SUPERTHREADING: INTEGRATING COMPILATION TECHNOLOGY AND PROCESSOR ARCHITECTURE FOR COST-EFFECTIVE CONCURRENT MULTITHREADING

BY

JENN-YUAN TSAI

B.S., National Chiao-Tung University, 1987
M.S., National Taiwan University, 1989

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1998

Urbana, Illinois

SUPERTHREADING: INTEGRATING COMPILATION TECHNOLOGY AND PROCESSOR ARCHITECTURE FOR COST-EFFECTIVE CONCURRENT MULTITHREADING

Jenn-Yuan Tsai, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1998
Urbana, Illinois 61801
Pen-Chung Yew, Advisor

This thesis presents a concurrent multiple-threaded architectural model, called *superthreading*, for exploiting fine-grained thread-level parallelism on a processor. This architectural model adopts a thread pipelining execution model that allows threads with data dependences and control dependences to be executed in parallel. The basic idea of thread pipelining is to compute and forward recurrence data and possible dependent store addresses to the next thread as soon as possible, so the next thread can start execution and perform run-time data dependence checking. Thread pipelining also forces contiguous threads to perform their memory write-backs in order, which enables the compiler to fork threads with control speculation. With run-time support for data dependence checking and control speculation, the superthreaded architecture can exploit loop-level and instruction-level parallelism from a broad range of applications.

In this thesis we also present the compiler techniques for superthreaded processors. Many existing compiler techniques used in traditional parallelizing compilers for multiprocessors as well as some specific compiler techniques for superthreaded processors are needed for generating superthreaded codes and enhancing parallelism between threads. We evaluate the performance of the superthreaded architecture with a trace-driven, cycle-by-cycle superthreaded processor simulator by using codes transformed by hand and codes generated by our superthreading compiler prototype. The simulation results show that a superthreaded processor can achieve good performance by exploiting both thread-level and instruction-level parallelism in programs.

To my wife, Chiung-Mei.

Acknowledgments

I am deeply grateful to my PhD advisor, Professor Pen-Chung Yew, for the constant guidance and encouragement he provided in the course of this research. I value his insightful advice greatly and feel honored to have worked with him for this thesis.

I would like to thanks the members of my committee, Professors David Padua, Josep Torrellas, and Wen-Mei Hwu, for taking time out of their their busy schedule to serve on my thesis committee and for their helpful comments and suggestions. I would also like to thank Professors David Lilja and Zhiyuan Li for providing me with precious advice during my stay at the University of Minnesota.

I would like to acknowledge my colleagues at the University of Illinois and colleagues at the University of Minnesota for their throughout my studies. I learned a lot from Ding-Kai Chen, Lynn Choi, Pavlos Konas, Dave Poulsen, and other members at the Center for Supercomputing Research and Development in the first two years of my study at University of Illinois. I thank Zhenzhen Jiang and Eric Ness for their help with the implementation of the superthreaded processor simulator. I would also like to thank Bixia Zheng, Xin Wang, Yonghong Song, Stephen Schwinn, Tong Chen, Jian Huang, Sangyeun Cho, Hock-Beng Lim, and other members of our group for their comments and suggestions for this research.

I am very grateful to my parents and my brothers for their unfailing concern, care, and support throughout my life. Finally, I want to thank my lovely wife, Chiung-Mei Huang, for her unconditional love and support. I feel very lucky to have her be with me during my studies at the United States.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As the rapid progress of VLSI technology allows microprocessors to have more functional units on a single chip, future microprocessors will soon be able to issue more than a dozen instructions per machine cycle. Existing microprocessors, which mostly use a superscalