3.9 shows the speedups for the second group of programs. Most of the

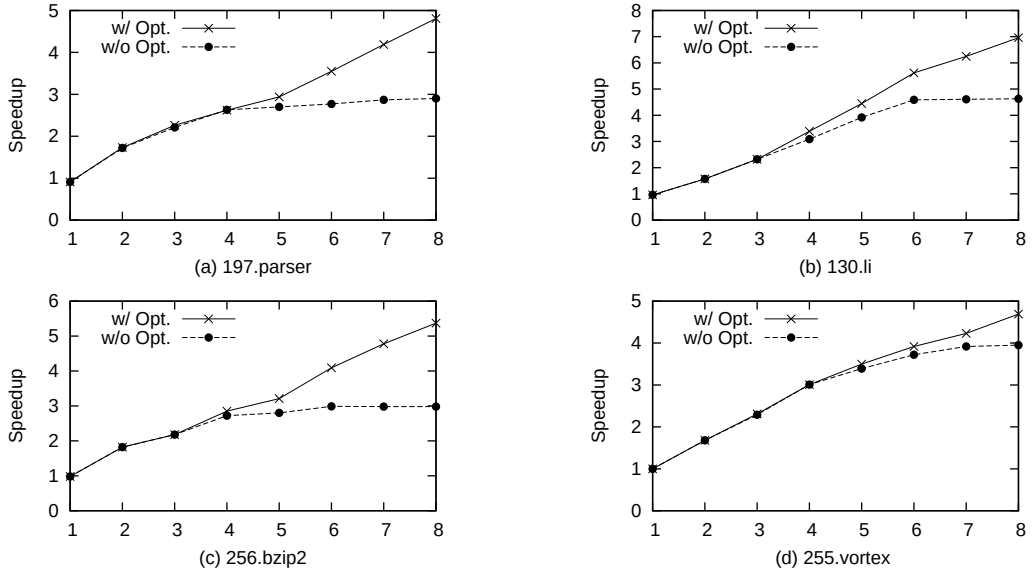


Figure 3.8: The Effectiveness Of Handling Thread Idling.

computations performed in these programs are performed on either the global heap data or the thread local data. The accesses to the global heap are the ones that give rise to serialized code segments inside a loop which are commutative and thus as long as these regions are executed atomically, the correctness will be ensured regardless their execution order. In the implementation of the parallel versions, software transactional memory system [29] was used to execute the commutative regions. The rest of the speculative code segments are executed using the CorD model.

Figure 3.9 shows that for all 5 programs used, some benefit from parallelizing the execution is obtained. However, the speedups results are smaller than those that were observed for the first set of programs. This is because of several reasons. First, in some benchmarks, the non-parallelizable code takes a significant amount of execution time (`bayes`, `yada`). Second, the execution of serialized code limits the speedup that can be obtained. Third, more interactions between the main thread and parallel threads are

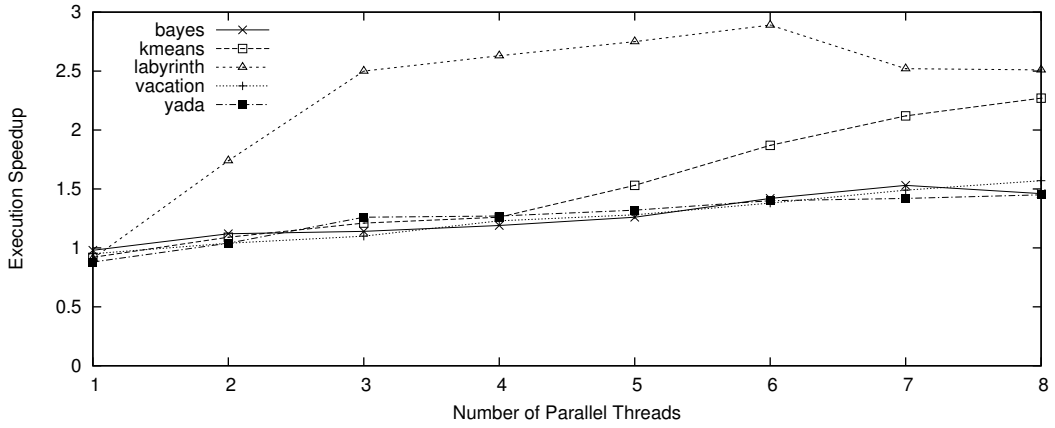


Figure 3.9: Execution Speedups For STAMP Programs.

required due to the partition of the body code which increases the overhead of using the parallelization strategy.

3.5.3 Overhead Analysis

Time Overhead. Software speculative parallelization involves overhead due to instructions introduced during parallelization. This execution time overhead is measured in terms of the fraction of total instructions executed on each core. The results are based upon an experiment in which 8 parallel threads are used, and the overhead is broken down into five categories as shown in Table 3.3. The second column *Static Copy* is the fraction of the total number of instructions used for performing copy-in and copy-out operations by the main thread. This overhead ranges from 0.02% to 5.28% depending on how many variables need to be copied. The third column *Dynamic Copy* gives the fraction of instructions for on-the-fly copying. The *Exception Check* column shows the fraction of instructions used by parallel threads to check if a variable has been copied into the local space. According to the results, these two numbers are very low for the benchmarks used. Another category

of overhead comes from the *Misspeculation Checking*. This uses 1%-2% instructions for all SPEC benchmarks and less than 1% instructions for CRC32 and STAMP benchmarks which do not have many variables to copy. Besides the above four categories, there are other instructions executed for *Setup* operations (e.g., thread initialization, mapping table allocation and deallocation etc.). The last column shows the setup overhead. In total, no more than 7% of total instructions executed due to the execution model on each core.

Space Overhead. Since the memory is partitioned into three states during the execution, and each parallel thread has its own C and P space, extra space certainly needs to be used in the execution model. So the space overhead of the executions of parallelized loops is measured. The space overhead is shown in Figure 3.10. The space used by the sequential version serves as the baseline.

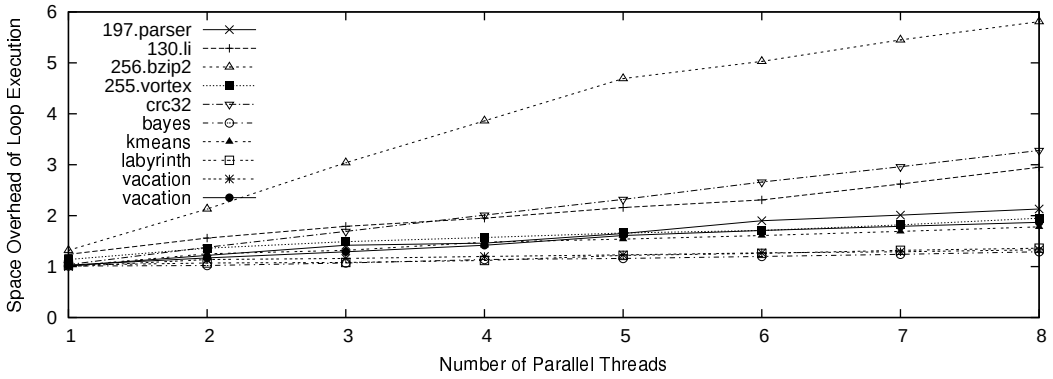


Figure 3.10: Memory Overhead

The overhead for most benchmarks is between 1.3x-3.2x when 8 threads are used. Given the speedup achieved for these benchmarks, the memory overhead is acceptable. For `256.bzip2`, a large chunk of heap memory allocated in D space is used during the compression. In the execution model, each parallel thread makes a copy of this memory

Program	Static Copy	Dynamic Copy	Exception Check	Misspec. Check	Setup
197.parser	3.51%	0.33%	0.02%	1.76%	0.62%
130.li	0.08%	0	0	1.08%	0.07%
256.bzip2	1.32%	0.25%	0.06%	1.03%	0.48%
255.vortex	5.28%	0.04%	0.01%	1.25%	0.39%
CRC32	0.02%	0	0	0.01%	0.32%
bayes	0.43%	0	0	0.15%	0.11%
kmeans	0.57%	0	0	0.02%	0.38%
labyrinth	0.13%	0	0	0.01%	0.09%
vacation	0.89%	0	0	0.66%	0.30%
yada	0.87%	0	0	0.53%	0.26%

Table 3.3: Overhead Breakdown on Each Core.

space to execute the speculative body. Therefore, as more parallel threads are used, more memory is consumed.

Program	Sequential Version	Parallel Version
197.parser	234K	239K
130.li	179K	183K
256.bzip2	53K	57K
255.vortex	1336K	1370K
CRC32	8K	10K
bayes	165K	170K
kmeans	47K	48K
labyrinth	106K	111K
vacation	170K	173 K
yada	182	186K

Table 3.4: Size of Binary Code.

Besides the dynamic space consumption, the increase in the static size of the binary is also examined. As shown in Table 3.4, the increase varied from 1K to 5K for most programs, a very small fraction of the binary size.

3.6 Summary

This chapter presented a profile-guided parallelization algorithm to speculatively parallelize streaming applications. Optimizations were proposed to reduce the communication overhead between parallel threads and the main thread. The experiments performed show that the presented approach achieves significant speedups on a server with two Intel Xeon quad-core processors.

Chapter 4

Handling High Misspeculation Rate Via Multiple Speculations

The presence of cross-iteration dependences in a sequential loop causes misspeculations when loop iterations are speculatively executed in parallel. If such dependences frequently take place at runtime, the speculative parallelization technique cannot improve the program performance. The reason is that these frequent dependences cause the speculation to fail very often and thus wipe out the benefits of parallelism. Although the idea of dividing one loop iteration into different segments proposed in the previous section can alleviate this problem to some extent, it is not a general solution because it requires absence of cross-segment dependences and commutative property of statements involved in frequent cross-iteration dependences. Another solution is to explicitly pass the value of live-in variables [88] between threads. Synchronization and *send/receive* instructions are inserted by the compiler to enforce cross-iteration dependences. However, a *receive* call blocks the recipient till the needed value becomes available. This blocking results in a significant degree

of serialization. In addition, significant amount of time is spent on communication between different threads when considerable number of variables' values are communicated.

This chapter proposes a general technique, called *multiple speculations*, which combines a realization of CorD model and multiple value prediction to speculatively parallelize loops in a sequential program that have frequently arising cross-iteration dependences. The organization of this chapter is as follows. Section 4.1 motivates the work and describes how multiple speculations are incorporated in CorD. Section 4.2 describes how multiple value predictions are generated and how the code is transformed. A runtime adaptive scheme for exploiting more parallelism is proposed in section 4.3. Section 4.4 gives the evaluation results followed by the summary.

4.1 Overview Of Multiple Speculations

4.1.1 Motivation

Figure 4.1 shows a speculative parallelization example where the loop iterations can be speculatively executed in parallel. Specifically, there is a cross-iteration dependence on variable *latest_config* between statements at lines 1 and 9. If the condition *cond1* is always or frequently true at runtime, using speculative parallelization technique cannot speed up the execution as misspeculations occur frequently. The dependence carried by variable *latest_config* causes the loop iterations to be executed sequentially. Moreover, the overhead of the technique such as isolating speculative states and dealing with misspeculation could make the performance even worse.

To address this problem, this section presents a technique *multiple speculations*,

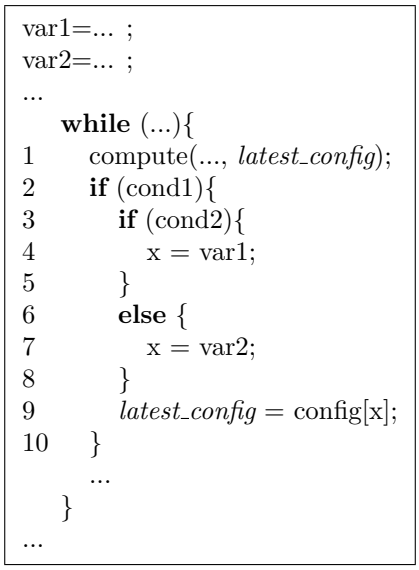


Figure 4.1: Speculative Parallelization Example.

that allows the parallelism to be exploited when such *frequent* dependences exist. The key idea is that for every two consecutive iterations that have data dependences on some variables (a.k.a *live-ins*), the value of such variables for the second one is predicted. If the prediction is correct, then executing these two iterations in parallel using the predicted values for the later iteration yields a speedup. For example, in Figure 4.1, if both *cond1* and *cond2* are always true, then predicting *latest_config* to be *config[var1]* enables speculative parallelization to succeed and lead to a better performance.

However, a single predicted value may not be very accurate. For instance, consider the scenario for Figure 4.1 in which *cond1* and *cond2* keep evaluating alternately to true and false. Thus, a single prediction for the value of *latest_config* is not effective as it is not frequently successful. To solve this problem, multiple predictions are employed, each giving rise to a distinct version of the second iteration. The idea is that among all predictions that are chosen, it is highly likely that one prediction will turn out to be correct and the

corresponding version will generate the correct result. More importantly, the correct result is computed in parallel with the execution of the first iteration. In other words, parallelism is exploited by executing two consecutive iterations in parallel.

In the example shown in Figure 4.1, for the second iteration of every pair of consecutive iterations, three versions that are capable of generating the variable *latest_config* based on different path selections can be created. Table 4.1 shows these versions. In the first version, it is assumed that both *cond1* and *cond2* are true in the first iteration and thus *latest_config* is set to *config[var1]* when the computation starts in the second iteration. In the second version only *cond2* is assumed to be false, and *latest_config* is set to *config[var2]* at the beginning of the second iteration. Finally, in the third version *cond1* is assumed false in the first iteration. Thus, the computation of the second iteration can be directly started. For these three versions, one of them must be correct and thus lead to an execution speedup. This prediction method is essentially based upon collecting data slices of *latest_config* along different paths and creating a version of the loop iteration for each distinct data slice. Next section describes further details of the prediction method.

4.1.2 Adapting CorD For Multiple Speculations

To support the *Multiple Speculations* scheme, the CorD model presented in chapter 2 needs to be adapted. A program is still compiled to contain one *main* thread and multiple *parallel* threads. However, for any two consecutive iterations, the first iteration is executed by the main thread and each version of the next iteration is executed by a parallel thread.

Figure 4.2 shows the thread execution model. The original sequential execution, which consists of 4 iterations, is shown on the left. The corresponding parallel execution

Version Number	Path In An Earlier Iteration	Prediction In A Later Iteration
1	*cond*1=true, *cond*2=true	x=*var*1; *latest_config= config[x];*
2	*cond*1=true, *cond*2=false	x=*var*2; *latest_config= config[x];*
3	*cond*1=false	No Prediction Code

Table 4.1: Three Versions For Generating *latest_config*.

is shown on the right. There is one main thread and multiple parallel threads. The main thread executes iteration 1 and 3 while parallel threads execute different versions of the iterations 2 and 4. A parallel thread does not begin execution until it receives the *start* signal from the main thread.

The dark region at the start of each parallel thread represents execution of code that predicts the values of the shared data (live-in variables) that are to be computed by the previous iteration. Following this code, the computation of the current iteration is performed. The dark region at the end of the main thread's execution represents execution of code that validates the results. After finishing its own computation, the main thread needs to identify the parallel thread executing the correct version of the next iteration, and uses its result to continue the execution. The parallel thread that generates the correct result is also called the *winner*. Figure 4.2 shows that parallel threads P2 and P3 are the winners for iterations 2 and 4 respectively. In this execution model, every two dependent iterations can be executed in parallel and hence the theoretical speedup for the parallel execution is 2.

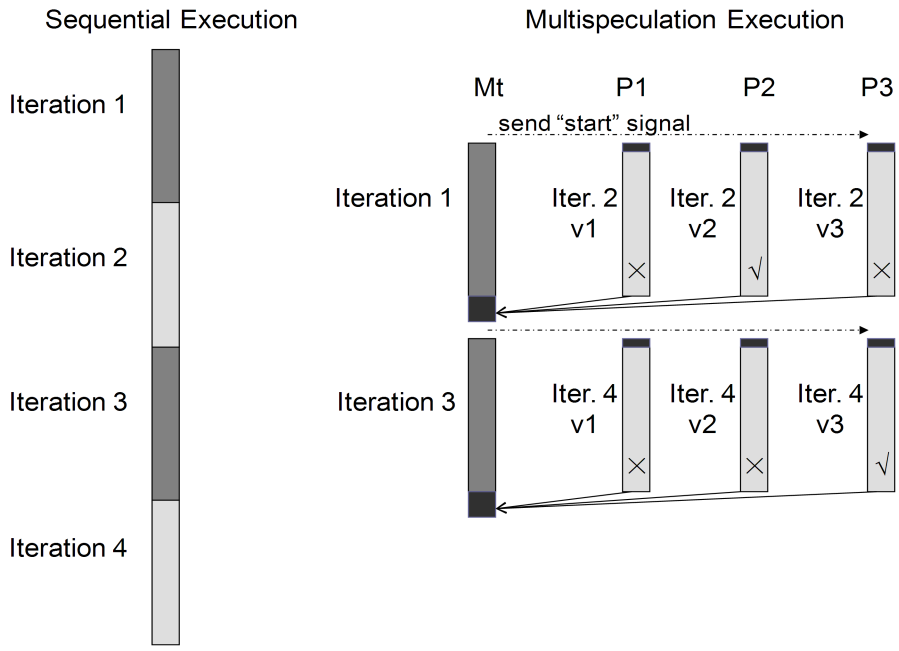


Figure 4.2: Thread Execution Model.

4.2 Basic Scheme Of Multiple Speculations

4.2.1 Choosing Parallelization Candidate

The *multiple speculations* technique is based on value predictions, and therefore resolves the situation where tasks of a program cannot be done in parallel due to dependences. A loop is a good candidate for multiple speculations if the following two conditions are satisfied:

- The loop has *frequent* cross-iteration dependences; and
- The values carried by cross-iteration dependences are *predictable*.

The first condition requires the examination of cross-iteration dependences. If the ratio of the number of iterations involving such dependences to the total number of

iterations is above a threshold, the dependence is considered as being frequent.

The second condition emphasizes that values of live-in variables are predictable. A variable is considered to be predictable if its value can be computed through a *small backward data slice* [2]. This can be checked by analyzing the trace of each iteration in the profiling run and computing dynamic slices from the traces. If a variable's slice is very large, the slice can be further shrunk by applying some value prediction methods. More details are described in section 3.2.

The dependence frequency and predictability conditions must both be satisfied for the application of the technique. They both can be checked based on the information collected from the profiling run. Note that if the first condition is not satisfied, then the program is a good candidate for application of the approach described in the previous chapter or other speculative parallelization approaches [15, 34, 38, 40, 41].

4.2.2 Generating Multiple Versions

Using Data Slices and Control Flow Paths. To construct a speculative version of the second iteration, the value prediction code of live-in variables needs to be inserted before the original loop iteration code. The most accurate value prediction for a variable is to compute the value by executing its *full slice* extracted from the first iteration. However, the size of the full slice can be as large as the computation of the whole iteration. Using such a slice to obtain the value is the same as executing the two iterations sequentially.

To construct small prediction code and take advantage of the multiple value prediction model, following steps are used to generate multiple versions of the second iteration. First, only the *backwards data slices* of a live-in variable needs to be computed. All the

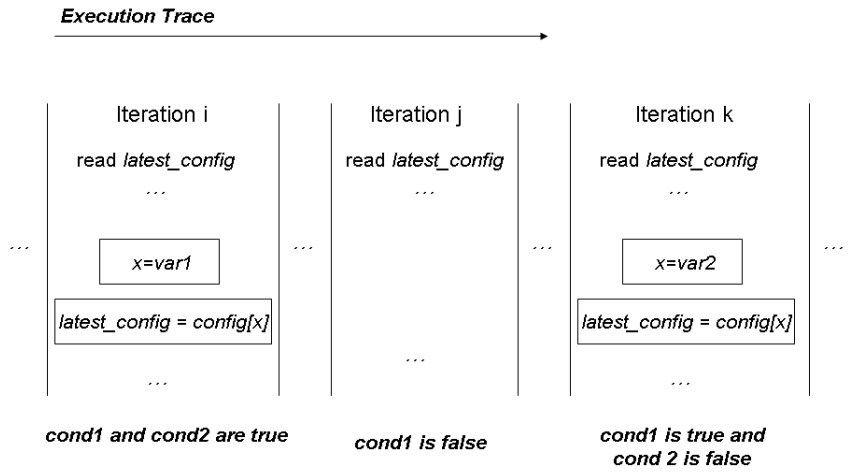


Figure 4.3: Trace Of Figure 4.1.

control dependences and the dependence chains of predicates are removed. Since different control flow paths may be taken in the first iteration, the data slice is computed on each *different path*. Consequently, multiple data slices are obtained for a live-in variable. At this point, multiple versions can be created for the second iteration based on different control flow paths taken by the first iteration. Specifically, each path corresponds to one version and the data slice on that path is used to predict the live-in variable. The data slice and path information are computed based on the profiling trace. Figure 4.3 shows an example profiling trace of the *while* loop in Figure 4.1.

In the example, three things are observed. First, there exists a frequent loop dependence on variable *latest_config* because two thirds of the iterations are involved in a dependence on variable *latest_config*. Second, there exist three different ways or data slices for computing this variable as shown in iterations i, j and k. All three are very small in comparison to the computation of the whole iteration. Therefore, three versions are created for the second iteration. Last but not the least, the path execution frequency, which equals

to the frequency of the occurrence of each kind of slice, can be easily computed. This number is used to compute the version confidence (VC) which reflects the probability of a version being correct.

$$VC = path\_frequency \times prediction\_confidence$$

The *prediction_confidence* for each live-in is always 1 if a data slice is used as the prediction code. If there are multiple live-ins, their data slices are merged for the same control flow path. In other words, one big data slice is created, that is a union of all live-in variables' slices for each path and use it as the prediction code. The overall prediction confidence is the product of individual live-in's confidence, which is still 1 in this case.

Reducing Data Slice on Each Path. Although by using data slices and path frequencies in generating multiple speculative versions, the likelihood of covering the correct prediction of live-in variables is greatly increased, the performance can still be limited due to large sizes of the data slices. For example, if variables *var1* and *var2* in Figure 4.1 are computed within the loop, then the data slices of *latest_config* may be very large. Besides, merging the slices of multiple live-in variables can also lead to a large slice. Executing a large slice in each parallel thread can nullify the benefits of parallelism.

To tackle this problem, the data slice on each path can be reduced by computing a *partial slice*, where the value of a live-in variable can be computed based on the predictions of other variables in the original slice. Given the data slice on each control flow path, the algorithm shown in Figure 4.4 is used to traverse the slice backwards and construct the prediction code of a live-in variable.

```
function compute_partial_slice(){
    input = a data slice of var;
    output = {}; //prediction code
    partial_slice = {};//partial slice of var
    size = predefined maximum size of output;
    i = 0;
    OCBQ = 0;
    boundary = a FIFO queue;

1.  boundary.enqueue(var);
2.  while(i < size) {
3.      c = get_predictability(boundary)
4.      if  (c > OCBQ) {
5.          ouput = prediction_stmt(boundary) +
6.                  partial_slice;
7.          OCBQ = c;
8.      }
9.      v = boundary.dequeue(var);
10.     stmt = the definition of v in input;
11.     partial_slice += {stmt};
12.     boundary.enqueue(source variables in stmt};
13.     i++;
14. }
15. return output;
}

function get_predictability(){
    input = a set of variables;
    overall_confidence = 1;

16. for each var in input{
17.     c1 = valuePredictor1_confidence(var, trace);
18.     c2 = valuePredictor2_confidence(var, trace);
        ...
19.     cN = valuePredictorN_confidence(var, trace);
20.     var.pred_flag = method i whose confidence is the highest;
21.     overall_confidence *= max(c1,c2,...,cN);
22. }
23. return overall_confidence ;
}
```

Figure 4.4: Prediction Code Construction For *var*.

The idea behind this construction is that a point in a data slice is searched, where all variables in the slice can be either computed or predicted with high confidence using some simple value predictor. The search range is limited by a predefined value *size* (line 2 and 13), which can be set to a fraction of total number instructions in the iteration.

In the algorithm, the *boundary*, implemented as a FIFO queue, stores those variables that need to be predicted to execute the statements stored in *partial_slice*, which compute the remaining variables appearing in the slice. This boundary queue initially contains the live-in variable *var* (line 1), and is used to traverse the slice backwards in a breadth-first fashion. Specifically, in each search iteration, the first variable in the *boundary* is popped out (line 9), and its definition in the slice is identified and stored in the *partial_slice* (line 10-11). After that, all source variables in the definition are pushed into the *boundary* (line 12).

A function is called every time the boundary queue changes (see line 3 which calls *get_predictability()*). It looks up the trace of the profiling run to find the values of every variable stored in the queue (also called *boundary variables*). Since each variable may have a different value in a different loop iteration, a value sequence is made for each variable by considering its value at the beginning of each iteration. Then different value predictors are applied on this sequence to compute a confidence number of the prediction (line 17-19). For every variable, its prediction method is chosen by identifying the one with the highest confidence number. This prediction method is stored in the global flag *pred_flag* maintained for the variable (line 20). These highest numbers are multiplied and store the result into $overall\_confidence$ (line 21). This product is used as the overall confidence of the boundary queue (OCBQ) as it indicates how good the combined prediction of all variables

in the current boundary queue is. A boundary queue that has the highest OCBQ is used to construct the prediction code, which basically contains the predictions of variables in the boundary and the statements in the *partial_slice* (line 3-6). The function *prediction_stmt* is called every time a higher OCBQ is found (line 4-8). When generating the prediction statements for each variable, it uses the best prediction method determined in function *get_predictability*.

To construct function *get_predictability*, three different value predictors are used, *last value predictor* [47], *stride predictor* [68] and *context predictor* [86]. In the *last value predictor*, the same value is assumed as being used again. Therefore, the last used value needs to be saved. The *stride predictor* assumes that two consecutive values of a variable have a constant difference (*stride*). Therefore, the most recent stride and values are used to predict the next value. The *context predictor* assumes that the most recent values are most likely to be used. Thus, it maintains a history of most recent values (normally 4) in a buffer. The prediction is made by referring to the buffer. The confidence of a method is defined as the percentage of the correct prediction.

If a version is constructed through a partial slice instead of a complete data slice, its VC is computed using OCBQ instead of 1.

$$VC = path\_frequency \times OCBQ$$

When more than one live-in variable is considered, this algorithm needs to be applied to all their data slices. The generated prediction code is merged if the data slices are on the same path. As a result, the VC is computed using the overall OCBQ of each

path, which is the product of individual live-in's OCBQ.

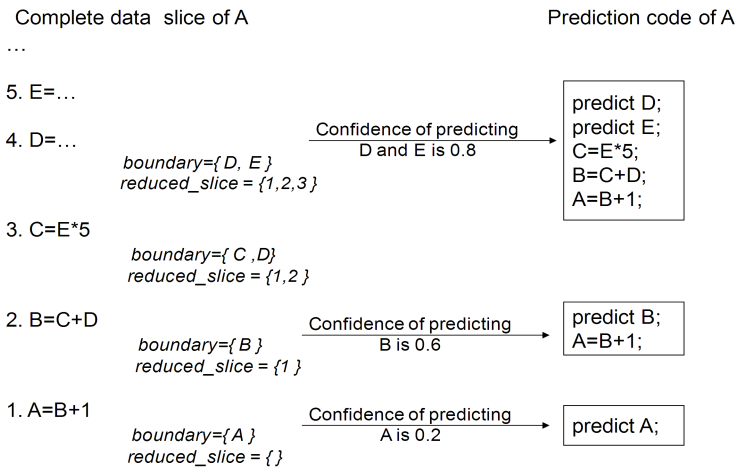


Figure 4.5: An Example Of Reduced Slice Construction.

Consider the example in Figure 4.5. Suppose A is a live-in variable. On the left, the backward data slice of A on one control flow path is shown. If the slice is very large, the algorithm can be used to compute the prediction code for A. At the beginning, A is the only variable in the boundary queue. Since it can be predicted by one method with 0.2 confidence, A can be directly predicted and the prediction is put into the output (as shown in the bottom right). Then this process continues to build better prediction code by examining the data slice backwards. According to the algorithm, the first statement $A = B + 1$ is added into the *partial_slice* set and recompute the boundary queue, which now contains B. Since the confidence of predicting B is 0.6 which is higher than 0.2, the prediction code for A becomes the prediction of B and first statement. By continuing backwards traversal of the data slice the highest confidence is found when predicting both D and E. Therefore, the best prediction code for A can be obtained as shown on the top right, and the corresponding OCBQ is 0.8.

Figure 4.6: Code Transformation.

4.2.3 Code Transformation

This section describes the code transformation performed by the compiler – Figure 4.6 shows the transformation. Given a loop as shown in Figure 4.6(a), trace-analysis tools are first used to analyze the execution trace of the profiling run and generate the prediction code for live-in variables for different paths. The prediction code is generated by the algorithm in Figure 4.4. It is associated with a path represented by the branch history. The transformed parallel version contains the code for the non-speculative thread shown in Figure 4.6(b) and for the speculative parallel threads shown in Figure 4.6(c).

In Figure 4.6(b) one can see that the main thread creates parallel threads before entering the loop. Each created thread executes the function *func* which contains a *while* loop waiting for the "start" signal so as to execute the loop body shown in Figure 4.6(c).

After threads are created, the main thread enters the loop. It first creates multiple versions of the next iteration by executing *next_spec*. Then it executes the current iteration. Finally, it checks the speculation of each thread by executing *result_validation*.

Copying in the main thread. In function *next_spec*, the non-speculative thread needs to perform copying operations. In particular, the variables in the boundary set and the variables that are modified in the loop body are copied to parallel threads' space. This is important to avoid data races. Figure 4.7 illustrates this with an example.

Assume B is a live-in variable and its backwards slice can be traced back to the statement $A = x$; (Figure 4.7(a)). In other words, A is a boundary variable. To predict B, a parallel thread needs to read A and execute the slice of B as shown in Figure 4.7(b). However, the main thread modifies A after A is used for computing B. Hence, it is possible

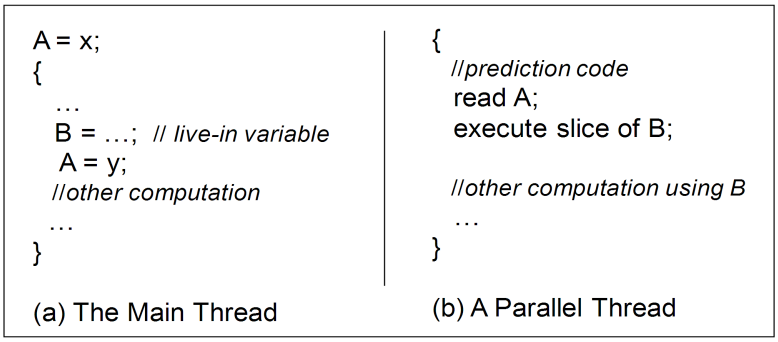


Figure 4.7: An Example Of A Possible Race.

that the parallel thread reads the wrong value of A (y in the example) because of the race condition, and leads to a misprediction of B. To overcome this situation, the main thread should copy A for the parallel thread instead of allowing it to read A. After the copying operation, the main thread sends the "start" signal to parallel threads.

Copying in parallel threads. Parallel threads also need copying operations to ensure state separation. In particular, when a variable is about to be modified during the execution, it is copied from D space to P space. This is well known as the *copy-on-write* scheme. A mapping table is also needed for each parallel thread to store the variables' mapping information. This is used when the results are merged.

Result validation. The result validation work is performed by the main thread. It needs to identify the correct version of the next iteration among all versions. To do that, the main thread needs to track the branch history of the current iteration and record the values of the boundary variables. When validating the result, the main thread simply needs to examine this information of the prediction code in each parallel thread. If a match is found, the corresponding parallel thread is the winner and its results are merged into non-speculative

state. Otherwise, the main thread has to reexecute the next iteration.

Committing results. The winner's results are committed by the main thread. Since the mapping table stored in C space contains the D space addresses of the modified variables, the main thread simply walks through the table and performs memory copying operations.

4.3 Adaptive Multiple Speculation Scheme

Although in the basic scheme the parallelism between every two consecutive iterations is exploited if one of the versions of the second iteration is correct, there are two problems with the basic scheme. First, it is possible that a small number of versions cover all popular execution paths, and thus the VCs of these versions are very high. Therefore, executing other versions, whose VCs are small, from the same iteration on additional cores would be a waste of resources. Second, the computation of each version's VC relies on the path frequency information of the profiling run. In a real run, different inputs may exhibit different path frequencies and thus change the VC. As a result, some versions may be less likely to be correct than expected causing cores to be wasted. Wasting cores can dramatically decrease the system throughput, when multiple applications co-exist on the system. If the parallelized application runs alone, use of extra cores leads to waste of power.

To tackle this problem an adaptive technique is proposed for better use of available cores. The key idea is to consider the versions with a higher VC as candidates for executing additional iterations beyond the second iteration. Figure 4.8 illustrates the idea.

Suppose there are n different paths in a loop iteration. From the figure one can see that iteration *i+1* has n versions as denoted by v_1, v_2, ..., v_n, each of which corresponds

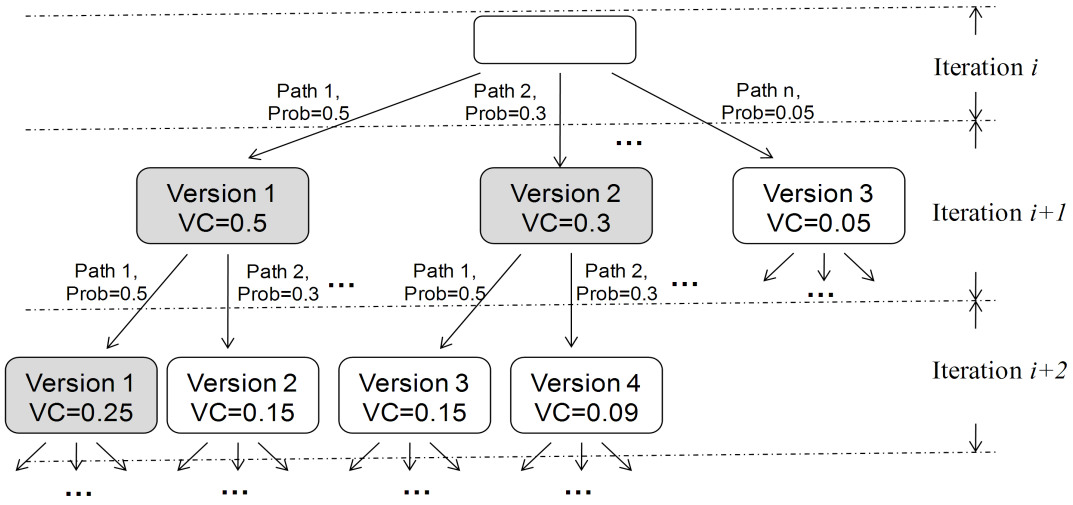


Figure 4.8: Selecting Versions With A Higher VC.

to a certain path that might be taken in iteration i. The probability of each path being taken is marked on the edge. The VC of each version is shown in the node. When the first version of iteration *i+1* is executed, it still takes one of n paths in its own computation. As a result, the prediction code of iteration *i+2* has n^2 different cases leading to n^2 different versions. To calculate the VC of a version in iteration *i+2*, all the probabilities along the path back to the root (iteration i in the example) need to be multiplied. For example, the VC of the second version of iteration *i+2* is the product of 0.3 (the probability of iteration *i+1* taking path 2) and 0.5 (the probability of iteration i taking path 1).

Suppose P is the number of available cores for parallel threads. To assign the work to each parallel thread at runtime, the main thread identifies P versions that have the highest VCs as follows. It first calculates the VCs of all versions of iteration *i+1*. A version with the highest VC is then selected. Next, it computes the VCs of this version's children in iteration *i+2*. Assuming each iteration has n versions, it now has the VCs of *n-1* versions in iteration *i+1* and the VCs of n versions in the iteration *i+2*. It continues to select a

version with the highest VC among these unselected versions and explores the children of the selected version. If two versions have the same VC, their parents' VCs are used to break the tie. Once the number of selected versions reaches P, the exploration terminates and the P versions are assigned to the parallel threads. In the example shown in Figure 4.8, the versions represented by the shaded node are selected if P is 3. This version-selection process in efficiently implemented using the maximum-heap data structure.

While the technique uses extra cores to improve performance, it is important to avoid using cores to execute the versions that are unlikely to be a winner. This is important for achieving fairness among multiple applications being executed. Therefore, a threshold number is used to prevent a version with small VC from being executed. If the main thread cannot find P versions with VCs larger than the threshold, then the extra parallel threads are set to be inactive so that OS can schedule other applications on the remaining cores.

4.4 Experiments

4.4.1 Experimental Setup

Implementation. The multiple speculations technique was implemented using the Pin [48] instrumentation framework and the LLVM compiler infrastructure [44]. Figure 4.9 shows the procedure for parallelizing a sequential program. A sequential program is first compiled into its executable with debugging information. Then a profiler is used to collect runtime information for outermost loops under a small input – the profiler is implemented by instrumenting the executable using Pin. The information collected includes dependences, values of variables at start of each iteration, and control flow paths taken with their execution

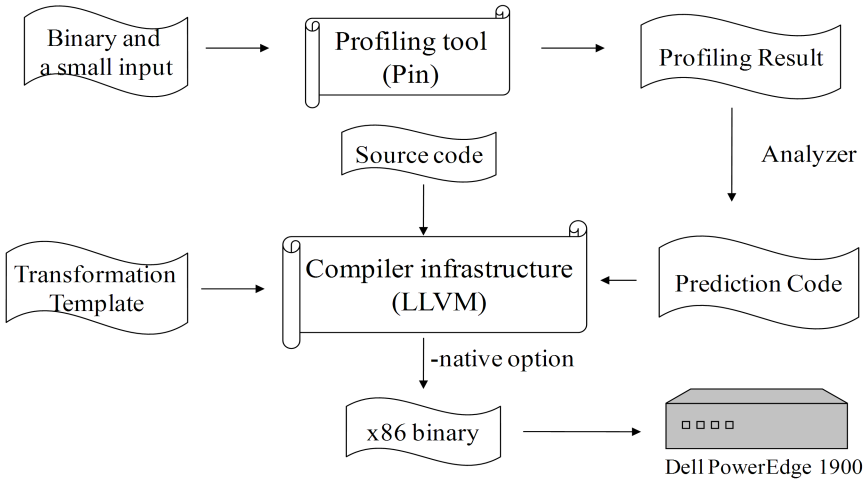


Figure 4.9: Experimental Framework.

frequencies. The dependences are classified into intra-iteration and cross-iteration dependences. The live-in variables are identified by looking at the cross-iteration dependences. The prediction code of these variables on each path is identified using intra-iteration dependences, value sequences, and control flow paths based on the algorithm shown in Figure 4.4. Then, LLVM uses these predictions and a transformation template to recompile the sequential program into the parallelized version. Finally, the parallelized program is executed with a larger input and collect the data under CentOS 4 OS running on a dual quad-core Xeon machine with 16 GB memory. Each core runs at 3.0 GHz.

Benchmarks. In the experiments, ten programs are used. Five of them, namely `dry`, `fldry`, `llu`, `mechcall` and `objinst`, are from the benchmark suite distributed with LLVM. Another five programs are from the SPEC2000 suite: `164.gzip`, `175.VCR`, `255.vortex`, `253.perlbmk` and `300.twolf`.

4.4.2 Performance

Performance of the Basic Scheme. Figure 4.10 shows the performance of each program when the basic scheme is applied where for every two consecutive iterations, different number of versions are created for the second one. It should be noted that throughout the experiments, the maximum number of speculative versions (threads) allowed is 7 since the machine has 8 cores and one core is reserved for the main thread. To avoid thread idling, 10-20 iterations for the programs from LLVM are unrolled. For the SPEC programs, no unrolling is needed because the loop body is large enough.

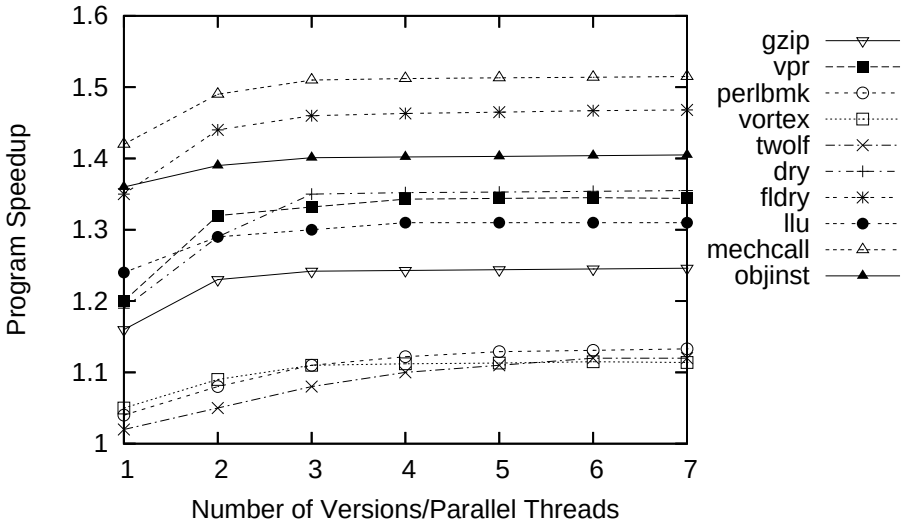


Figure 4.10: Execution Speedups.

As one can see, the speedup for all benchmarks increases faster at the beginning and slower as more versions are used. The highest speedups ranging from 1.12x to 1.51x are achieved when three or four versions are used. This can be explained by Figure 4.11 which shows the cumulative speculation success rate of each benchmark when increasing number of versions are executed in parallel. Figure 4.11 shows that using the first 3 to 4 versions

significantly increases the success rate and thus leads to a big increment in the speedup for every program. When more versions are used, the total speculation success rate does not significantly increase any more, so these later versions provide little contribution to the performance. This is primarily because the paths corresponding to the later versions have low execution frequencies. In these programs, three or four hot paths are taken with over 95% probability. Consequently, the versions corresponding to the infrequently taken paths are not likely to be winners. In the case of *mechcall* and *objinst*, the control flow graphs are very simple and there are only three ways of computing live-in variables. Therefore, using three versions (threads) is enough to execute two consecutive iterations in parallel and the other threads remain idle.

From Figure 4.11, one can also observe that all SPEC benchmarks except for *gzip* have lower speculation success rates (less than 60%) than the other five LLVM benchmarks even when 7 versions are executed. This is because the data slices of *live-in* variables in the LLVM programs are small and hence the speculation success rates are only determined by the path coverage of the speculative versions. For most SPEC benchmarks, however, the value mispredictions on hot paths also cause the speculation to fail, and thus the success rate is reduced. In the case of *gzip*, the value predictions of the live-in variables are very accurate and hence a very high speculation success rate is observed.

Performance of the adaptive scheme. A set of experiments were also conducted for each benchmark using the adaptive scheme. Figure 4.12 shows the results. For each program, two numbers when different number of parallel threads are allowed are measured. The first number is the speedup demonstrated by the line with markers. The left y-axis

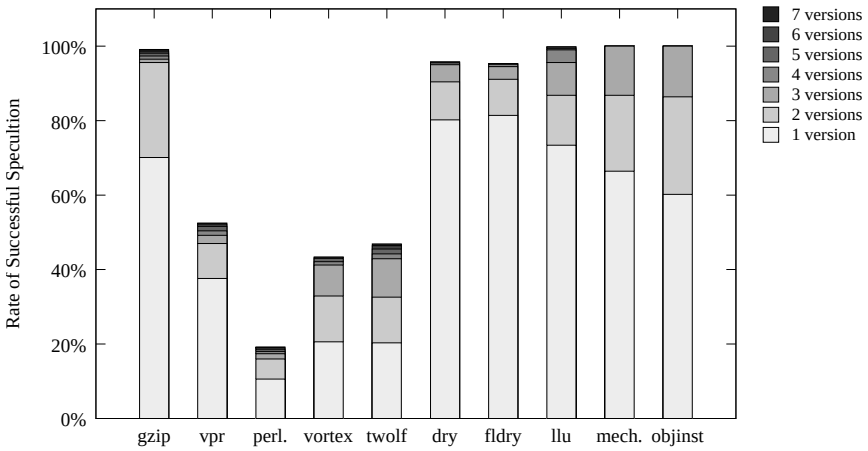
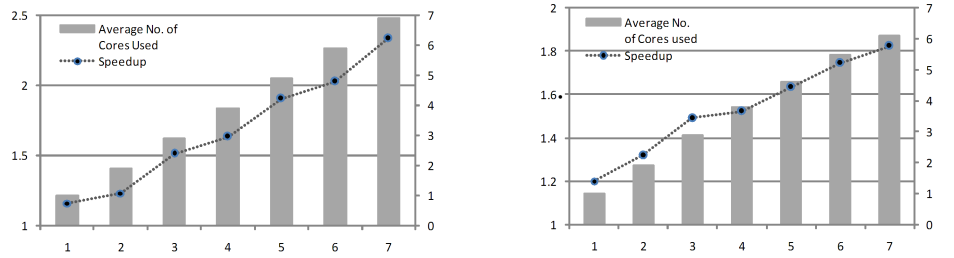


Figure 4.11: Speculation Success Rate.

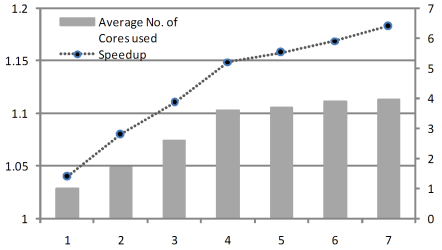
shows the speedup values and the x-axis shows the maximum numbers of parallel threads that are allowed. Compared to the basic scheme, additional speedups are achieved when more threads are used. The highest speedups range from 1.18x (*perlbmk*) to 2.33x (*gzip*) for all benchmarks. These numbers are achieved when 7 versions (parallel threads) are allowed to be used. This is because the adaptive scheme executes a few versions of the third or even fourth iteration at runtime. Since these versions are more likely to be correct than some versions in the second iteration, executing them allows us to exploit more parallelism and achieve higher speedups.

The second number measured is the average utilization of cores. Figure 4.12 shows the results. As described in section 4, to avoid wasting CPU resources, versions with very small VCs (less than 0.05 in the experiments) are not executed and the remaining threads are set to inactive. To measure the actual CPU utilization of each parallelized program, the number of versions that are selected every time by the main thread is recorded. The selected versions are the ones whose VCs are not only above a threshold number, but also among the P highest ones, where P is the maximum number of parallel threads allowed

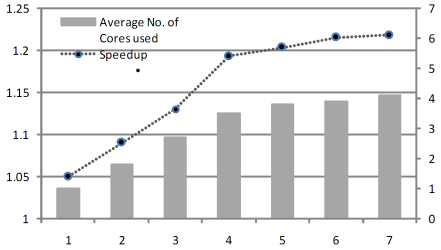


(a) gzip

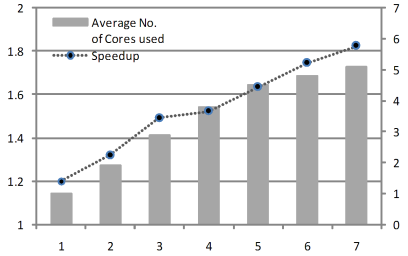
(b) vpr



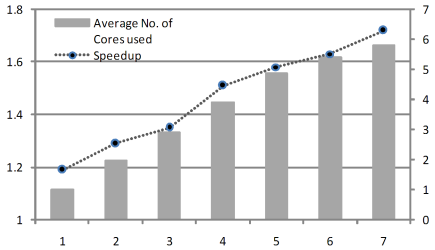
(c) perlbmk



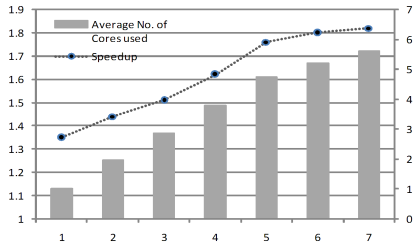
(d) vortex



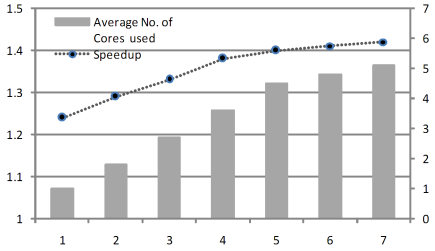
(e)twolf



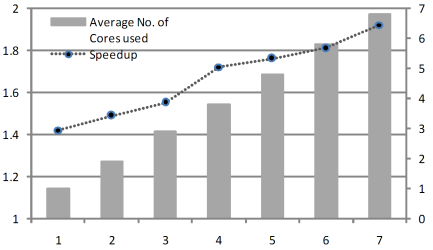
(f) dry



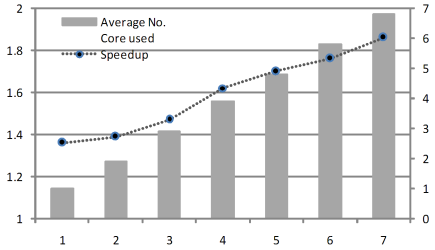
(g) fldry



(h) llu



(i)mechcall



(j) objinst

Figure 4.12: Performance Of Adaptive Scheme.

(shown on the x-axis). Then the average of all these numbers obtained throughout the whole execution is computed. The data are represented by the bars in each figure and the y-axis on the right shows the values. For all benchmarks, the utilization is very high when the maximum number of parallel threads allowed is less than 4. After that, some programs may not always use all cores to execute the speculative versions. In particular, when 7 threads are allowed, for *vortex* and *perlbmk*, the best speedup is achieved by only using around 4 cores on an average. In other words, the main thread normally cannot find more than 4 versions with VCs above the threshold at runtime and hence saves the extra 3 cores without sacrificing performance. This saving for *llu* and *twolf*, is about 2 cores, and for *vpr*, *dry*, and *fldry* is one core on an average. In the case of *gzip*, *objinst*, and *mechcall*, the parallelized program often uses all cores to exploit the parallelism.

Performance Comparison with Other Techniques. The effectiveness of the technique is compared with three other techniques. The first is DOACROSS [8, 45] where the values of all live-in variables are passed through explicit messages. In the experiments, the message-passing scheme is implemented through a POSIX pipe which supports *send/receive* calls. The second technique is TLS, which optimistically assumes no cross-iteration dependences exist. Since the technique is implemented purely in software, the CorD implementation described in the previous chapter is used for comparison. The last technique in the comparison is Mitosis [61] where all live-in variables are pre-computed through full slices. As mentioned earlier, the slice size is normally large. Therefore, an optimization has been proposed in [61] where the data slice on the path that is most frequently taken is kept. Note that Mitosis requires architectural support for speculative execution. For a fair comparison,

Mitosis is evaluated using the software speculation approach. Specifically, Mitosis is simulated by generating one version for each iteration and constructing the pre-computation code by using the complete data slice on a path with the highest frequency.

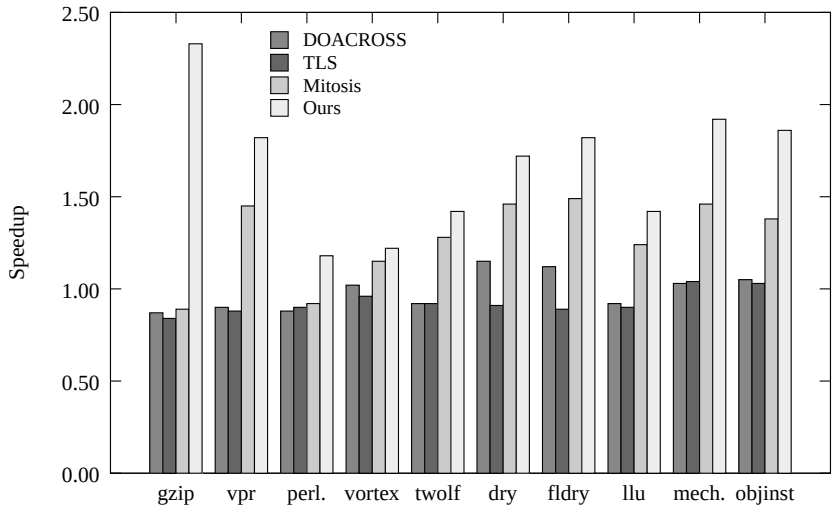


Figure 4.13: Performance Comparison With Other Techniques.

Figure 4.13 shows the results of the comparison where 7 parallel threads are allowed for all techniques. From the figure, one can see that the multiple-speculations technique outperforms the other three techniques for all benchmarks. In most cases, DOACROSS and TLS slow down instead of speeding up sequential executions. The reason for DOACROSS is that it spends significant time on waiting for the values of live-ins, especially when such variables are used very early by a speculative thread. In the case of TLS, being too optimistic leads to excessive misspeculations.

Mitosis performs much better than DOACROSS and TLS for most programs. Compared to TLS, Mitosis has lower misspeculation rate because it uses full data slice to calculate live-ins. The reason for Mitosis not being as good as multiple-speculations technique is that it only uses one version of each iteration. Therefore, if more than one hot

path exists, which is true for most benchmarks as indicated in Figure 4.11, the misspeculation rate is high because the speculation on later iterations is prone to be wrong. As a result, the parallelization benefit is diminished. For *gzip* and *perlbmk*, Mitosis slows down the execution because in these two programs, the size of live-in variables' slices on the most frequent paths is very large. Without value predictions, Mitosis executes the loop almost sequentially and the overhead of the runtime system further degrades the performance.

4.4.3 Overhead Analysis

Time Overhead. The execution model imposes the overhead on the execution of the parallelized program. This overhead is measured by breaking down the execution time into different categories for the parallel threads and the main thread respectively.

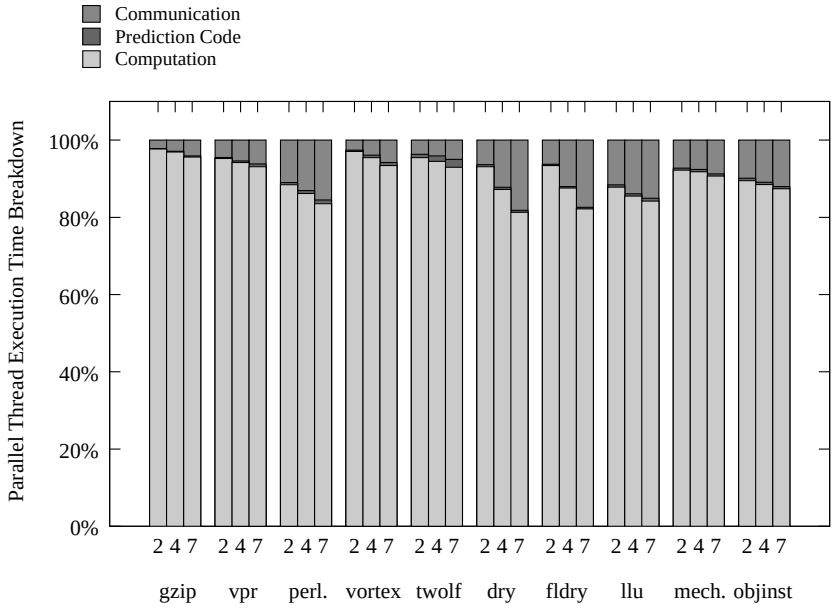


Figure 4.14: Time Breakdown: Parallel Threads.

Figure 4.14 shows the average time breakdown of one parallel thread in case of using a total of 2, 4, and 7 parallel threads. Each parallel thread spent most time on the

computation and less than 1% time on executing the prediction code. The rest of the time is spent on communication with the main thread. According to the results, the communication overhead rises as the number of threads increases. This is because the main thread controls the parallel threads by sending the start signal and examining the results sequentially and thus using more parallel threads leads to a longer waiting time for each.

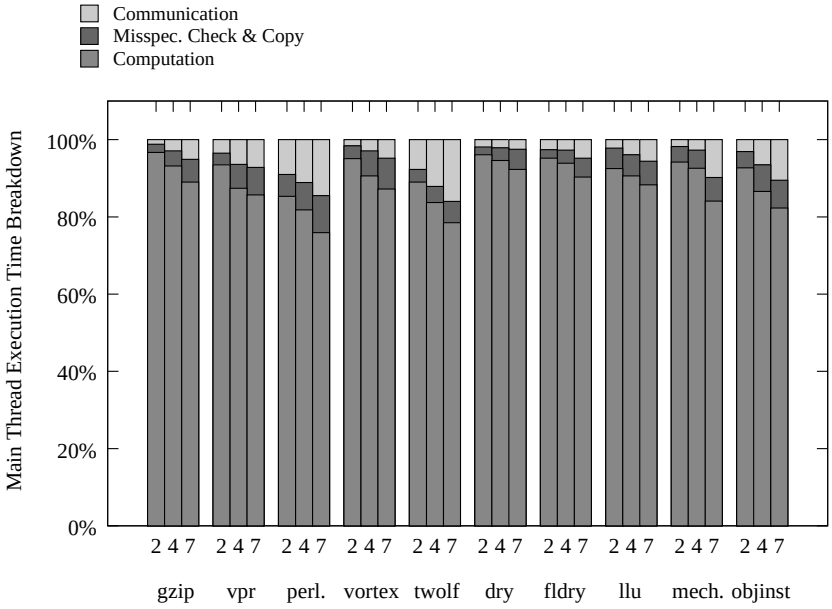


Figure 4.15: Time Breakdown: Main Thread.

Figure 4.15 shows the breakdown of time for the main thread that is responsible for executing the sequential part and some iterations of the parallelizable loops, communicating with the parallel threads, and performing misspeculation checks and copying operations. From the figure, one can see that the *computation* category dominates the execution time for all programs. The fraction of time spent on communication, misspeculation check, and copying operations increases when more parallel threads are used. The sum of these two fractions, which reflects the overhead imposed by the execution model, is less than 25%.

Considering the speedups obtained, the benefit of exploiting parallelism outweighs the cost of implementation overhead.

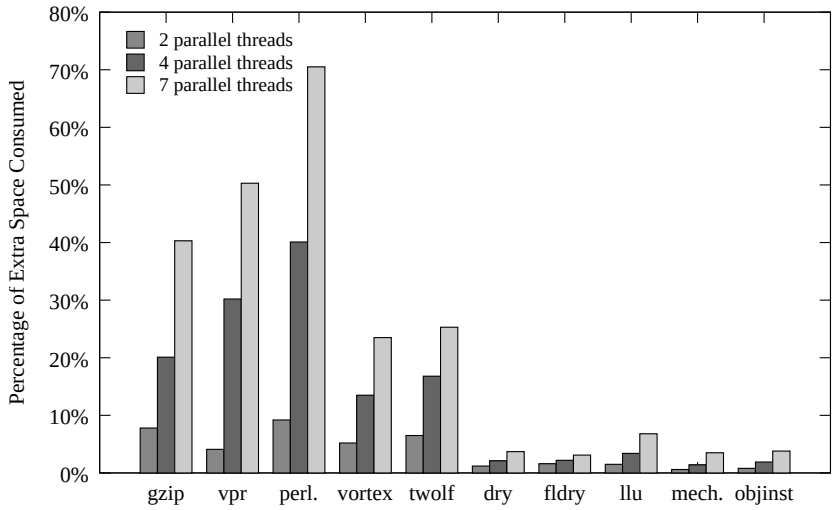


Figure 4.16: Space Overhead.

Space Overhead. The space overhead incurred to obtain speedups is also measured by monitoring the peak value of memory consumption while the parallelized program is run with varying number of threads. As one can see from Figure 4.16, when more threads are used, more space is used for all benchmarks. The parallelized SPEC benchmarks consume more memory especially when 7 threads are used (20%-80%). On the other hand, the parallel versions of the other 5 programs consume less than 7% extra memory regardless of the number of parallel threads. This is because SPEC programs are much larger and have more variables being used in each loop iteration. Consequently, each parallel thread has to maintain a copy of these variables which requires more memory.

4.4.4 Summary

This chapter presented a speculative parallelization technique *multiple speculations* that is implemented based on the CorD model. By using multiple value predictions, this technique resolved frequent cross-iteration dependences and exploited the parallelism between consecutive loop iterations. The experimental results show that, on an average, the technique achieves 1.7x speedup across ten benchmarks.

Chapter 5

Reducing Misspeculation Cost Via Incremental Recovery

For streaming applications, the frequency of cross-iteration dependences is often input dependent. If the misspeculation rate is very low, applying CorD model leads to good speedups. Otherwise, the performance cannot be improved by speculative parallelization. Although the previous chapter proposes a value-prediction based approach to deal with high misspeculation rate, its effectiveness is determined by the accuracy of value prediction. The fundamental reason for the performance loss upon a misspeculation is that all results generated are assumed to be incorrect and hence discarded. However, it is observed that a misspeculation on a live-in variable may not necessarily cause all speculatively computed results to be incorrect. Based on this observation, this chapter describes an approach for *Incremental Recovery* that mitigates the performance loss caused by high misspeculation rate.

The organization of the chapter is as follows. Section 5.1 presents a motivating

example and shows how the incremental recovery technique can be incorporated into CorD. Section 5.2 describes the details of this technique. The experimental results are presented in section 5.3. Section 5.4 summarizes the work.

5.1 Overview Of Incremental Recovery

5.1.1 Motivation

In the techniques presented in the earlier chapters, recovery from misspeculations is achieved by discarding all speculatively computed results. This can be attributed to the CorD model, where each thread has its own space to store all temporal results when performing a computation. The space is automatically created along the parallel thread creation, and after the creation, all writes are performed on this space. If misspeculation occurs on **any** live-in variable, **all** results in the space are discarded. While this recovery scheme is generally better than expensive roll-back schemes, invalidating all speculatively computed results is not an optimized solution. It is observed that during speculative execution, not all speculatively computed results are necessarily incorrect if only the speculation of one variable is wrong. Figure 5.1 shows a simple example.

From Figure 5.1, one can see that each loop iteration has N live-in variables *var1*, *var2*, ..., *varN*. Each of them is updated if the corresponding condition is true. The values of these live-ins are used in the computation functions and the computation results are stored in *r1*, *r2*, ..., *rN*. Suppose all conditions are not true most of time. As a result, this loop is a good candidate for speculative parallelization technique described in chapter 3. During runtime speculative execution, if condition i becomes true, all following iterations that are

```
1:  while (...) {
2:      ...
2:      if (condition1) {
3:        var1++;
5:      }
4:      r1 = comp1(var1);
2:      if (condition2) {
3:        var2++;
5:      }
4:      r2 = comp2(var1, var2, ...);
2:      ...
2:      if (conditionN) {
3:        varN++;
5:      }
4:      rN = compN(var1,var2, ..., varN);
2:      ...
2:      print(rN);
5:  }
```

Figure 5.1: Motivating Example.

being executed in parallel are considered failed because they use the stale value of *var i*. In the original scheme, all speculatively computed results *r1*, ..., *rN* computed by parallel threads are discarded. However, it is obvious that only the values of *ri*, ..., *rN* need to be recomputed, because the wrong value is not used in the remainder of the computation. Discarding all results and reexecuting the entire iteration may lead to a significant waste.

5.1.2 Adapting CorD For Incremental Recovery

To reduce the repeated executions caused by high misspeculation rate, the speculatively computed results generated by a thread have to be separated from each other. This chapter proposes a new adaptation of CorD – decoupling the speculative thread from speculative space – to support the incremental recovery during speculative execution. The key idea is to create multiple write buffers for each thread so that only incorrect results are discarded when misspeculation occurs. Specifically, a new space is created when a

speculatively-used variable is first read, and other variables that are modified use the speculative value are copied into the new child space. Figure 5.2 illustrates the idea.

As shown in Figure 5.2, a parallel thread has three spaces S_1, S_2 and S_3 instead of just one space during the speculative execution. Space S_2 is created when the first access of live-in variable a is encountered, and S_3 is created when the first access of live-in variable b is encountered. With this memory scheme, the recovery can be performed more efficiently. Specifically, if the misspeculation occurs due to a, the results stored in S_2 and S_3 are discarded and recomputed using the correct value of a. However, if the misspeculation is caused by b, only the results in S_3 are recomputed. The results in S_1 are only computed once and are never be discarded during recovery because the computation does not use any speculative variable.

With this scheme, recovery cost can be reduced effectively. Now let us consider the example in Figure 5.1 again. Since the first access to each live-in *var_i* creates a new space, the results using *var_i* are separated from those not using it. As a result, a misspeculation on *var_i* does not invalidate the results not using this variable. Instead, only the results stored in the space corresponding to this live-in and the later created space are recomputed. Hence, this scheme reduces the cost of recovery.

5.2 Realizing Incremental Recovery

5.2.1 Creating Multiple Subspaces

To allow each thread to have multiple subspaces when performing a speculative task, a unique *space identifier* (SID) is assigned to each subspace. Each thread also main-

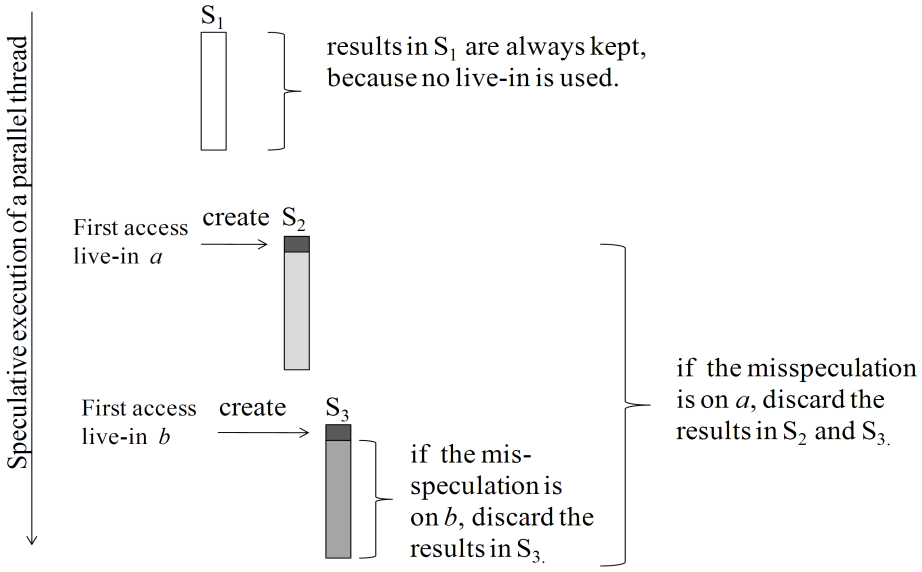
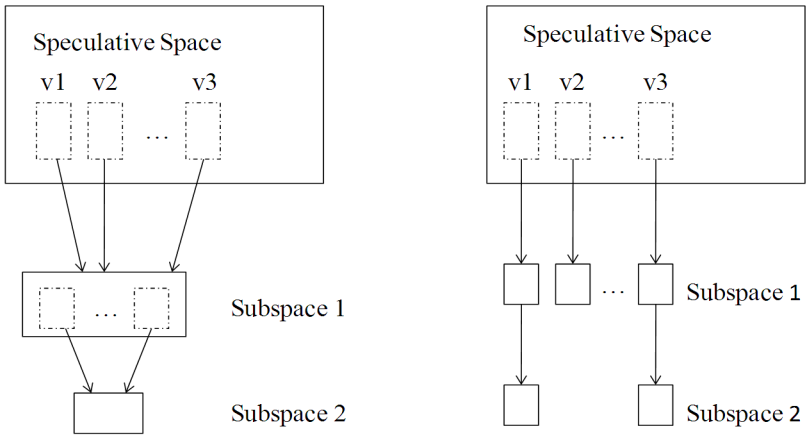


Figure 5.2: Decoupling Space Allocation From Thread Creation.

tains the *current* SID (CSID) as a thread local variable. The SIDs and CSID are useful for copying variables between different subspaces to ensure the separation of multiple subspaces.

While SIDs allow logical separation of different subspaces, the creation of each subspace at virtual memory level can be performed using different schemes. One simple scheme is to create a subspace by allocating a contiguous memory region as shown in Figure 5.3(a). However, the allocation may fail when the size of subspace is very large, because the memory allocator may not always find a free memory chunk of the requested size. The situation becomes even worse when more threads are used. To solve this problem, subspaces are maintained by using the *state history* for each variable. In particular, the state of an individual variable in a particular subspace is represented as a pair of the value and SID, and these states are linked together in ascending order of space creation time as shown in Figure 5.3(b). With this scheme, updating a variable in a different subspace can be simply implemented by adding a (value, SID) pair into this variable's state history.



(a) Allocating Contiguous Subspace (b) Using State History To Maintain Subspace

Figure 5.3: Allocating A Subspace.

Moreover, the memory allocation requests are now at the granularity of variable's size and hence more variables can be handled.

In the scheme, a subspace is created when a live-in variable is first read in a speculative task. All variables that are defined directly or indirectly using the value of the live-in variable are copied to the new subspace. These operations are implemented through live-in access checks and statement transformations as shown in Figure 5.4.

From the figure, one can see that the live-in access check code is inserted in front of each statement. For each source operand that is a live-in variable and first read by a speculative thread T (line 1), a new subspace is created by incrementing *CSID* by 1 (line 3). As a result, each subspace corresponds to a speculative use of one live-in variable, and later created space has a larger space ID. The old CSID and the PC of current statement are stored and are used if the recovery is needed at this point (line 2). The live-in variable is also copied into this new subspace by appending the pair of CSID and variable's value

```
CSID: the current space ID, initialized to 0;
stmt: a statement dst = src1 op src2 executed by thread T

Insert the following live-in access check code before the stmt :
For every source operand s in stmt:
1:  if (s is a live-in variable and is first read by T) {
2:    saving CSID and the PC of the statement for recovery;
3:    CSID++;
4:    sid = CSID;
5:    val = the value of s in D space;
6:    append (sid, val) into the state history of s;
7:  }

To ensure dst is updated in the current subspace, stmt is transformed
into the following:
8:  sid = GetLastSpaceID(dst);
9:  val = GetLastValue(src1) op GetLastValue(src2);
10: if (sid != CSID){ // copy is required
11:   sid = CSID;
13:   append (sid, val) into the state history of dst;
14: }
15: else {
16:   update the last state history record of dst to (sid, val);
17: }
```

Figure 5.4: Live-in Access Checks And Statement Transformation.

in the non-speculative state into its state history (lines 4-6). In other words, the ID of the subspace created for a live-in and the speculatively-read value of the live-in is always stored in the first record of this live-in's state history.

To update the destination operand *dst*, its subspace ID is first computed (line 8) and the updated value is computed using the latest values of its sources operands (line 9). Since the update has to be performed on the current subspace, the space ID of *dst* is compared with CSID (line 10). If they are different, the *(sid, val)* pair is appended into *dst*'s state history. Otherwise, the last record in the state history is updated with the new pair *(sid, val)*.

Function *GetLastSpaceID()* and *GetLastSpaceValue()* are auxiliary functions which retrieve the space ID and value respectively from the last record in a variable's state history.

Besides the live-in access checks and statement transformation performed during speculative execution, a mechanism also needs to be developed to identify the live-in variables that were accessed during speculative execution. As stated in chapter 2, this is important because the values of live-in variables are the ones that require validation and copy-out operations after speculative threads finish their executions. Therefore, a *live-in table* is maintained for each speculative thread to track the live-ins accessed by the thread. The table has the 3 fields as shown below.

NonSpecAddr	SpecAddr	WriteFlag

When a live-in variable is first accessed by a speculative thread T, an entry for this variable is created in the table. The *NonSpecAddr* field and *SpecAddr* field contain the live-in variable's non-speculative address and speculative address respectively. The *WriteFlag* field shows whether the live-in has been updated ever during speculative execution. Once a write access to this variable is encountered, this field is set to true.

Subspaces and the live-in table are maintained during the speculative execution; they are crucial for the misspeculation detection and copy-out operations which will be described in the next two subsections.

5.2.2 Handling Speculative Results

After a speculative thread finishes its task, the main thread needs to perform misspeculation check to validate the speculatively computed results. If the speculation succeeds, the results should be copied back to the non-speculative state. Otherwise, recovery execution should be invoked so that the correct values can be recomputed. In our approach, the results computed by speculative threads are handled in the same order as the tasks are

created. The in-order result-handling ensures the updates to non-speculative state memory is consistent with the sequential semantics.

A misspeculation occurs if a speculatively-read live-in variable has been updated during an earlier speculative task. On the other hand, if a speculative thread updates a live-in variable, then all other speculative threads that are using this variable to perform later tasks should be considered failed because they are using the incorrect value. Based on this idea, a scheme is developed for the main thread to perform the copy-out operations and misspeculation checks. This scheme is presented in Figure 5.5.

Copy-out. Each parallel thread has a flag *Speculation* indicating if its speculation has failed. When a parallel thread finishes its speculative task and the main thread determines that it is the turn of this thread to committing its result, this flag is examined. If the flag indicates that no misspeculation occurred (line 1), the main thread starts to copy back the results. Specifically, the main thread goes through this parallel thread's live-in table (line 2) and identifies the modified live-in variable (line 3). The *WriteFlag* of a live-in variable is set when an access to this variable is a write. For each modified live-in, the main thread finds its latest value (line 4) and copy the value to the non-speculative state (line 4). The non-speculative address of a live-in is available because it is stored in the live-in table when the live-in is first accessed.

Misspeculation Check. The misspeculation check is performed by executing lines 6 to 11. In particular, when a live-in variable is committed, the main thread searches the live-in table of all other threads that are executing later speculative tasks. If this variable is found in another thread t's live-in table (line 7), then thread t is using a stale value because the

```
T.LTable: live-in table of a speculative thread T;
T.SepcFail: the flag indicating a failed speculation of T;
T.FailedAt[]: the live-in variables which cause the
speculation of thread T to fail;

1:  if (T.SpecFail != True) {
2:    foreach entry e in T.Ltable {
3:      if (e.WriteFlag == True) {
4:        val = GetLatestValue(e.SpecAddr);
5:        CopyBack(e.NonSpecAddr, val);
6:        foreach thread t executing a later speculative task {
7:          if (e.NonSpecAddr is in the t.LTable ){
8:            t.SpecFail = True;
9:            add e.NonSpecAddr into t.FailedAt[] ;
10:         } //endif
11:       } //end foreach
12:     }//end if
13:   }//end for each
14  }
15: else { //recovery
16:   pc = GetRecoveryPC(T.FailedAt[]);
17:   sid = GetRecoverySpaceID(T.FailedAt[]);
18:   ask thread T to reexecute from instruction pc using
      the latest values of live-in variables, and
      the values stored in the latest subspace whose SID
      is no more than sid for none-live-in variables ;
    }
```

Figure 5.5: Misspeculation Checks And Recovery.

value of this variable has just been updated. As a result, the *SpecFail* flag of t is set to *true* (line 8). This variable identified by its none-speculative address is also added into another thread attribute *FailedAt* (line 9). The variables stored in *FailedAt* will be used to identify the starting point of the reexecution during the misspeculation recovery.

Recovery. If the *SpecFail* flag of a thread t is set to *True*, then reexecution is required for the recovery. The *FailedAt* attribute of t indicates which live-in variables caused the speculation to fail. Since, when each live-in variable is first read, the instruction address of the read and the space ID before the read are recorded (Figure 5.4, line 2), the main thread can retrieve these two values of the first accessed live-in variable by calling another two auxiliary functions *GetRecoveryPC* and *getRecoverySpaceID* (lines 16-17). These two

values determine the staring point of the reexecution. The main thread now asks thread t to reexecute from instruction pc. During the reexecution, the latest value of every live-in variable is used. For other variables, their values stored in the latest subspace whose SID is no more than sid are used. This can be done by searching for the state history of each variable. In this way, the reexecution is consistent with the sequential semantics and all incorrect speculatively computed results are simply discarded. More importantly, all correct speculatively computed results are exploited because they are stored in different subspaces.

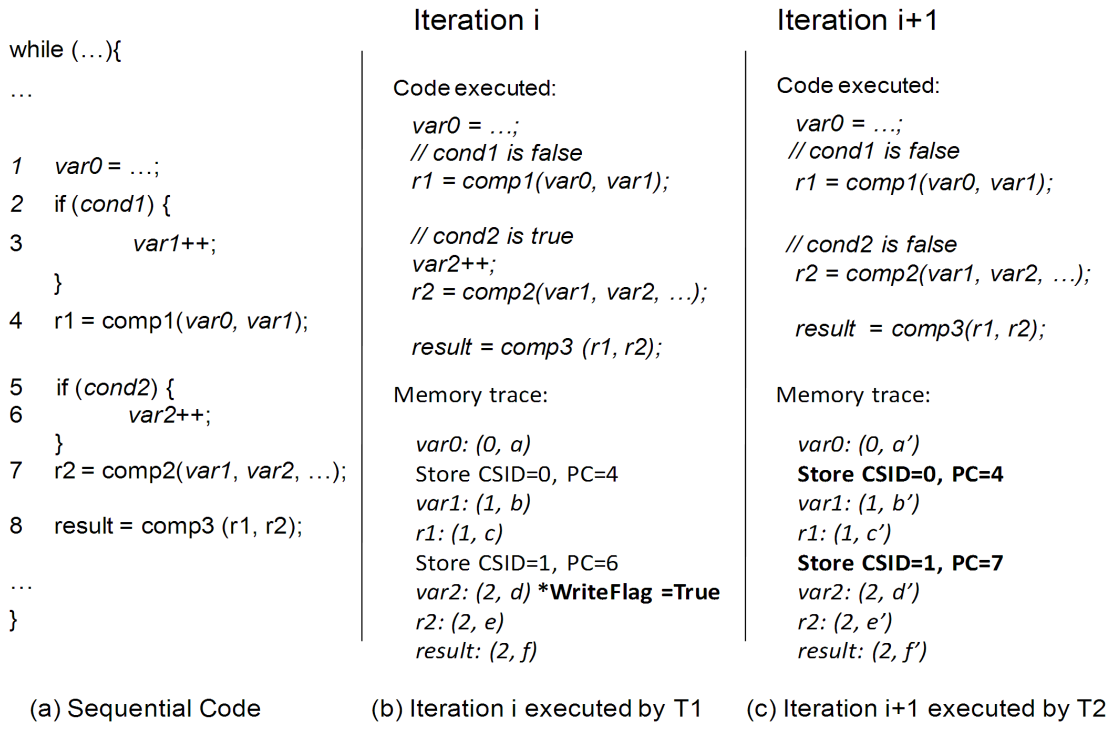


Figure 5.6: An Example Of Recovery.

Figure 5.6 shows an example. Figure 5.6(a) shows the sequential code where two live-in variables *var1* and *var2* can be observed. Consider iteration i and $i+1$, which are speculatively executed by thread *T1* and *T2* in parallel. The executed code and memory operations for these two threads are shown in figure (b) and (c) separately. During *T1*'s

execution, it updated *var2*, and hence, the *WriteFlag* of this variable is set to true. When the main thread performs the misspeculation check, it searches for all other threads that are currently using *var2* and sets the corresponding flag *SpecFail* to true (Figure 5.5). In this example, *T2.SpecFail* is set to true and *var2* is added to *T2.FailedAt*. When the main thread starts to examines *T2*'s results, it can realize *T2*'s speculation fails because of *var2*. Hence, it retrieves the CSID and PC of the recovery point that corresponds to *var2* (CSID=1 and PC=7 in this case), and sends the latest value of *var2* to *T2*. With such information, *T2* now can start the incremental recovery. In particular, it reexecuted the code from the instruction whose PC is 7, and uses the latest value of *var2* and the value stored in space 1 for all other variables.

5.3 Experiments

5.3.1 Experimental Setup

To verify the effectiveness of our techniques, 5 benchmarks are used as shown in Table 5.1. The first three columns of the table give the name, lines of source code, number of live-ins for these programs. For each benchmark, 3 different inputs are created which create cross-iteration dependences at low, medium, and high frequency respectively as shown by the last three columns. In particular, when a sequential version of these program is executed with these inputs, `input-low` generates less than 1% dependences, `input-medium` generates around 40% dependences, and `input-high` generates about 80% dependences. Experiments are conducted using these different inputs to evaluate the effectiveness of the approach.

In the experiments, the parallel version of every benchmark program is generated

Name	LOC	Number of Live-ins	Cross-iteration Dep. Frequency		
			input-low	input-medium	input-high
`197.parser`	9.7K	8	1%	49.3%	88.4%
`130.li`	7.8K	6	2%	40%	80%
`256.bzip2`	2.9K	9	0.5%	38.2%	79.2%
`255.vortex`	49.3K	11	0.5%	43.2%	81.3%
`CRC32`	0.2K	1	1.0%	40%	80%

Table 5.1: Benchmark Description.

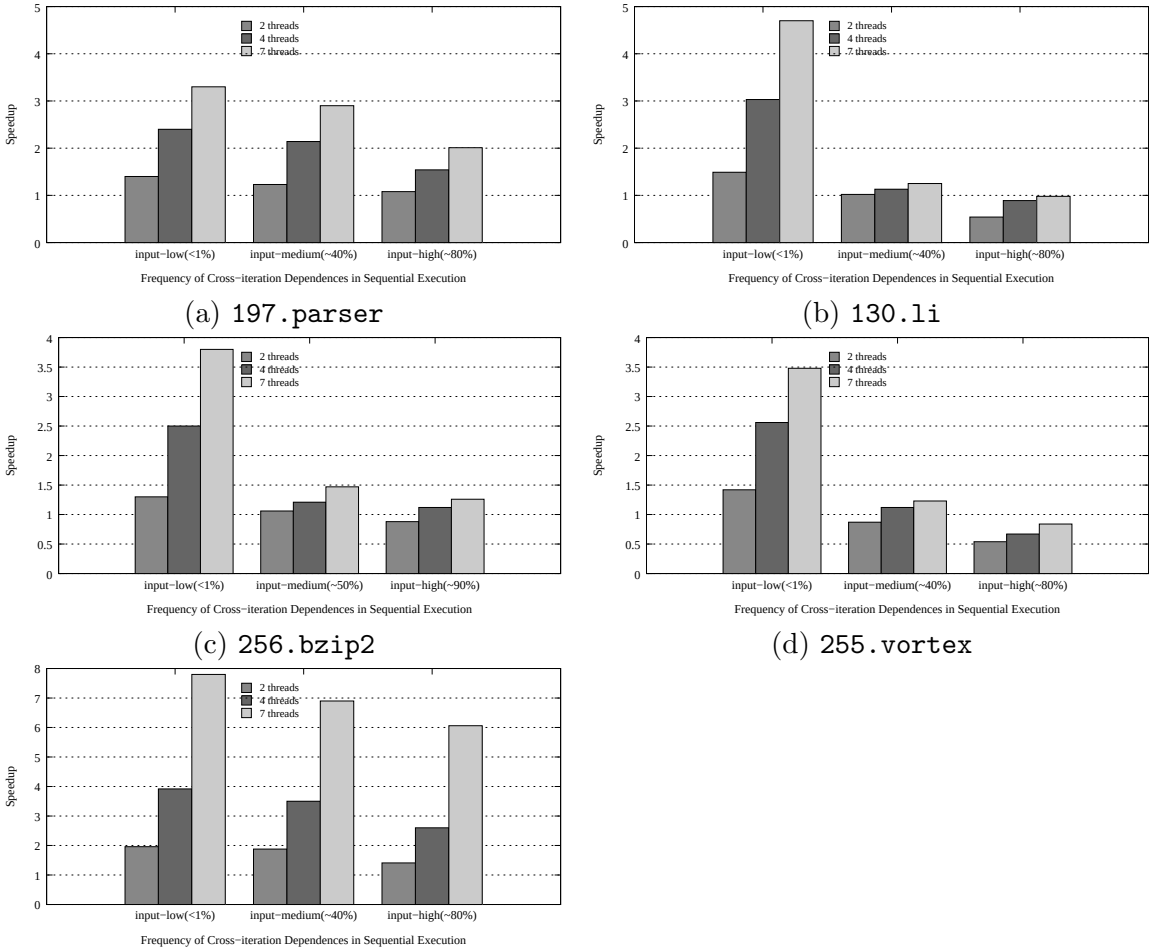
through a source-to-source transformation. Specifically, to support the space and thread decoupling scheme, every basic type in C (e.g., int, char etc.) is redefined as a new class in C++. The class contains not only the original type, but also the type of state history in parallel spaces. When a variable is declared of a class based type, its non-speculative storage is allocated during the compilation and its state history for each thread is created and dynamically maintained as shown in Figure 5.4. The speculative threads are created and controlled using the same template described in chapter 2, but rewritten in C++. The runtime system that detects the misspeculation and handles the speculatively computed result described in Section 3 are also implemented in C++.

All the experiments were conducted under CentOS 4 OS running on a dual quad-core Xeon machine with 16GB memory. Each core runs at 3.0 GHz.

5.3.2 Performance

Comparison With Sequential Execution. The first experiment conducted measured the performance and misspeculation rate when different inputs and different number of parallel threads were used. Figure 5.7(a)-(e) shows the result of each benchmark program. In these figures, the performances when using 2, 4 and 7 threads on different inputs are

shown on the left and the corresponding misspeculation rates on the right.



(a) `197.parser` (b) `130.li`

(c) `256.bzip2` (d) `255.vortex`

(e) `CRC32`

Figure 5.7: Execution Speedups – Baseline: Sequential Execution.

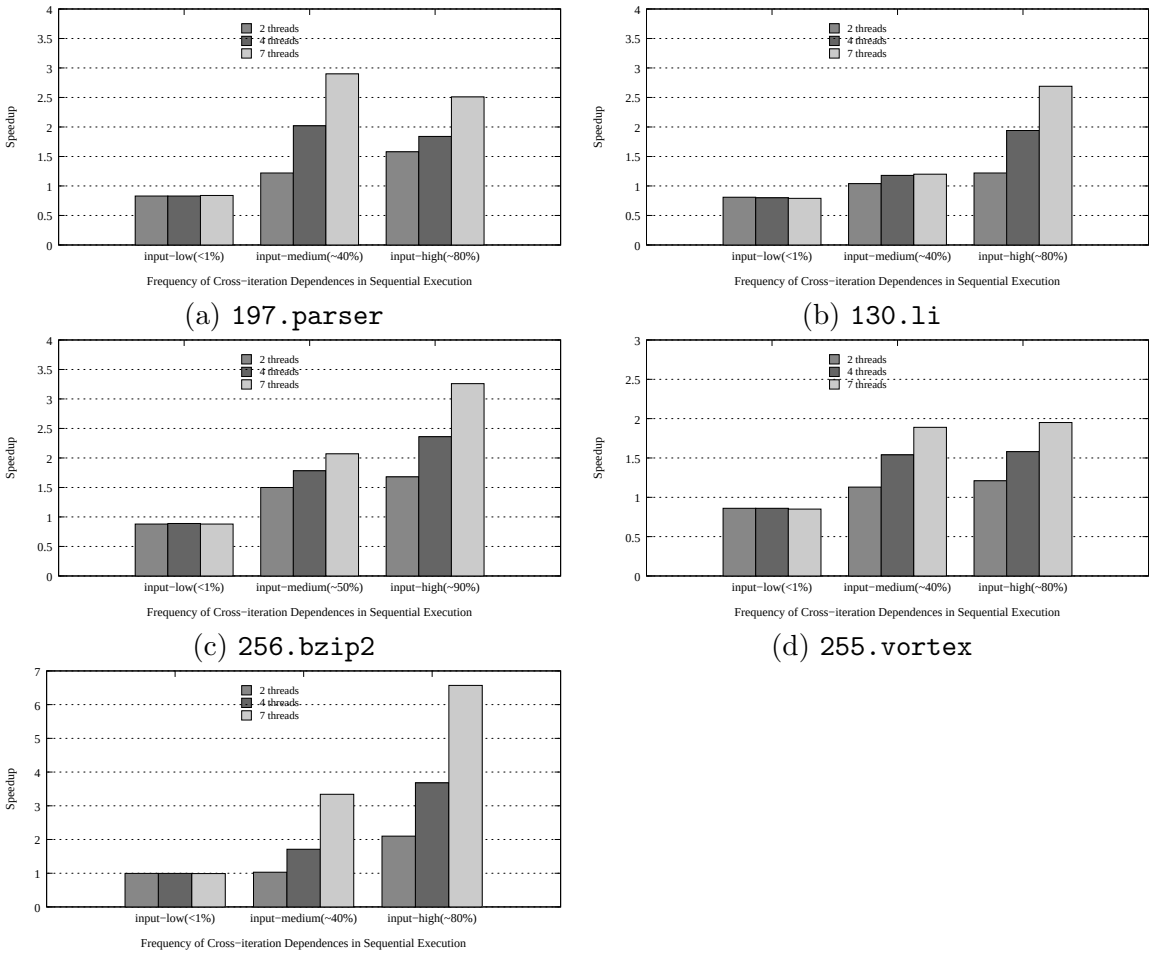
From these figures, one can see that for each benchmark running with the same input, spawning more threads always yields a better performance. When 7 threads are executed on `input-low`, all benchmarks obtained the best speedups which range from 3.7 to 7.8. When `input-medium` and `input-high` are used, performance gains can still be observed in most cases, but these gains drop dramatically for `130.li`, `255.vortex`, and `256.bzip`. Especially when running on `input-high`, the first two benchmarks experience slowdowns.

This is primarily due to the misspeculation recovery overhead. When a misspeculation occurs in these programs, a significant portion of the speculative results still have to be recomputed. In the case of `input-high`, such overhead outweighs the parallelism benefits.

However, for `197.parser` `CRC32`, speedups can be observed even when `input-high` is used. In these two programs, only one statement needs to be recomputed upon a misspeculation, and hence the performance is not limited by the misspeculation rate. The highest speedups achieve on `input-medium` and `input-high` are 2.9 and 2.0 for `197.parser`, 6.9 and 6.1 for `CRC32` respectively.

Comparison With Original Scheme. To show the advantage of this technique, experiments are conducted to compare incremental recovery technique with the original scheme where all speculatively computed results are simply discarded upon a misspeculation. The performance difference are shown in Figure 5.8. In particular, the execution time of the original parallelized version is used as the baseline in this experiment.

From the figure, one can observe that for all benchmarks running with `input-medium` and `input-high`, incremental recovery technique has much better performance than the original parallelized execution. It brings more benefit when more threads are used, because using more threads makes the misspeculation rate higher, which significantly slows down the original parallelized version. When incremental recovery is applied, only partial speculatively computed results need to be recomputed and the overhead of recovery can be largely reduced. As a result, it still can speed up the sequential program in most cases as shown in Figure 5.7. It should be noted that for `255.vortex` and `130.li` running on `input-high`, Figure 5.7 indicates that recovery technique cannot speed up the sequential



(a) `197.parser`

(b) `130.li`

(c) `256.bzip2`

(d) `255.vortex`

(e) `CRC32`

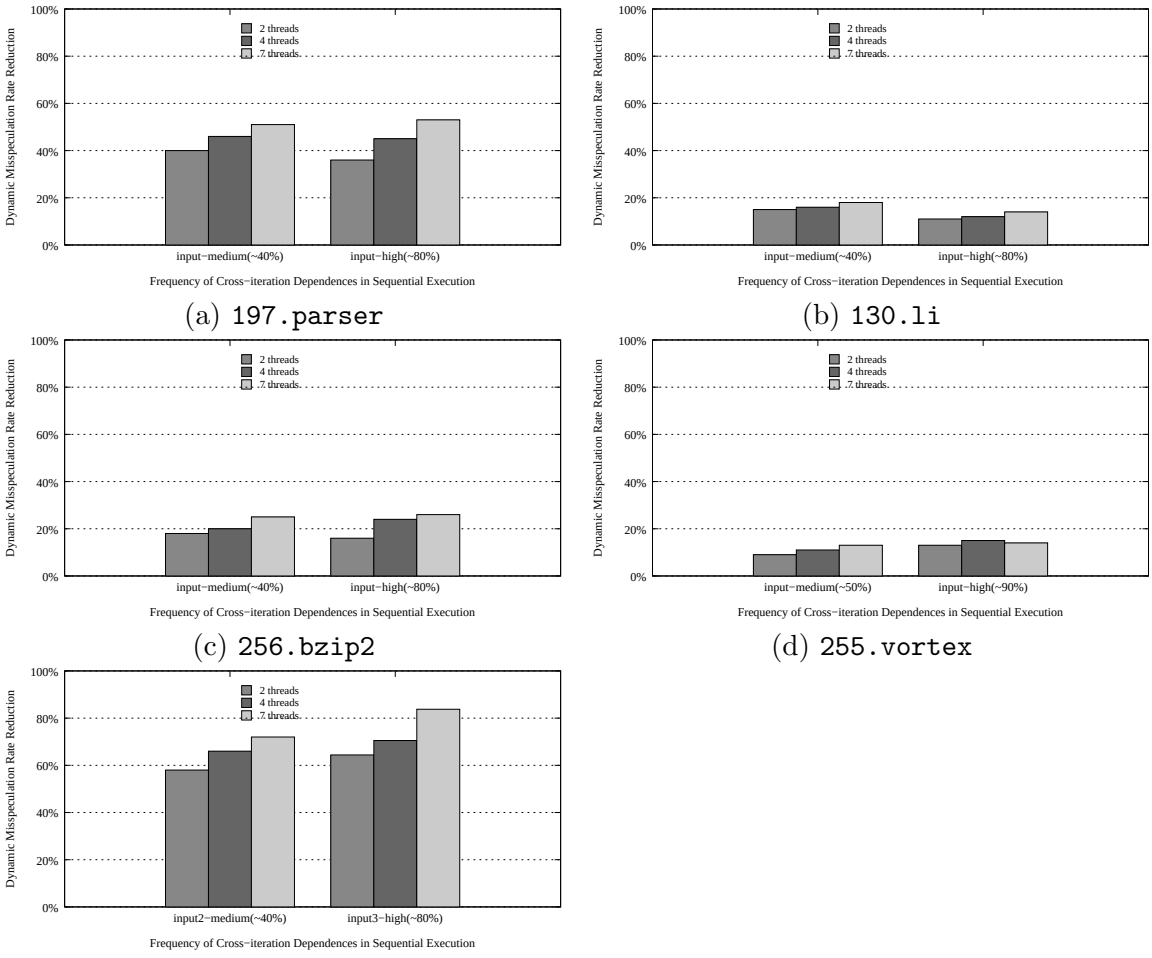
Figure 5.8: Execution Speedups – Baseline: Original Scheme.

executions. However, the performance of the original parallelized version is even worse. That explains why the speedups are still obtained in this experiment.

In the case of `input-low`, the performance of using incremental recovery is worse than the original scheme. The performance loss is about 15% on an average. This is because of the overhead of the recovery system. In particular, every load or store has to be performed on a state history – a list of the pairs of space ID and value. This is more expansive than operating on a single value. The results of the experiment shows that incremental recovery is much more effective if the misspeculation rate is very high and the original scheme is better otherwise.

Another reason for incremental recovery technique performing better on `input-medium` and `input-high` is that it reduces the number of misspeculations. If a variable is copied-in immediately before its first access instead of copied-in at the beginning, the chance of obtaining the correct value of this variable is increased as some earlier thread may just commit their latest values. Incremental recovery technique significantly increases this chance because it allows the recovery to be performed on a part of the entire computation. This speeds up the result-committing stage of every thread. Experiments are conducted to measure this effect. In particular, the misspeculation rate is collected for both incremental recovery technique and the original parallelization technique. The misspeculation rate reduction is then computed based on these two numbers and the results are shown in Figure 5.9. Note that the comparisons are made on `input-medium` and `input-large`.

From the figure, one can see that for `197.parser` and `CRC32`, up to 50% and 85% misspeculations are reduced respectively. As mentioned earlier, in these two programs, only one statement needs to be recomputed during the recovery and the statement defines



(a) `197.parser`

(b) `130.li`

(c) `256.bzip2`

(d) `255.vortex`

(e) `CRC32`

Figure 5.9: Misspeculation Rate Reductions.

the live-in variable. With incremental technique, the live-in variable can be committed faster, which increases the chance of later threads getting the correct version. As a result, performances of these two benchmarks are improved by the technique very well. For other programs, the number of reexecuted statements during the recovery far greater, and hence the observed misspeculation reduction is around 20% for `256.bzip`, 10% for `130.li`, and 10% `255.vortex` on an average.

5.3.3 Overhead Analysis

Execution Time Breakdown. The execution time spent by each parallel thread has been broken down into three different categories: *communication*, *recovery* and *speculative computation*. Figure 5.10 shows the results. They are measured and averaged across all 7 threads.

According to the figure, the time spent on the communication is only a small portion for all programs regardless of the input. This means all parallel threads are busy in performing computations. For all parallelized programs running on `input-low`, each thread spends most of the time on speculative parallelization. When the input changes, the more misspeculations take place, more time is spent on the recovery. In the case of `256.bzip2`, `130.li` and 255.vortex running on `input-high`, half of the time is spent on reexecuting the speculative computation.

Figure 5.11 shows the time breakdown for the main thread. Three categories are considered: *communication*, *Misspeculation Checks* and *copy-out* and *computation*. Different from parallel threads, the main thread spent a large portion of the execution time on communication. Another significant portion is spent on misspeculation checks and result-

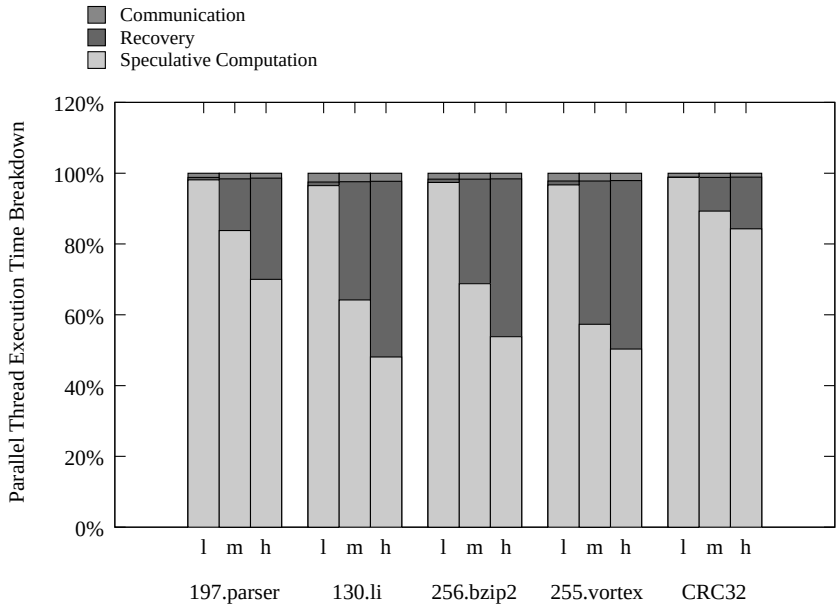


Figure 5.10: Time Breakdown: Speculative Threads.

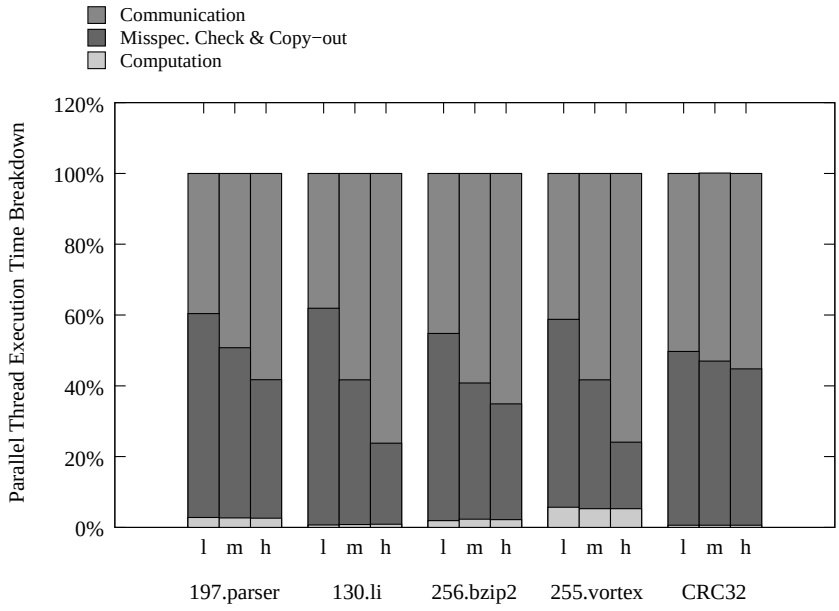


Figure 5.11: Time Breakdown: Main Thread.

committing. When different input is used to create more misspeculations, one can observe that communication time increases, because parallel threads fail more frequently and have to reexecute unsuccessful tasks. As a result, the main thread has to spend more time on waiting for the correct results to be produced.

Space Overhead. Maintaining multiple P spaces for one thread consumes more space. An experiment is conducted to monitor the peak value of memory consumption when different number of parallel threads are used during the execution. The memory used by the corresponding sequential program is considered as the baseline. Figure 5.12 shows the space overhead. As expected, using more threads consumes more space because each variable accessed by a parallel thread takes more memory space in this technique. When 7 parallel threads are used, the largest overhead is observed which ranges from 3.2 to 5.1 across all benchmarks. Note that the data in this figure is collected when `input-high` is used. It is observed that space overhead is not sensitive to the misspeculation rate. The results of using the other two different inputs are similar.

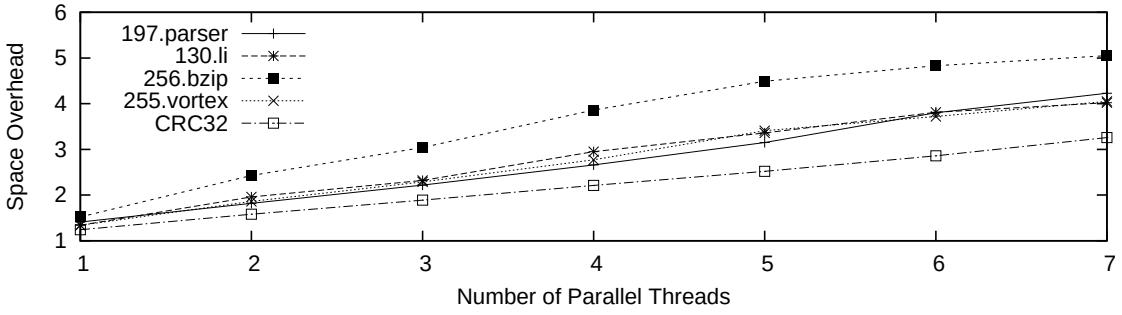


Figure 5.12: Space Overhead.

5.4 Summary

This chapter presented an adaptation of CorD to enable *incremental recovery* for dealing with high misspeculation rate. The idea is to decouple the creation of each parallel thread from the creation of its P spaces. Using multiple spaces during a speculative execution allows the speculative computational results to be stored and reused when a misspeculation occurs. Experiments show that this technique not only improves the performance but it also reduces misspeculation rate compared to original scheme.

Chapter 6

Applying CorD In The Presence Of Dynamic Data Structures

A dynamic data structure consists of large number of nodes such that each node contains some data fields and pointer fields. The pointer fields are used to link together the nodes in the data structure (e.g., link lists, trees, queues etc.). Such data structures are also called dynamic data structures because the shape and size of the data structure can change as the program executes. Size of the data structure changes as nodes are added or removed and changes in link pointers can further change the shape of the data structure. The memory for each node is dynamically allocated from the heap when the node is created and freed by returning it to the heap when the node is eliminated. Dynamic data structures are extensively used in wide range of applications. The applications considered in this chapter are C programs with following characteristics:

- each node is allocated and deallocated through explicit calls to library functions *malloc* and *free*; and

- nodes are accessed through pointers variables that point to the nodes such as pointer fields that link the nodes in the data structure.

Applying CorD speculative parallelization technique in applications that intensively use dynamic data structures is much more challenging than in those using scalar variables or static data structures such as arrays. In particular, the following challenges need to addressed in the presence of dynamic data structures:

- *What to Copy-In ?* Complexities of pointer analysis makes it difficult to identify the portion of the dynamic data structure that is referenced by the speculative computation. Conservatively copying the entire data structure may not be practical when the size of the data structure is very large.

- *How to Copy-Out ?* The copying of updated data structure from speculative state to non-speculative state is made complex due to the changes in the shape and size of the dynamic data structure that may be made by the speculative computation.

- *How to handle internal pointers ?* In addition, both copy-in and copy-out operations must contend with the need to translate pointers internal to dynamic data structures between their non-speculative and corresponding speculative memory addresses.

This chapter addresses all of the challenges outlined above. The organization of this chapter is as follows. Section 6.1 describe the challenges in detail by using an illustrative example. Section 6.2 proposes three techniques *copy-on-write*, *heap prefix* and *double pointers* to address them. Several optimizations are proposed to further improve the performance in section 6.3. The experimental results are presented in section 6.4 and followed by the summary in section 6.5.

6.1 Challenges For Dynamic Data Structures

Given a parallelized computation which consists of a non-speculative main thread that spawns multiple speculative threads, in CorD model state separation is achieved by performing speculative computations in separate memory space. While for array or scalar variables the separation can be simply achieved by creating a copy of such variables in the speculative thread, achieving state separation for programs using dynamic data structures poses many challenges. A dynamic data structure may contain millions of nodes (e.g., the program *Hash* in the experiments creates 3 million nodes at runtime). This leads to a large overhead due to copying operations, mapping table accesses, and misspeculation checks. Moreover, need for address translation arises because a node may have many pointers pointing to it and after the node has been copied, accesses via these pointers must be handled correctly.

For a program using dynamic data structure, four types of changes to the dynamic data structure can be encountered:

- pointer fields in some nodes are modified causing the shape of the data structure to change;
- a new node is created and linked to the dynamic data structure;
- an existing node is deleted from the dynamic data structure causing the size of the data structure to change; and
- values of data fields in some nodes are modified.

Figure 6.1 shows an example where all the above changes are encountered. In this

```
typedef struct node{
        int key;
        int val;
        struct node *next;
}
NODE *head;

while (...) {
        ...
1:      NODE *tmp = find_key(head, key);
2:      if (tmp!=NULL and tmp!=head) {
3:          NODE *prev = get_prev_node(head, tmp);
4:          prev → next = tmp → next;
5:          tmp → next = head;
6:          head = tmp;
7:      }
8:      else {
9:          if (!tmp) { //update the lru queue
10:             //insert the new node
11:             NODE *n = (NODE *)malloc(sizeof(NODE));
12:             n→ key = ...;
13:             n→ val = ...;
14:             n→ next = head;
15:             head = n;

                //delete the least-recent used node
16:             NODE *m = get_second_last_node(head);
17:             free(m→ next);
18:             m→ next = NULL;
19:         }
20:     }
21:     if (writeflag)
22:     {
23:         head→val = ...;//modify data
24:     }
25:     ... = head→val;
26:     ...
}
```

Figure 6.1: Least Recent Use Buffer.

example, a link list is used to implement a least-recent-use (LRU) buffer. A LRU buffer has a fixed length and it buffers most recently used data. When data is requested, any matching elements in LRU is first searched (line 1). If a match is found and the element is not in the front of LRU buffer, this element is moved to the front by adjusting the pointers (lines 2-7). If no match is found, a new node is created, insert it in the front and delete the last node (lines 8-20). After the requested data is put at the front, a flag is checked to see

if the data needs to be modified (lines 21-24). Finally the data in this buffer is read (line 25). Assume that the requested data is frequently at the front of LRU buffer and branches at lines 2, 8 and 21 are rarely taken. Thus, iterations in the *while* loop (lines 1-25) can be speculatively executed in parallel.

When a parallel thread is created, all pointers (*head*, *tmp*, *prev*, *m* and *n*) have their own local storage in the corresponding speculative space. Note that *head*'s local storage (denoted by *head'*) contains the content of *head*, as it is defined outside the parallelizable region. **P* is used to denote a node pointed to by pointer P. Next we describe the challenges via this example.

Overhead Challenge. As one can see the function call *find_key* at line 1 traverses all nodes in the LRU buffer. If the original CorD model is used, and the LRU buffer contains a million nodes at runtime, then the overhead of this traversal is prohibitively high. First, the copying overhead is large as all these nodes can be potentially modified by speculative thread and hence are copied into speculative state. Second, the mapping table that maintains correspondence between addresses in non-speculative and speculative memory is large, because for every copied node, a mapping entry has to be maintained. A large mapping table leads to an expensive lookup and update. Last but not least, the version table is large. In the original CorD model, a global version number of each node is stored in the version table and used in performing misspeculation checks. In particular, if a node is modified, its global version is compared with its local version stored in the mapping entry. Searching for the global version number of a node in a large version table requires time. Besides, the search has to be done for all modified nodes. Finally, finding modified nodes from a large

mapping table is very time-consuming. Consequently, the misspeculation check is very time consuming.

Address Translation Challenge. A node in a dynamic data structure is allocated on the heap at runtime. Its address is stored in one or several pointer variables and its access is performed through such pointers. This creates the address translation problem. In particular, when a node is copied into speculative state by a copy-in operation, all pointers in speculative thread holding its address and being used in the computation must change their content to the address of the copied node accordingly. For example, in Figure 6.1 line 5, a parallel thread uses *head'* which is holding a non-speculative state address as it is a copy of *head*. If the node **head* has been copied during the execution of function *find_key*, the value of pointer *head'* has to be changed to the address of the head node's local copy, and then assign this new value to $tmp \rightarrow next$. This requires comparison of each pointer being accessed (in this example *head'*) with the addresses stored in the mapping table. Similarly, when a node is copied back to non-speculative state by a copy-out operation, all pointers containing its current local copy address need to hold its non-speculative state address now. In Figure 6.1, when line 6 is executed, *head'* points to the node **tmp*, a local copy of some node in the non-speculative state. Therefore, when the value of *head'* is copied back to *head*, the address needs to be changed to the address of that node. This can be done by consulting the mapping table with the address stored in *head'*.

If the branch at line 8 is taken, copying out node **n* is a challenge. At line 14, the *next* field of node **n* is assigned with the address of head node's local copy. However, when committing the result, the node **n* is represented as the starting address and length.

Therefore, it is impossible to find which part of *n* is the starting address of the *next* pointer field, and thus, cannot translate the address. One solution might be to store the address of *next* pointer in the mapping table, but again, this may lead to an even larger mapping table as one node may contain multiple pointer fields that are modified. Similarly, when line 17 is executed, we also need the address translation so that the correct node in non-speculative state is deallocated. In the case of data field modification (line 23), however, there is no need for address translation.

6.2 Adapting CorD For Dynamic Data Structures

6.2.1 Copy-On-Write Scheme

To address the overhead challenge, primarily one must find a way to reduce the number of nodes that are copied to speculative state. Therefore, the use of *copy-on-write* scheme is proposed to limit the copying to only those that are modified by the speculative thread. A node in non-speculative state is allowed to be read by speculative threads. It is copied into a thread's speculative state only when it is about to be modified. The copying is implemented through an *access check* – a block of code inserted by compiler to guard every node reference via a pointer. Based on the type of reference, read or write, *access check* code differs.

Write Access. Upon a write to a node, the *access check* determines if the node is already in the speculative state. If this is the case, the execution can proceed. Otherwise, the *access check* concludes that the address belongs to a node in non-speculative state. In this case the speculative thread must determine if this is the first write to the node and thus the

node must be copied into the speculative space. However, if this is not the first write to the node, then the node has already been copied into speculative state. Thus, the address being referenced in the non-speculative state has to be translated into the corresponding address in the speculative state. This translation is enabled by ensuring that the mapping table is updated by creating entries for copied nodes. In other words, the access checks consult the mapping table to determine if the current pointer refers to a node in speculative state or non-speculative state.

Read Access. Upon a read to a node, the *access check* allows the execution to continue if the node is in speculative state. However, if the node is in non-speculative state, the *access check* stores the thread task ID for this node indicating when the node has been read. After this step, the execution can proceed. The thread task ID is an integer maintained by each thread. It is initially zero and incremented by one every time the thread is assigned a task to perform by the main thread. As one will see shortly, this information is used during misspeculation checks. It is worth noting that in this copy-on-write scheme, if a node is only read by a speculative thread, it never has an entry in the mapping table. In other words, all mapping entries contain nodes that have been modified by the speculative thread.

An Example. Applying this scheme to the example in Figure 6.1, one can observe the following two advantages. First, there is no need to copy every node in a dynamic data structure into speculative state when function *find_key* is executed. Therefore, the size of the mapping table is reduced. Second, the need for address translation is greatly reduced. In particular, if node **head* is never updated during execution, then no copy of this node is made. Consequently, the pointer *tmp→next* at line 5 and *n→next* at line 14 get the correct

non-speculative address of this node without address translation. For line 17, the address stored in $m \rightarrow next$ can also be simply marked as deallocated, instead of making a copy of a node $*(m \rightarrow next)$ and then translating the address.

6.2.2 Heap Prefix

Although the copy-on-write scheme can reduce the size of mapping table, the access and update of this table may still impose large overhead on the parallel execution. In particular, for each heap access, the *access check* needs to consult the mapping table to see if a node has been copied or not. This requires a walk through the entire table. Similarly, the *misspeculation check* needs to search the version number of each modified node in the version table. This requires traversing the table multiple times. To efficiently perform the access checks and misspeculation checks meta-data is associated with each node that tracks certain information related to accesses of the node. This meta-data is called *heap prefix*. Next sections describe the details of heap-prefix and show how it is effective in reducing the overhead of using mapping table and version table.

For each memory chunk allocated on the heap, $2 * n$ additional bytes is allocated in front of it where n is the total number of speculative threads. These bytes represent the

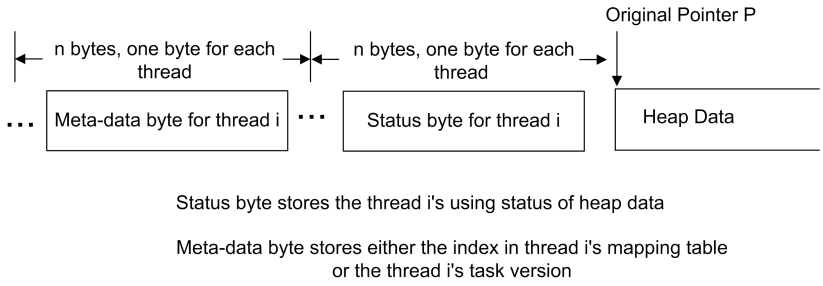


Figure 6.2: Heap Prefix Format.

heap prefix which is used to store important information to assist in access checks. The format of the heap prefix is shown in Figure 6.2. The first n bytes immediately before the program's original heap data are the *status bytes*. The additional n bytes are *meta-data bytes*. In the status byte, byte i represents the status for speculative thread i and it can represent four different possible status values. Status NOT_COPIED means the heap data has not been copied into thread i's speculative state. Status ALREADY_COPIED means the heap data has been copied into thread i's speculative space, and the index of this entry in thread i's mapping table is stored in the corresponding meta-data byte. Status ALREADY_READ means the heap data has been read by thread i, and the corresponding meta-data bytes stores the *task ID* of thread i. Status INTERNAL indicates that the node is already in the speculative state. Therefore, status NOT_COPIED, ALREADY_COPIED and ALREADY_READ only appear in heap elements of non-speculative state and status INTERNAL only appears in heap elements in speculative state. In the meta-data bytes, meta-data byte i stores either a index number of the mapping table of thread i, or the *task ID* of thread i.

Note that one can put the meta-data associated with each node in a different place and use hash function to locate it [73]. However, after the *hash-based* solution is applied, it is observed that it caused over 6x slowdowns for the benchmarks used. The reason is that a hash based lookup requires the execution of a hash function, which takes more time than performing a simple offset calculation. Thus, large number of lookups make the hash based solution yield visible slowdowns.

```
1:  type = access type, READ or WRITE;
2:  p  = the pointer holding the starting address
         of the node being accessed;
3:  len  = the size of the node being accessed;
4:  s = thread i's status at *p;
5:  m = thread i's meta-data byte at *p;

6:  if (s == NOT_COPIED)
7:    if (type == READ) {
8:        s = ALREAD_READ;
9:        m = task_ID;
10:   }
11:   else { // type == WRITE
12:       s = ALREADY_COPIED;
13:       pointer q = make_copy(*p);
14:       set thread i's status at *q to INTERNAL;
15:       index = update_mapping_table(p, q, len);
16:       set thread i's meta-data byte at node to index;
17:       p = q;
18:   }
19: }
20: else if (s == ALREADY_COPIED) {
21:   index = thread i's meta-data byte at node;
22:   p = get P address from mapping table entry index;
23: }
24: else if (s == ALREADY_READ) {
25:   if (type == WRITE) {
26:       s = ALREADY_COPIED;
27:       pointer q = make_copy(*p);
28:       set thread i's status at *q to INTERNAL;
29:       index = update_mapping_table(p, q, len);
30:       set thread i's meta-data byte at node to index;
31:       p = q;
32:   }
33: }
34: else  { //s == INTERNAL
35:   ; //do nothing
36: }
```

Figure 6.3: Access Checks.

Implementing Access Checks. With the status bytes and meta-data bytes in the heap prefix, the *access check* for a heap node access in thread i can be implemented as shown in Figure 6.3. Thread i's status s in the node's prefix is examined and following actions are taken.

If s is NOT_COPIED and the access is a read, s is updated to ALREADY_READ and the task ID of thread i is stored in the meta-data byte i (lines 7-10). If the access is

a write, s is updated to ALREADY_COPIED. A new local copy of the node is then created with the corresponding status byte to be set to INTERNAL. Next, a mapping entry is added into the mapping table to reflect this copy operation and the index of the entry is stored in the meta-data byte i. Finally, the pointer points to the newly created node (lines 11-18).

If s is ALREADY_COPIED, then that means the node has been copied; thus, address translation is needed. Fortunately, the mapping entry can be quickly located through meta-data byte i and the pointer only needs to be adjust to point to the address of the local copy (lines 20-23). Finally, if s is ALREADY_COPIED and the access is a write, then we perform the copy-in operation as when s is NOT_COPIED (lines 25-32). Otherwise, the access is a read or s is INTERNAL. In both cases, no further actions are required.

Figure 6.4 shows an example of using heap prefix to perform access checks. First, assuming the node is allocated at 0xA in non-speculative state (D space), consider the execution of speculative thread T3. Before any reference to this node in T3, the status byte for T3 shows that the node has not been copied into its speculative state (P space) yet (as shown on the left).

Suppose there is a write to this node during the execution, the access check makes a copy of this node as it sees the status is NOT_COPIED. Therefore, the following actions are taken. A new node is allocated at 0xB in P space and initialized with the original node value; a mapping entry is created in the mapping table (its index is x); T3's status of the original node is changed to ALREADY_COPIED indicating that this node has been copied into speculative memory, and the corresponding meta-data byte of T3 stores the index of mapping entry (x); T3's status of the copied node is set to ALREADY_COPIED which means this node is already in speculative state. In the later execution of T3, if the original node is

accessed through some other pointers, the access checks can easily translate those pointers to point to 0xB by looking at the heap prefix and the x-th mapping entry. Similarly, if the node starting at 0xB is about to be accessed by a pointer, the access check code confirms the access to be valid by simply looking at the prefix of this node. After committing T3's result, the local copy of the node is deallocated and T3's status byte and meta byte in the prefix of the original node is reinitialized to zero (shown on the right).

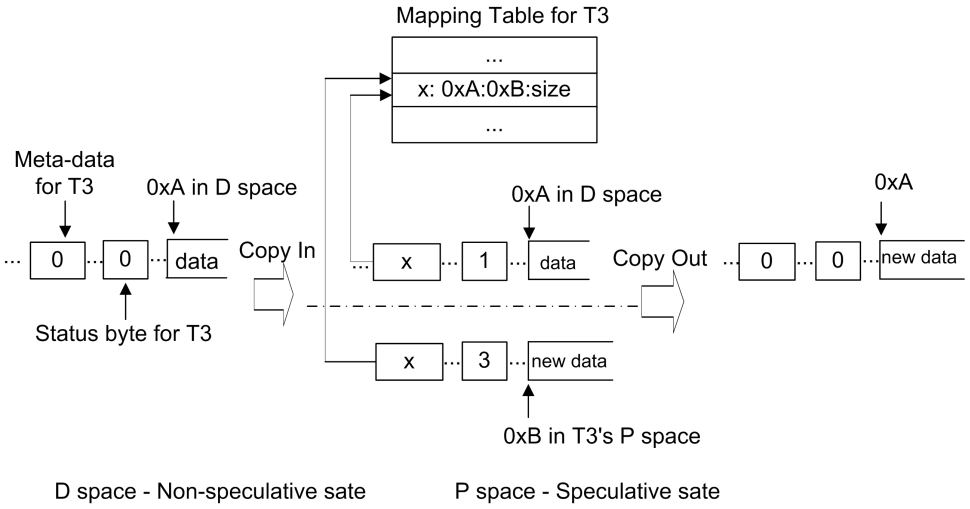


Figure 6.4: An Example Of Heap Status Transition.

In summary, there are two main advantages of using heap-prefix to implement access checks. First, the status byte can tell the access checks whether or not a node has been copied. Second, the meta-data bytes allow the speculative thread to find the mapping entry in O(1) time, which speeds up the process of address translation for copy-in operations.

Implementing Misspeculation Checks. To determine if speculation is successful, misspeculation checks have to be performed. The main thread maintains a version number for each variable in a version table. When a speculative thread uses a variable, it makes note

of the variable's current version number. When the results of a speculative thread are to be committed to non-speculative state, misspeculation check is performed by the main thread. The main thread ensures that the current version number of a variable is the same as it was when the variable was first used by the speculative thread. If no mismatch is found, the speculation is considered as successful. Otherwise, misspeculation occurs and the results are discarded. This is because speculative thread must have prematurely read the variable. This method worked effectively for array variables and scalar variables.

In a program using dynamic data structures, however, the number of nodes in the structure can be very large, and hence the version table can become very large. Consequently, searching a node in the version table can impose large runtime overhead. Now that the heap prefix can tell how the node is being used by other threads at any time, this information can be exploited to perform the misspeculation check for the dynamic data structure without using a version table.

The key idea of the approach is that when the main thread checks the result of thread i, it also checks if any other thread is using a node modified by thread i. If so, that thread's execution fails as it is working on an incorrect speculatively read value. This method works because the main thread commits results of speculative threads' in a sequential order.

Figure 6.5 shows the algorithm. For each node mapping entry e in the mapping table of thread i, the main thread examines other thread's status byte of the node starting at *e.addr_non-spec*. If another thread's status byte is ALREADY_COPIED, then speculation of that thread fails (line 5-6). Note that status ALREADY_COPIED means the node has been modified and hence has an entry in the mapping table. When the node is copied back,

```
1:  if (spec[i] == FAIL)
2:    return FAIL;
3:  for each node mapping entry e in mapping[i] {
4:    for each thread j's status on e.addr_non-spec s[j] {
5:      if (s[j] == ALREADY_COPIED)
6:        spec[j] = FAIL;
7:      if (s[j] == ALREADY_READ
          and meta-data[j] == taskID[j])
8:        spec[j] = FAIL;
9:    }
10: }
11: return SUCCESS;
```

Figure 6.5: Misspeculation Checks For Heap Objects.

the status byte and meta-data byte for thread i is reset to zero.

If the main thread finds that another thread j's status byte is ALREADY_READ, then situation may be more complex because the status ALREADY_READ can be set during the current work assigned to thread j or during a previously assigned work to thread j. The latter case happens if in an earlier iteration, thread j only read this node. Therefore, there was no entry in the mapping table and hence the status byte and meta-data byte were not cleared. However, these two cases can be distinguished using the *task ID* stored in the meta-data byte j. The main thread only needs to check if the meta-data byte j's value is equal to thread j's current task ID. If they are the same, then thread j's execution is marked as failed.

Note that there exists a data race between checking thread j's status byte and setting the byte. However, this data race is harmless as it does not affect correctness. This is because we require the main thread to commit the result before checking other threads' status bytes and the speculative thread to update the status byte before accessing the node. Figure 6.6 shows an example where step 2 and 3 are clearly racing against each other. If step 2 reads the value after step 3, then thread j's execution is marked as failed which is

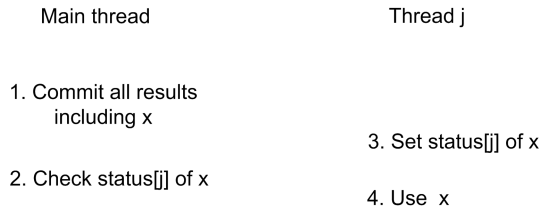


Figure 6.6: A Possible Data Race During Misspeculation Check.

correct, because thread j may read the old value of x (step 4 happens before step 1). If step 2 reads the value before step 3, thread j's execution is not be marked as failed. This is also correct because thread j is using the latest value of x.

In summary, the advantage of using heap-prefix in implementing misspeculation checks is that status bytes is used to identify any two threads that are accessing the same node. This eliminates the requirement of maintaining a version number of each node.

Discussion On Meta-data Bytes. For each thread, one byte is used to store the meta-data, i.e., the index of the mapping entry or the *task ID*. Since one byte can at most hold 256 numbers, using one byte may impose some limitations in certain situations and hence needs to be discussed.

First, if the meta-data byte of a thread is used to store mapping entry indexes, the mapping table size must have less than 256 entries to avoid overflow. This means for each task, the number of modified node should be less than 256. In some cases, this assumption may not be true. If a mapping table overflow occurs, the corresponding task should be considered as failed to ensure the correctness. However, having too many overflow events means that using one byte is not enough and performance loss results. To solve this problem, profiling can be used to find out how many nodes are modified in each task on an

average and choose multiple bytes to store the indexes for each thread if necessary.

Second, if the *task ID* is stored in one byte, the number may also wrap around and cause a problem in a very extreme case. Specifically, when thread i writes a node at iteration a and thread j reads the same node at iteration b where $b \equiv a \pmod{256}$ and it never uses the node after that, thread j may be incorrectly marked as failed. However, even if this extreme case arises, the correctness is not affected at all – a false misspeculation is reported and the computation is unnecessarily repeated.

6.2.3 Double Pointers

As described earlier, the need for address translation can be reduced by using copy-on-write scheme. The address translation is needed only for a copied node. It can be done by using the status byte and meta byte during the copy-in operation. However, the address has to be efficiently performed translation for a pointer during a copy-out operation. This is because nodes being copied-out may have many pointer fields.

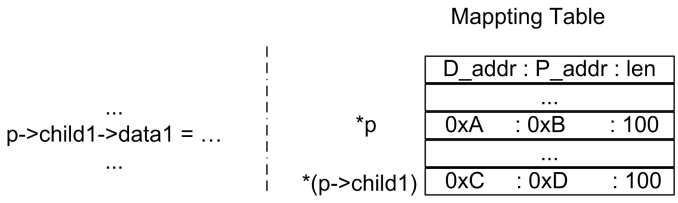


Figure 6.7: Internal Pointer.

Figure 6.7 shows a simple example involving two nodes pointed to by pointers p and $p{\rightarrow}child1$. The node $*p$ has already been copied in from non-speculative address 0xA to speculative address 0xB as shown in the mapping table. Now when the assignment is about to execute, the node $*(p{\rightarrow}child1)$ is copied from 0xC into 0xD. The value of $(p{\rightarrow}child1)$

also changes to 0xD so that the assignment takes effect on the local copy starting at 0xD. At the time of committing results to non-speculative state, the main thread must scan the mapping table to copy these two nodes back to the non-speculative state. However, the value in *(p→child1)* is still 0xD and of course it needs to be translated to 0xC. To do this, one way would be to locate the field by adding it to the mapping table, and then comparing the value in this pointer with all P_addr in the mapping table. This process entails significant overhead in programs using dynamic data structures as all nodes are linked into the structure through pointers.

To tackle the above problem, an augmented pointer representation – *double pointers* is presented. For each pointer variable *p*, the compiler allocates 8 bytes. 4 bytes for the non-speculative state address (denoted by *p.D_addr*) and 4 bytes for speculative state address (denoted by *p.P_addr*). When a node is allocated by the main thread and pointed to by *p*, its starting address is stored in *p.D_addr*. When a node is created by a speculative thread and pointed by a pointer *p'*, the starting address of this node is stored in *p'.P_addr*. If the node is created as a part of the copy-in operation, *p'.D_addr* is set to be *p.D_addr* (assuming *p'* is the local copy of *p*). Otherwise, *p'.D_addr* is set to be *p'.P_addr*. For any reference of a pointer *p*, if it is in the main thread, then *p.D_addr* is used; otherwise *p.P_addr* is used. For a pointer assignment $p = q$, all 8 bytes are copied.

With this scheme, the problem can be easily resolved as shown in Figure 6.7. Consider now the illustration in Figure 6.8 where A denotes **p* and B denotes **(p→child1)*. As one can see in Figure 6.7(a), before the assignment, A's local copy keeps B's non-speculative address in the *D_addr* field. Its *P_addr* field has been set to NULL. After the assignment, a local copy of B has been created and pointed by the P_addr field of pointer

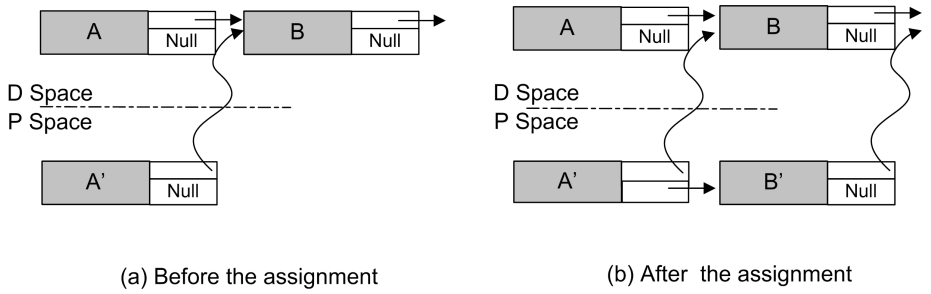


Figure 6.8: Double Pointer.

p→child1. When these two nodes are copied back, there is no need to go to the mapping table for address translation. Instead, they can be directly copied back, as A still holds B's address in the non-speculative state.

The double pointers scheme also ensures the correctness when the shape of a dynamic data structure changes due to the update of some pointer fields in the computation. Let us consider the example in Figure 6.1 again where three possible pointer related changes are encountered.

Changing the Shape. If the branch at line 2 is taken, the node pointed to by *tmp* is moved into the front of the buffer and pointed by *head*. Figure 6.9 shows the process of this shape transformation under the scheme. As shown in Figure 6.9(a) Before executing line 4, the parallel thread has two local pointers *prev'* and *tmp'*, which are the copies of pointers *prev* and *tmp* respectively. Since the statement at line 4 updates the node pointed to by *prev'* (node B), a local copy of node B is created through the access check, and the *P_addr* field of *prev'* points to this copy. After line 4, the *next* pointer field of node *B'* contains the address of node *D*.

When executing line 5 as shown in Figure 6.9(b), a local copy of node *C* is created

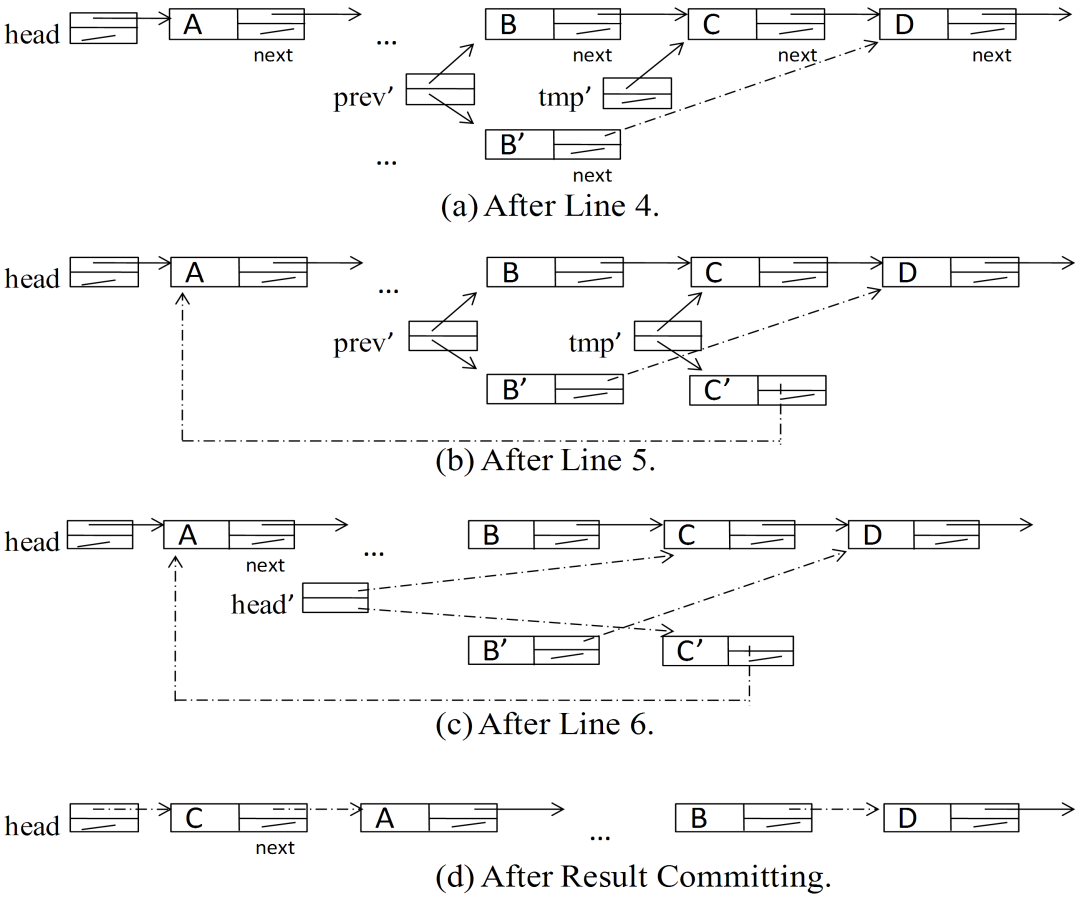
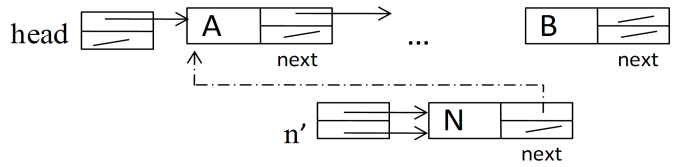
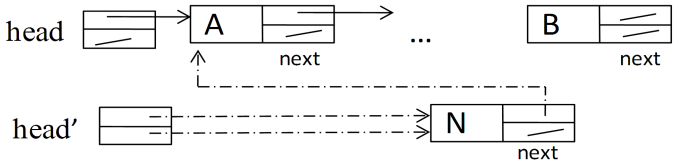


Figure 6.9: Changing The Shape Of A Dynamic Data Structure.

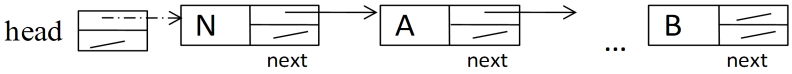
and pointed by the *P_addr* field of pointer *tmp'*. After this statement, all 8 bytes of the original pointer *head* are copied into the *next* pointer field in node *C'*. Thus, the *D_addr* field in *next* contains the address of node *A*. The statement at line 6 creates a local copy of pointer *head*. As shown in Figure 6.9(c), after copying the contents of pointer *tmp'*, its *D_addr* field has the address of node *C* and *P_addr* has the address of node *C'*. Finally, copy-out operations in the result-committing stage change the content of pointer *head* and the *next* pointer field of node B and C. The updated pointer is represented by the dash line in Figure 11(d), which reflects the update to the LRU buffer.



(a) After Line 14.



(b) After Line 15.



(c) After Result Committing.

Figure 6.10: Adding A New Node To A Dynamic Data Structure.

Adding A New Node. If the branch at line 9 is taken, a new node is allocated and the least recent used node is deallocated. Figure 6.10 shows the process of adding a new node. In Figure 6.10(a), a new node N is created by a parallel thread and pointed by a local

pointer *n'*. After line 14, the *next* field of this new node is the same as the *head* pointer whose *D_addr* bytes are storing the address of node A. After line 15, the parallel thread changes the content of *head* pointer by creating *head'* and making its both *D_addr* and *P_addr* fields store the address of node N (see Figure 6.10(b)). When copy-out operation, the main thread can recognize the new node by checking if the two fields of *head'* are the same. After the copying operation, pointer *head* is pointing the new node N which is now in the front of the LRU buffer (see Figure 6.10(c)).

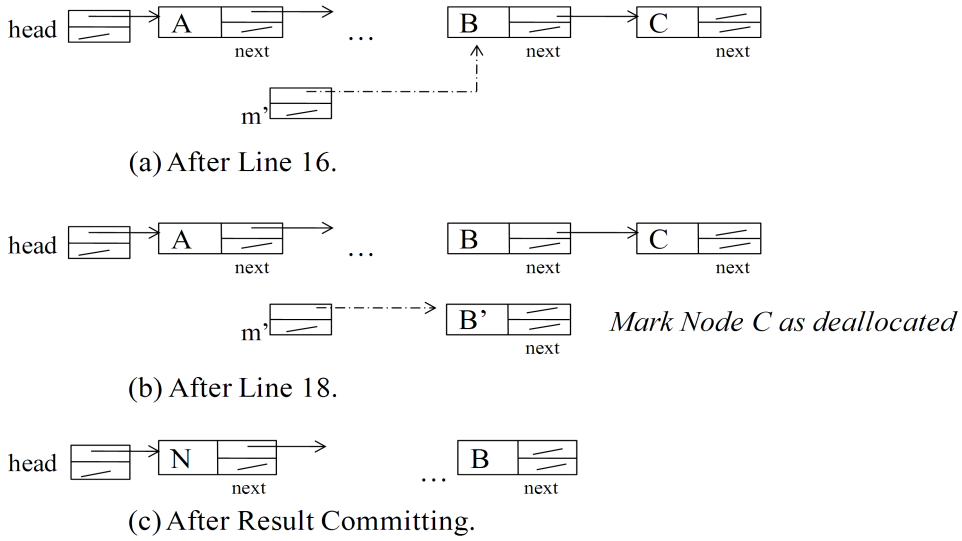


Figure 6.11: Deleting A Node From A Dynamic Data Structure.

Deleting A Node. Figure 6.11 shows the process of executing lines 16-18, which deallocates the last node in the LRU buffer. As shown in Figure 6.11 (a), after executing line 16, a local pointer *m'* is created by a parallel thread and pointing to the second last node B. When *free* is called on node C at line 17, the parallel thread simply marks the node as deallocated in the mapping table instead of actually call the function. This is because node C is in the non-speculative state and it cannot be deallocated until the speculative

computation performed by this parallel thread is decided to be correct. After line 18, a local copy of node B is created because its pointer fields is speculatively set to *NULL*. In the result-committing stage, the main thread actually deallocates the node C based on the mark in the mapping table and the *next* pointer field (8 bytes) of node *B* is also set to *NULL* due to the copy-out operation.

6.2.4 Techniques And Their Benefits

Challenges	Copy-on-write Scheme	Heap Prefix	Double Pointers
Copying Operation Overhead	Reduced	-	-
Mapping Table Access Overhead	Reduced	Reduced	-
Misspeculation Check Overhead	-	Reduced	-
Address Translation Overhead (Copy-in)	Reduced	Reduced	-
Address Translation Overhead (Copy-out)	-	-	Reduced

Table 6.1: Techniques And Their Benefits.

In this section, three techniques are introduced to address the overhead and address translation problems when speculatively parallelizing a program using dynamic data structures. The three techniques were: copy-on-write, heap prefix, and double pointers. Table 6.1 summarizes the benefits of these techniques.

6.3 Other Optimizations

6.3.1 Eliminating Unnecessary Checks

An access check precedes each write access that is performed to the heap in speculative state. Although implementing access checks via heap prefix can greatly reduce their overhead, the overhead of access checks can be still significant due to the frequency with which they are performed. Therefore, in this section, additional compile-time optimizations are developed for eliminating access checks.

Locally-created Heap Objects. When a node is created by a speculative thread, it has a valid speculative state address. Therefore, accesses performed to a locally created node do not require access checks. The algorithm shown in Figure 6.12 provides simple compile-time analysis to identify accesses that are guaranteed to always access locally created nodes. For each basic block, pointers that hold an address returned from a *malloc* function call is first identified. Then the propagation of these pointers to other pointer variables are tracked and thus additional accesses that do not require an access check can be identified. In the analysis, each pointer assigned by *malloc* is placed into the *GEN* set. If a pointer is assigned with a pointer that is not holding a local heap address, it is placed in the *KILL* set. Then the *IN* set is computed for every basic block in a control flow graph based on the equations shown in this figure. Given the *IN* set of a basic block, it is easy to determine whether or not to introduce an access check before a pointer dereference.

Already-copied Heap Objects. Given a reference to a node, if it is certain that there is an earlier write which caused the node to be copied, then no access check for the reference is

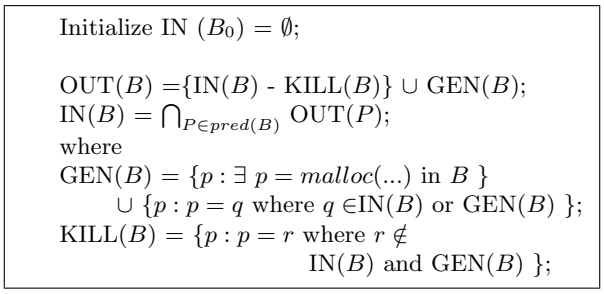


Figure 6.12: Locally Created Heap Objects.

needed. The analysis required for this optimization is quite similar to the analysis described in the preceding optimization. The difference is that the *GEN* set contains pointers through which a write is performed to a node instead of pointers assigned by *malloc*.

Read-Only Heap Access. If, following initialization, a node is only read throughout the execution, then it is impossible for such a node to cause a misspeculation. Therefore, access checks are not required for such nodes at all. However, it is challenging to identify such nodes at compile time due to pointer aliasing. Specifically, the same memory address may be pointed to by two or more pointers at runtime. During compile time, even if it is identified that the access to a node through one pointer is always a read, the node may still be modified through another pointer at runtime.

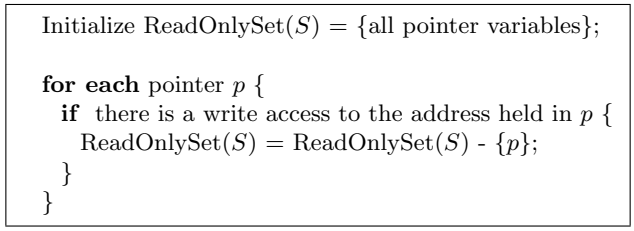


Figure 6.13: Finding Read-Only Heap Objects.

Fortunately, there has been much research work on alias analysis. For any two pointers, the alias analysis responds with three possible answers, “yes”, “maybe” and “no”,

indicating they do or maybe or do not point to the same location. Such analysis can be exploited to conservatively identify read-only nodes. In particular, any two pointers with answer "yes" or "maybe" from alias analysis, are considered as aliases. Next, the analysis shown in Fig, 6.13 can be performed. For any access to a node through a pointer in *ReadOnlySet*, no access check is inserted, because these accesses must involve read-only nodes.

6.3.2 Optimizing Communication

In the implementation of CorD model described in earlier chapters, the communication between the main thread and speculative threads was implemented through expensive system calls such as read and write to pipes. Use of pipes strictly prevents speculative threads from accessing the non-speculative state memory, as values required by parallel threads are passed through pipes. This mechanism works fine in the streaming applications because few values need to be communicated at runtime. However, the same is not true in this work – many more values are communicated to speculative threads even when copy-on-write is used. Thus, overhead of using pipes to pass values is high.

In this work, the restriction of the state separation is relaxed by allowing a speculative thread to directly read the non-speculative state memory. This results in highly efficient communication. Busy-waiting algorithms are also used to synchronize threads because they achieve low wake-up latency and hence yield good performance on a shared memory machine [53].

6.4 Experiments

6.4.1 Experimental Setup

To show the effectiveness of the techniques, 7 benchmarks are used, which are from the LLVM test suite and SPEC2000 that make intensive use of heap based dynamic data structures. In Table 6.2 the first two columns give the name and the description of each program. The next column shows the type of dynamic data structure used and the last column shows the original source of the program.

Name	Description	Dynamic Data Structure	Original Source
`BH`	Barns-Hut Alg.	Tree	Olden
`MST`	Mininum Spanning Tree	Tree, hash	Olden
`Power`	Power pricing	Graph, hash	Olden
`Patricia`	Patricia trie	Tree, hash	Mibench
`Treesort`	Tree sorting	Tree	Stanford
`Hash`	Hash table	List, hash	Shootout
`Mcf`	Vehicle scheduling	List, graph	Spec2000

Table 6.2: Dynamic Data Structures Benchmarks.

In the experiments, Pin [48] instrumentation framework is used to profile loops in these programs with a smaller input. Then the runtime dependences are analyzed and regions that are good candidates for speculative parallelization are identified. Next the LLVM [44] compiler infrastructure is used to compile these programs together with the analysis result and the parallelization template, so that the sequential version can be transformed into the parallel version. The parallelization template contains the implementation of the runtime system including thread creation, interaction, mapping table, misspeculation check etc. During the transformation of the code, access checks are inserted preceding

each heap access. The profiling is performed for a small input and the experimental data is collected by executing parallelized programs on a large input. All the experiments were conducted under CentOS 4 OS running on a *dual quad-core* (i.e., 8 cores) Xeon machine with 16 GB memory. Each core runs at 3.0 GHz.

6.4.2 Performance

The execution time of a program between its sequential version and parallel version are first compared. In this experiment, all the techniques and optimizations are used. It is observed that running these programs under the implementation of CorD described in earlier chapter led to at least 2x slowdown of parallel versions over sequential versions regardless of the number of speculative threads. However, with the proposed techniques, significant speedup is obtained. Figure 6.14 shows the execution speedup of these programs for varying number of speculative threads created by the main thread.

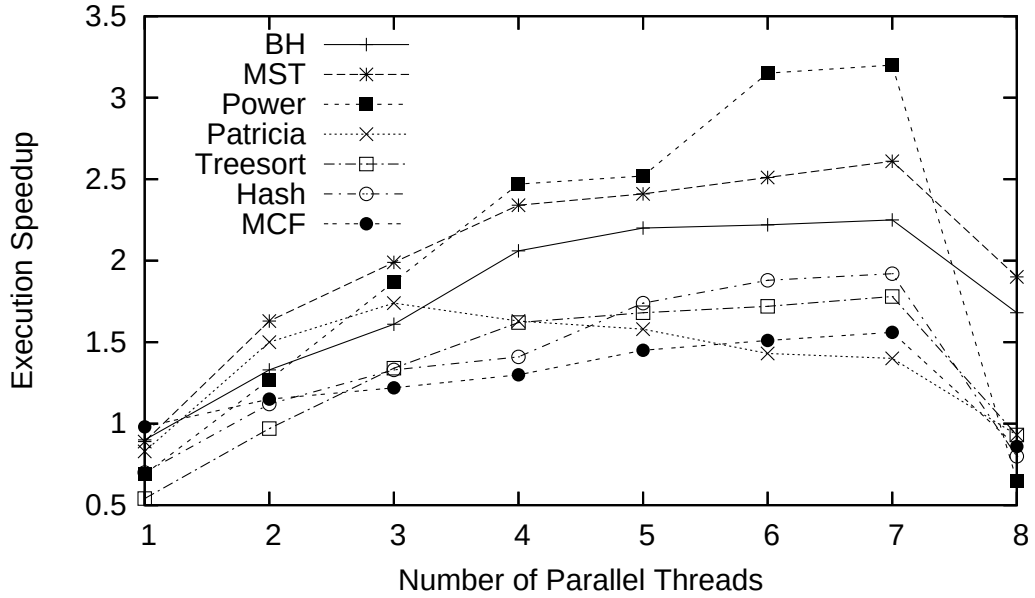


Figure 6.14: Performance On An 8-core Machine.

Our results show that for all programs except `Patricia`, the speedup continues to increase as the number of speculative threads is increased from 1 to 7. In particular, the highest speedup of `Power` is 3.2, and that of other programs is between 1.56 and 2.5 when 7 speculative threads are used. In the case of `Patricia`, significant amount work has to be done sequentially by the main thread. Also, every speculative thread needs to copy up to 1 MB memory during execution. When more threads are used, more memory is copied. This can cause L2 cache pollution and hence dramatically affect the performance. For `Patricia` best speedup of 1.74 is achieved when three speculative threads are used.

Since a total of 8 cores are available, when 8 speculative threads are used in addition to the main thread, the speedup decreases. Except for `MST` and `BH`, all programs actually have a slowdown. The reason is due to the use of busy-wait synchronization. When 8 speculative threads and one main thread are run on an 8 core machine, context switch is required. However, using busy-wait constructs makes each thread to aggressively occupy a core. This leads to even worse performance. It is worth noting that using pipe does not have this problem as the main thread will be descheduled by OS. According to the experiments, however, the use of pipe causes the parallel execution to be much slower than the sequential one regardless of the number of parallel threads.

One can also observe that using one speculative thread is slower than the sequential version. In the case of `Treesort`, even using two threads cannot obtain any speedup. This behavior can be attributed to the overhead introduced by the model that more than nullifies the limited parallelism benefits.

The misspeculation rate of each execution is also measured. The highest rate observed is 10.2% for `mcf` when using 7 parallel threads. This benchmark also has a large

sequential portion. These two factors make the highest speedup of this program only 1.56. For other programs, the misspeculation rate is less than 1%. Thus, misspeculations have little impact on the performance.

6.4.3 Overhead Analysis

Time Overhead. The execution time is classified into 4 categories: *communication* (time spent on busy-waiting constructs), *access checks*, *misspeculation check followed by copy-out operations*, and *computation*. For speculative threads, these times and averaged them across the threads are measured. The experiment was conducted for 2, 4, and 7 threads. The results are shown in Figure 6.15.

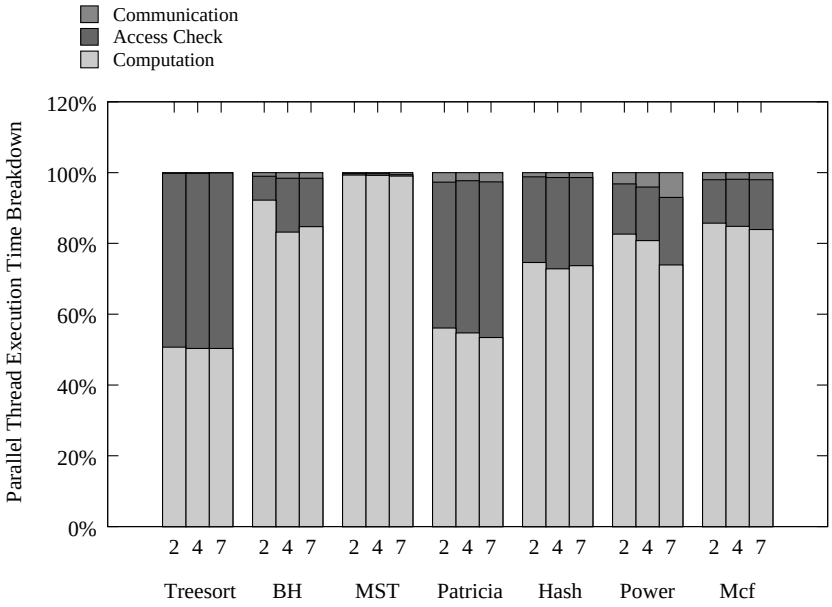


Figure 6.15: Time Breakdown: Speculative Threads.

From the figure, one can clearly see that regardless of the total number of speculative threads, each thread, on an average, spent from at least 50% (`Treesort`) to nearly 100% (`MST`) of the time on the computation. For some benchmarks like `Treesort` and `Patricia`,

a significant amount of time is spent on access checks. The communication time is very low for all benchmarks, i.e., these threads do not spend much time on waiting for their work.

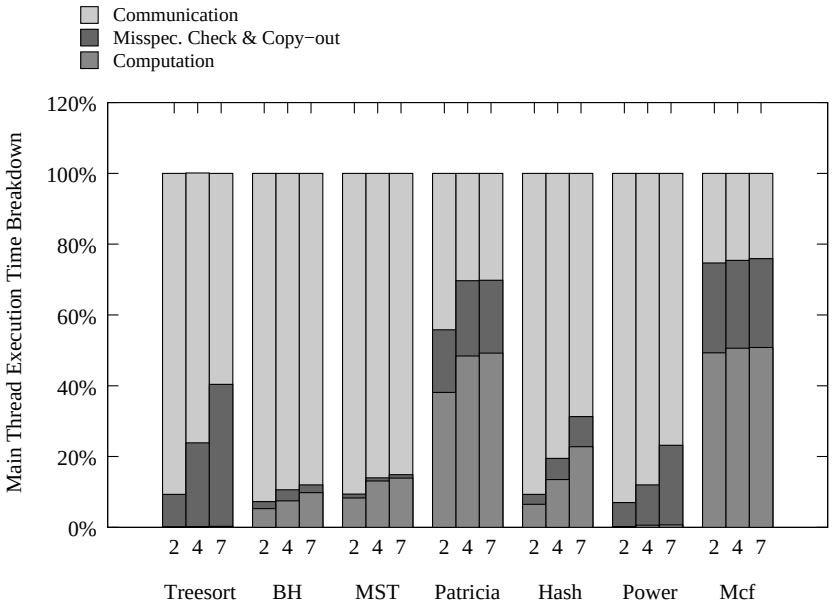


Figure 6.16: Time Breakdown: Main Thread.

Figure 6.16 shows the execution time breakdown for the main thread, which is responsible for assigning work to speculative threads, performing misspeculation checks followed by copy-out operations, and executing the sequential part of the program. As one can see, for all programs except for `Patricia` and `Mcf`, the communication dominates the main thread's execution, which means the main thread is waiting for the results of speculative threads most of the time. During the rest of the time, the main thread does more work on misspeculation checks and copy-out operations for `Treesort` and `Power`, and more work on sequential computation for `Patricia`, `Mcf`, `BH`, `MST` and `Hash`. For the last 3 programs, the sequential computation portion becomes larger as the number of parallel threads increases from 2 to 7. This is because the total execution time is reduced due to

greater parallelism and hence the sequential part becomes a greater fraction of the total execution time.

Space overhead. While the parallel execution is faster, it requires more memory space as each node has extra bytes for heap prefix and double pointers and each thread needs its own space. Therefore, an experiment was conducted to measure the peak value of memory consumption of the parallelized program for varying number of threads. Figure 6.17 shows the results.

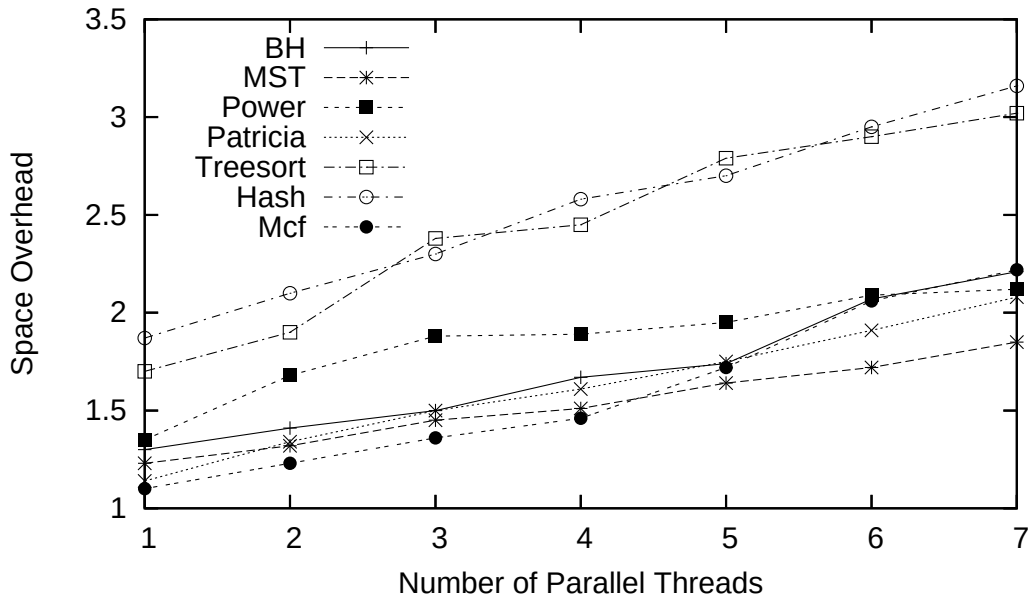


Figure 6.17: Space Overhead.

As one can see, the memory consumed by the parallel version of all programs is between 1.1x and 3.2x compared to the sequential version. Note that for most benchmarks except for `Treesort` and `Hash`, the space overhead caused by heap prefix and double pointers is at most 50% (when 7 threads are used) and often less than 20%. This is because each node in these programs takes over 60 bytes with about 2 to 6 pointers. Other space overhead

mostly comes from the coping operations. Thanks to the copy-on-write scheme, only a small number of nodes need to be copied and hence the total overhead is not very large. For `Treesort` and `Hash`, however, the node size is only 12 bytes. Therefore, the double pointer scheme and heap prefix cause significant space overhead, especially when more threads are used as shown in the figure.

6.4.4 Effectiveness of Optimizations

As unnecessary access checks can be eliminated by the proposed analysis, the number of static and dynamic access checks with and without optimizations are compared. Table 6.3 shows the number of eliminated checks and the total number of checks without any elimination. From this table, one can observe that a small number of static access checks lead to millions of dynamic checks. This is because access checks are inserted inside loops that have millions of iterations. One can also see that, on an average, the optimization eliminates 69.5% of static access checks which correspond to 71.5% of dynamic access checks. So without the optimization, each thread may waste significant number instructions at runtime on performing unnecessary checks.

Program Name	Checks Eliminated	
	Static	Dynamic (Million)
`BH`	5/7 (71.4%)	0.55/0.67 (82.1%)
`MST`	7/11 (63.6%)	8.9/13.5 (65.9%)
`Power`	55/60 (91.6%)	9.5/10.8 (88.0%)
`Patricia`	53/66 (80.3%)	62.4/79.7 (78.3%)
`Treesort`	3/6 (50%)	72.7/143.2 (50.7%)
`Hash`	7/12 (58.3%)	312.8/463.3 (67.5%)
`Mcf`	20/28 (71.4%)	968.2/1418.9(68.2%)
Average	69.5%	71.5%

Table 6.3: Effectiveness Of Eliminating Access Checks.

6.5 Summary

For programs using heap based dynamic data structures, speculative parallelization is challenging. This is because the size of the dynamic data structure can be very large and thus moving heap data between non-speculative state and speculative state can be expensive. In addition, address translation of accesses to data structure fields is needed. This chapter proposed techniques and optimizations that effectively address these challenges. The experiments show maximum speedups from 1.56 to 3.2 for a set of programs that make extensive use of heap based dynamic data structures.

Chapter 7

Related Work

7.1 Speculative Parallelization

7.1.1 Software Based TLS Techniques

Several works have proposed a software technique for speculative execution [25]. Some of them focus on the array based applications [25] [63]. In these works, each array is associated with several shadow arrays that keep track when (in terms of iteration numbers) each array element is read or write. The values in shadow arrays are used to test if a speculation succeeds. If not, then the loop is determined to be a sequential one and it needs to be reexecuted from the beginning or the latest checkpoint. In these works, state separation is achieved through copying operation, which is performed when an element is accessed.

Rundberg *et al.* [64, 65] presented another software technique for array based programs. The key idea is to associate a support data structure for each shared location and augment each speculative load and store with checking code. However, significant amount

of synchronizations have to be used between parallel threads, since no central control exists in their work. Cintra *et al.* [11, 12] proposed similar schemes. In their work, the memory fence is used to synchronize the parallel threads. The copying operation in both work is performed on speculative stores.

Note that all the above works except for Rundberg *et al.*'s work [64, 65] only rely on analyzing static information such as control flow graph and data dependence graph. Therefore, they can not parallelize a program that has heap accesses through pointers in the speculative region. Rundberg *et al.*'s work [64, 65], on the other hand, requires the complete trace of a program to perform the parallelization.

Ding *et al.* [15] proposed a *process based* runtime model that enables speculative parallel execution of Potentially Parallel Regions (PPRs) on multiple cores. However, there are several drawbacks of this approach. First, the speculative region is executed by a process instead of a thread in this approach. Each process can only communicate with its child process, and the last process cannot know the termination of the first process if more than two processes are created. Therefore, the work has to be assigned to the processes running on different cores in rounds. Thus, parallelism cannot be fully exploited.

Second, copying overhead for this approach is large. As already mentioned, the speculative region is executed by a process instead of a thread. The advantage of using a process is that all data required by a speculative process is supplied by the OS through copy-on-write scheme. While this makes implementation of the runtime system easy, the copying overhead is higher as copying at OS level is carried out at the granularity of a page. In particular, once a memory cell is written in a speculative process, OS makes a copy of the entire page containing that cell. It is worth noting that more pages need to be allocated

so as to solve the false sharing problem caused by tracking the dependence at page level. According to the description of this approach [15], each global variable needs to be allocated on a distinct page. This worsens the overhead of the copying operation.

Lastly, the overhead of process creation is high. In Ding *et al*'s approach [15], new processes are continuously created when old processes finish their execution. In the case a loop has large number of iterations, frequently creating processes negatively impacts performance. Compared to this approach, none of the above drawbacks are present in CorD.

Kulkarni *et al.* [38, 40, 41, 54, 39] also proposed a runtime system to exploit the data parallelism in applications with irregular parallelism. Parallelization requires speculation with respect to data dependences. The programmer uses two special constructs to identify the data parallelism opportunities. When speculation fails, user supplied code is executed to perform rollback. Besides, users need to use special constructors, mark all commute functions, which can be executed in any order, and define the reverse computation of each commute function in their programs. This places much burden on the users. In contrast, this work does not require help from the user, nor does it require any rollbacks.

7.1.2 Hardware Based TLS Techniques

There have been considerable research work carried out in developing hardware TLS system [13, 82, 74, 4, 91, 75, 76, 78, 17, 72, 49, 27, 24, 89, 36, 60]. In these techniques, speculative threads are spawned to venture into unsafe program sections. The memory state of the speculative thread is buffered in the cache, to help create thread isolation. Hardware support is required to check for cross thread dependence violations; upon detection of these violations, the speculative thread is squashed and restarted on the fly.

Unfortunately, most of the work is hardware based and not ready for use. This is due to the architectural redesigns requiring non-trivial hardware changes such as special buffers [27, 60], versioning cache [24], or versioning memory [23] for detecting misspeculations and handling speculatively computed results, which have not been incorporated in commercial multicore processors. Compared to these hardware based techniques, CorD does not require any hardware support and is implemented purely in software.

7.1.3 Software Pipelining

One commonly used approach for parallelization of loops is software pipelining. This technique partitions a loop into multiple pipeline stages where each stage is executed on a different processor. Decoupled software pipelining (DSWP) [59, 62, 81] is a technique that targets multicores. The proposed DSWP techniques require two kinds of hardware support that is not commonly supported by current processors. First, hardware support is used to achieve efficient message passing between different cores. Second, hardware support is versioned memory which is used to support speculative DSWP parallelization. Since DSWP requires the flow of data among the cores to be acyclic, in general, it is difficult to balance the workloads across the cores. Raman *et al.* [62] address this issue by parallelizing the workload of overloaded stages using DO-ALL techniques. This technique achieves better scalability than DSWP but it does not support speculative parallelization which limits its applicability. Other recent works on software pipelining target stream and graphic processors [7, 16, 32, 37, 77].

7.1.4 Other Parallelization Techniques.

To improve performance through loop parallelization, DOALL techniques were proposed decades ago [35, 42]. Due to the cross-iteration dependences, many programs cannot be simply parallelized. To solve this problem, DOACROSS technique was proposed [8, 45] which uses explicit send/receive calls or instructions to synchronize and exchange values between threads. However, the blocking at receives serializes the execution. As a result, DOACROSS techniques cannot greatly improve the performance, especially when the value of a live-in variable is used at the beginning of a thread's execution.

Vijaykumar *et al.* [83] presented some compiler techniques to exploit parallelism of sequential programs. A set of heuristics operate on the control flow graph and the data dependence graph so that the code can be divided into tasks. These tasks are speculatively executed in parallel and the hardware is responsible for detecting misspeculation and performing recovery. However, this work focuses specifically on Multiscalar processors. Kejariwal *et al.* and Praun *et al.* [33, 84], also presented different compiler techniques to quantify the amount of optimistic parallelism in sequential programs respectively.

Instead of concentrating on extracting coarse-grained parallelism, Chu *et al.* [10] recently proposed exploiting fine-grained parallelism on multicores. Memory operations are profiled to collect memory access information and this information is used to partition memory operations to minimize cache misses.

7.2 Related Work Of Multiple Speculations

7.2.1 Value Prediction Techniques

Value prediction techniques are very effective in breaking data dependences. They have been widely used in both instruction-level parallelism and thread-level parallelism. In 1996, Lipasti *et al.* [47] first proposed to predict the value of load instructions. The motivation is to use value prediction to hide the memory access latency, so that other instructions do not have to wait the value to be fetched from the memory hierarchy. They designed a load value predictor which can predict a load value based on the value history of the same load instruction. In their technique, each load instruction falls into one of three groups, *unpredictable*, *predictable* and *constant*, based on whether or not its most recent predictions are correct. Later, they extended their technique to other instructions aiming at hiding both memory access and computation latency[46]. In both work, the value is predicted as the same as the previous one for the same instruction. This prediction method is also called *last value* prediction.

Sazeides and Simith [68] extended the value prediction work by considering different prediction methods. They first examined the patterns of the different but commonly seen value sequences. Then they formalized two different types of predictors, *computational predictor* and *context predictor*. For the first type, they mainly discussed the *stride predictors*, which predict the next value of an instruction by adding the latest value and the difference between two most recent values produced by the same instruction. The difference is also called *stride*. In fact, last value prediction is a special case of the stride prediction in that the stride is always zero. Gabbay *et al.* [22, 21] also presented stride predictors, but

their scheme is based on profiling data.

For the context predictor, Sazeides and Simith [68] described a *finite context method*, which can be broken down to two phases, learning phase and prediction phase. In the learning phase, the value and the corresponding context presented by a finite ordered sequence of previous values are stored. In the prediction phase, the value appeared in the same context before is used as the predicted value. In their work, they argue that simply using last value prediction for instructions is not good enough. Instead, a hybrid method which contains both stride prediction and context prediction can achieve better performance. According to their experiments, 20 percent predictions using context predictor and the remaining using stride predictor yield the high prediction accuracies at lower cost.

Wang and Franklin [86] also presented their work on an improved version of stride predictor and a different version of context predictor. In their stride predictor, the value of an instruction is not predicted until the same stride is observed in the two consecutive instances. The condition is considered as a *steady* state for the stride predictor in their work, and can effectively reduce the mispredictions. They also developed a context predictor. Different from the finite context method, the predictor simply maintains 4 most recent used values produced by an instruction and a counter associated with each value indicating how often the value has repeated. If a new unique value is encountered, the least recent used value is replaced. When predicting a value, the predictor simply picks the value with the maximum counter value. They also combined these two predictors and gave a hybrid solution.

Since the hybrid predictors produce the best result, many research works have also been focusing on combining the last value prediction, stride prediction and context predic-

tion in different ways [5, 66, 9]. Besides these three common prediction methods, there also exist some other prediction methods. Tullsen *et al.* [79] proposed to use register values to predict values. Nakra *et al.* [57] presented a *global context* based value prediction. In all previous work, predictions are made based on the history of the same instruction. In Nakra *et al*'s scheme, however, two types of global information are considered, *path information* and *recently completed instructions*. The key idea is to use the correlation between instruction values and branch histories, and the correlation between values produced by close instructions. Zilles and Sohi [90] presented to use speculative slices to predict the value. A speculative slice is a small code fragment which can be executed earlier than the instruction that actually produces the value. Therefore, the latency of long-executed instructions can be tolerated by executing such small slices.

Value prediction techniques have also been applied to different architectures. Nakra *et al.* [58] describes how to predict values for VLIW machines. Fu *et al.* [18] presented a scheme to use value predictions for EPIC architecture. Marcuello *et al.* [50, 51] discussed the applicability of value prediction on multithreaded architectures. Yuan *et al.* [87] detailed the implementation of predictors on wide-issue superscalar processors.

While plenty of value prediction techniques have been implemented in hardware, some research also considered compiler solutions. Fu *et al.* [19] first proposed to use compiler to control the value prediction. However, the software overhead is too large to get good performance. As a result, a minimal hardware support is introduced to speed up their compiler based solution [20]. Larson and Austin [43] also proposed the use of the compiler to insert the prediction code and misspredict handler. In their technique, they use the confidence number to decide if a value should be predicted. Different from the hardware-

based predictor using a simple counter as the confidence [86, 9], the technique employed a branch instruction in the prediction code. Therefore, the confidence can be obtained by only using the confidence of branch predictors.

7.2.2 Pre-computation Technique

Quinones *et al.* [61] proposed the Mitosis compiler where the values of live-in variables are pre-computed through data slice on the most frequently taken path. Although this approach is effective in dealing with frequent cross-iteration dependences, it has two drawbacks. First, the data slice of one or more live-in variables on one particular path can be very large (e.g., *gzip*). Without value predictions, the pre-computation takes almost the same amount of time as executing one iteration in such cases. Second, the most frequently taken path is decided by the compiler using profiling results. However, there may exist more than one hot path in a loop execution at runtime. Moreover, the inputs used in the real runs are different from those used in the profiling runs. Therefore, the hot paths in the profiling runs may not be frequently taken in the real runs. Thus, picking one hot path at compile time may cause many misspeculations at runtime. Apart from these two drawbacks, Mitosis compiler does not fully support speculation. It relies on the hardware to detect misspeculations and handle speculatively computed results, and hence is not a purely software speculation technique, but rather a hybrid one.

7.2.3 Multipath Execution Techniques

Using control flow paths to create multiple executions has also been used in architectural designs [85, 80, 3]. In these works, spare hardware contexts are used to exe-

cute instructions along the paths corresponding to different predictions of hard-to-predict branches. If one of these redundant executions is correct, then the penalty of the branch misprediction is greatly reduced. Since these techniques focus on improving performance through hardware changes, they are different from the software based compiler technique.

7.3 Transactional Memory

Transactional memory (TM) [1, 14, 29, 56, 28, 67, 70] has been an active area of research. It is designed to enforce the atomicity of shared memory accesses in parallel programs and cannot be directly used to parallelize sequential programs [52]. However, since it has the capability of tracking dependences and detecting dependence violations between two transactions, an experiment was conducted to see the performance of using software based TM (STM) in the speculative parallelization work.

In this experiment, the program is manually transformed such that each task is put into a transaction and every access to the potentially shared memory in a task is monitored by the TM system. Similar to Mehrara *et al*'s work [52], the explicit synchronizations are added into transaction functions to enforce the in-order commit, This is important for maintaining the sequential program semantics. The TM implementation is based on a state-of-art algorithm - Sun's Transactional Locking 2 (TL2) [74]. Table 7.1 shows the speedup comparisons between CorD and STM-based solution when 2, 4 and 7 parallel threads are used respectively.

From the table, one can see that using STM in speculative execution has slowdowns in most cases. Only for *CRC*, speedups are achieved because only 4 variables in this program need to be tracked. However, STM yields much less speedups than CorD. For *Power* and

Programs	2 threads		4 threads		7 threads	
	CorD	STM	CorD	STM	CorD	STM
`BH`	1.33	0.59	2.06	0.68	2.25	0.73
`MST`	1.63	0.84	2.34	0.94	2.61	1.02
`Power`	1.27	0.93	2.47	0.97	3.20	1.08
`Patricia`	1.50	0.43	1.63	0.52	1.40	0.47
`Treesort`	0.97	0.34	1.62	0.41	1.78	0.40
`Hash`	1.12	0.64	1.41	0.73	1.92	0.87
`Mcf`	1.15	0.54	1.30	0.58	1.56	0.61
`197.parser`	1.73	0.31	2.62	0.55	3.72	0.71
`130.li`	1.57	0.64	3.19	0.78	5.05	0.87
`256.bzip`	1.82	0.59	2.85	0.75	3.98	0.83
`255.vortex`	1.68	0.48	3.01	0.57	4.23	0.62
`CRC`	1.98	1.18	3.92	1.56	7.82	1.88

Table 7.1: Speedup Comparisons.

MST, a slight speedup can be achieved when 7 parallel threads are used. The results are consistent with [52] which also shows that STM typically nullifies the performance gains in compiler parallelized sequential applications. There are several reasons for the performance loss. First, STM needs special mechanisms to avoid or resolve dead-lock and live-lock situations. Second, STM aims to achieve good throughput and fairness. This requires STM to consider the priorities of transactions [73]. Besides, STM internally uses locks to prevent data races [74] and barriers to ensure strong atomicity [69, 71] and in-order commit [52]. These special considerations are not necessary for speculative parallelization. Instead, they result in high runtime overhead for STM while providing a convenience for programmers writing parallel applications.

Note that Mehrara *et al.* [52] propose customized STM for speculative parallelization. Their work assumes dependent variables can be identified at compile time and thus they use a set of special registers to track such variables. However, for the programs using dynamic data structures, cross-iteration dependences cannot be recognized statically.

Therefore, their work is not applicable for the class of programs that are considered.

Chapter 8

Conclusions

8.1 Contributions

This dissertation makes contributions in the area of software-based thread-level speculative parallelization. A thread-based speculative execution model called CorD is proposed. It contains one main thread and several parallel threads and the main thread controls the entire execution. The memory state is divided into three disjoint partitions such that the execution of each thread is isolated. In particular, the main thread maintains the non-speculative state and parallel threads perform speculative computations on the speculative state. Coordinating state provides memory for bookkeeping important information needed to support speculation. In CorD the data communications between the main thread and parallel threads are performed through copying operations. The misspeculation is checked by the main thread, which also commits the speculatively computed results if the speculation is successful. If a misspeculation is detected, the speculative results are simply discarded and the failed task is performed again.

This dissertation also presents different realizations of CorD which adapt to different situations. For streaming applications, copying optimizations can be highly optimized based on the profiling results. A version based misspeculation detection scheme is also proposed. To deal with high misspeculation rate, two different approaches, *Multiple Speculations* and *Incremental Recovery* are proposed. In the first approach, the values of live-in variables are predicted and multiple speculative version of the same task are created and executed with a non-speculative task in parallel. If one of these versions is correct, parallelism between these two tasks is achieved. The second approach decouples the creation of each parallel thread from the creation of its speculative state. It allows speculatively computed results to be reused as long as they are not using the live-in variables that are the cause of misspeculations. Finally, challenges of applying CorD in the presence of dynamic data structures are studied and addressed.

The following research questions are addressed in this dissertation.

Can software-based TLS be applied to complex applications?

While hardware based TLS techniques have not been incorporated in any commercial processors due to non-trivial architectural changes, people wonder if software-based TLS can be general enough to parallels complex applications. Previous software techniques usually have limited applicability. Most of them only focused on array-based or scalar variable based applications [25, 63, 15]. Although the techniques proposed by Kulkarni *et al.* focused on heap-based irregular applications [38, 40, 41, 54], they require the application to use work-list type data structures and commutative statements.

The CorD model described in this dissertation does not have any specific requirement of the applications. As shown in earlier chapters, it can be adapted to both array

or scalar variable based applications and heap based applications. This dissertation shows that software-based TLS is able to be applied to different kinds of complex applications.

How well can software-based TLS work?

Previously, it was unclear how well a software-based technique can work. While parallelism benefit is obtained, prior techniques have drawn a lot of concern about overhead. Process based solution proposed by Ding *et al.* [15] incurs too much copying overhead. Thread based solution proposed by Kulkarni *et al.* [38, 40, 41, 54] suffers from recovery overhead because of the lack of state separation. The work presented in [64] and [11] have significant synchronization overhead since no central control exists.

The CorD model proposed in this dissertation overcomes all these shortcomings of software based TLS. It uses threads instead of processes, which significantly reduces the copying overhead. Moreover, copying optimizations and copy-on-write scheme are proposed to further reduce this overhead. It maintains separate states, which eliminates the need of rollback because the speculatively computed results can be simply discarded. The main thread controls the entire speculative parallelization, which largely reduce the synchronization overhead.

Besides these advantages, this dissertation also proposed other optimizations for CorD such as heap prefix, access check optimizations etc. to further reduced the overhead. All of them make the software-based solution practical and promising for speeding up sequential programs.

Can software-based TLS be applied when misspeculations frequently occur?

The most fundamental characteristic of all TLS technique is to assume dependences that pre-

vent parallelism do not manifest themselves frequently at runtime. Thus, such assumption being false is a disaster for all software-based TLS, since continuously dealing with misspeculations slows down the program execution dramatically. As a result, most software-based TLS research focus on the applications that have low misspeculation rates.

In this dissertation, two approaches are presented to specifically tackle high misspeculation rate problem. By appropriately adapting CorD model, parallelism in the sequential loop that has frequent cross-iteration dependences can still be achieved. These approaches make software-based TLS more powerful in exploiting coarse-grain parallelism.

8.2 Future Directions

Performance and power consumption tradeoff. This dissertation mainly focuses on improving performance of sequential program by exploiting multiple cores. In scientific domains, performance is the most important issue. However, in other domains, power consumption may become a big concern. In the future work, speculative parallelization technique should be adapted to meet different needs. In particular, misspeculation rate should be collected online so that the number of parallel thread can be controlled based on the amount of potential performance gain.

Online profiling and re-compilation. The parallelization algorithm and many optimizations proposed in this dissertation are based on the profiling results. In other words, the accuracy of profiling largely affects the performance of CorD. Since the input of the profiling run and the input of the real run are different, the information gathered in the profiling run may not lead to efficient parallelization. To address this problem, future work

should consider online profiling. Based on the results, the code should be re-transformed to better exploit frequently observed parallelism.

Architectural factors. While CorD can be purely realized in software as described in this dissertation, it does not reject architectural support. To further improve the performance of CorD based speculations, future work should consider offloading some functions to the hardware. Cache performance should also be taken into account because cache pollution exists when memory footprint of the parallelized application or other co-existing application is very large. Scheduling techniques should be merged into CorD to address this problem and thus better exploit the architectural features.

This dissertation currently considers the multicore processors as the target architecture. In the future, many-core processors and heterogeneous processors may become prevalent. Future work could consider adapting CorD to such new architectures.

Bibliography

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, 2006.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI '90: Proceedings of the 1990 ACM SIGPLAN conference on Programming language design and implementation*.

[3] P. S. Ahuja, K. Skadron, M. Martonosi, and D. W. Clark. Multipath execution: Opportunities and limits. In *International Conference on Supercomputing*, pages 101–108, 1998.

[4] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *PAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 99–108, 2002.

[5] B. Black, B. Mueller, S. Postal, R. Rakvic, N. Utamaphethai, and J. P. Shen. Load execution latency reduction. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 29–36, New York, NY, USA, 1998. ACM.

[6] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 69–84, 2007.

[7] I. Buck. *Stream computing on graphics hardware*. PhD thesis, Stanford, CA, USA, 2005.

[8] M. G. Burke and R. K. Cytron. Interprocedural dependence analysis and parallelization. *SIGPLAN Not.*, 39(4):139–154, 2004.

[9] B. Calder, G. Reinman, and D. M. Tullsen. Selective value prediction. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 64–74, Washington, DC, USA, 1999. IEEE Computer Society.

[10] M. Chu, R. Ravindran, and S. Mahlke. Data access partitioning for fine-grain parallelism on multicore architectures. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 369–380, 2007.

[11] M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 13–24, 2003.

[12] M. H. Cintra and D. R. L. Ferraris. Design space exploration of a software speculative parallelization scheme. *IEEE Trans. Parallel Distrib. Syst.*, 16(6):562–576, 2005.

[13] M. H. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *ISCA*, pages 13–24, 2000.

[14] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, 2006.

[15] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 223–234, 2007.

[16] K. Fan, H. Park, M. Kudlur, and S. A. Mahlke. Modulo scheduling for highly customized datapaths to increase hardware reusability. In *CGO '03: Proceedings of the 2003 International Symposium on Code Generation and Optimization*, pages 124–133, 2008.

[17] M. Franklin and G. S. Sohi. Arb: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5):552–571, 1996.

[18] C.-Y. Fu and T. M. Conte. Value speculation mechanisms for epic architectures. Technical report, 1998.

[19] C.-Y. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte. Software-only value speculation scheduling. Technical report, 1998.

[20] C.-Y. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte. Value speculation scheduling for high performance processors. In *ASPLOS'98: Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 262–271, 1998.

[21] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. Technical report, EE Department TR 1080, Technion - Israel Institue of Technology, 1996.

[22] F. Gabbay and A. Mendelson. Can program profiling support value prediction? In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 270–280, Washington, DC, USA, 1997. IEEE Computer Society.

[23] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *Transactions on Architecture and Code Optimization*, 2(3):247–279, 2005.

[24] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 195–205, 1998.

[25] M. Gupta and R. Nim. Techniques for speculative run-time parallelization of loops. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–12, Washington, DC, USA, 1998. IEEE Computer Society.

[26] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, 2001.

[27] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 58–69, 1998.

[28] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, 2003.

[29] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA'93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[30] http://llvm.org/docs/TestingGuide.html.

[31] http://www.spec.org.

[32] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *Computer*, 36(8):54–62, 2003.

[33] A. Kejariwal, X. Tian, M. Girkar, W. Li, S. Kozhukhov, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. Tight analysis of the performance potential of thread speculation using spec cpu 2006. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 215–225, 2007.

[34] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast track: A software system for speculative program optimization. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 157–168, 2009.

[35] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[36] V. Krishnan and J. Torrellas. The need for fast communication in hardware-based speculative chip multiprocessors. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 24–33, 1999.

[37] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, 2008.

[38] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 3–14, 2009.

[39] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 217–228, 2008.

[40] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 233–243, 2008.

[41] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, 2007.

[42] L. Lamport. The parallel execution of do loops. *Commun. ACM*, 17(2):83–93, 1974.

[43] E. Larson and T. Austin. Compiler controlled value prediction using branch predictor based confidence. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 327–336, New York, NY, USA, 2000. ACM.

[44] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the 2004 International Symposium on Code Generation and Optimization*, page 75, 2004.

[45] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Comput.*, 24(3-4):445–475, 1998.

[46] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 226–237, Washington, DC, USA, 1996. IEEE Computer Society.

[47] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 138–147, New York, NY, USA, 1996. ACM.

[48] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

[49] P. Marcuello and A. González. Clustered speculative multithreaded processors. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 365–372, 1999.

[50] P. Marcuello, A. González, and D. D. D. Computadors. A quantitative assessment of thread-level speculation techniques. In *In Proceedings of the 14th International Conference on Parallel and Distributed Processing Symposium (IPDPS '00)*, pages 595–604, 1999.

[51] P. Marcuello, J. Tubella, and A. González. Value prediction for speculative multithreaded architectures. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 230–236, Washington, DC, USA, 1999. IEEE Computer Society.

[52] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 166–176, 2009.

[53] 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.

[54] M. Méndez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali. Structure-driven optimizations for amorphous data-parallel programs. In *PPoPP '10: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 3–14, 2010.

[55] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Proceedings of The IEEE Intl. Symposium on Workload Characterization*, 2008.

[56] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 359–370, New York, NY, USA, 2006. ACM.

[57] T. Nakra, R. Gupta, and M. L. Soffa. Global context-based value prediction. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, page 4, Washington, DC, USA, 1999. IEEE Computer Society.

[58] T. Nakra, R. Gupta, and M. L. Soffa. Value prediction in vliw machines. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 258–269, Washington, DC, USA, 1999. IEEE Computer Society.

[59] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–118, 2005.

[60] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas. Removing architectural bottlenecks to the scalability of speculative parallelization. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 204–215, 2001.

[61] C. G. Quiñones, C. Madriles, F. J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 269–279, 2005.

[62] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *CGO '08: Proceedings of the 2008 International Symposium on Code Generation and Optimization*, pages 114–123, 2008.

[63] L. Rauchwerger and D. A. Padua. The lrpd test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans. Parallel Distrib. Syst.*, 10(2):160–180, 1999.

[64] P. Rundberg and P. Stenstrom. Low-cost thread-level data dependence speculation on multiprocessors. In *In Fourth Workshop on Multithreaded Execution,Architecture and Compilation*, 2000.

[65] P. Rundberg and P. Stenström. An all-software thread-level data dependence speculation system for multiprocessors. *J. Instruction-Level Parallelism*, 3, 2001.

[66] B. Rychlik, J. Faistl, B. Krug, and J. P. Shen. Efficacy and performance impact of value prediction. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 148–, 1998.

[67] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP'06: In Proceedings of the 11th ACM Symp. on Principles and Practice of Parallel Programming*, pages 187–197, 2006.

[68] Y. Sazeides, Y. Sazeides, J. E. Smith, and J. E. Smith. The predictability of data values. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 248–258, 1997.

[69] F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 181–194, 2008.

[70] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[71] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in stm. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 78–88, 2007.

[72] G. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 414–425, 1995.

[73] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2009.

[74] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *ISCA '00: In Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 1–12, 2000.

[75] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *HPCA '02: Proceedings of the 8th International Symposium on High Performance Computer Architecture*, pages 65–75, 2002.

[76] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. The stampede approach to thread-level speculation. *ACM Trans. Comput. Syst.*, 23(3):253–300, 2005.

[77] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, 2007.

[78] J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, 1999.

[79] D. M. Tullsen and J. S. Seng. Storageless value prediction using prior register values. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 270–279, Washington, DC, USA, 1999. IEEE Computer Society.

[80] A. K. Uht, V. Sindagi, and K. Hall. Disjoint eager execution: an optimal form of speculative execution. In *MICRO '95: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 313–325, 1995.

[81] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *PACT '07: Proceedings of the 2007 International Conference on Parallel Architectures and Compilation Techniques*, pages 49–59, 2007.

[82] T. N. Vijaykumar, S. Gopal, J. E. Smith, and G. S. Sohi. Speculative versioning cache. *IEEE Trans. Parallel Distrib. Syst.*, 12(12):1305–1317, 2001.

[83] T. N. Vijaykumar and G. S. Sohi. Task selection for a multiscalar processor. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 81–92, 1998.

[84] C. von Praun, R. Bordawekar, and C. Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 185–196, 2008.

[85] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *ISCA '98: Proceedings of the 25nd annual international symposium on Computer architecture*, pages 238–249, 1998.

[86] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 281–290, Washington, DC, USA, 1997. IEEE Computer Society.

[87] S.-J. L. Yuan, Y. Wang, and P. chung Yew. Decoupled value prediction on trace processors. In *In 6th International Symposium on High Performance Computer Architecture*, pages 231–240, 2000.

[88] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. *SIGARCH Comput. Archit. News*, 30(5):171–183, 2002.

[89] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 162–173, 1998.

[90] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 2–13, New York, NY, USA, 2001. ACM.

[91] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 85–96, 2002.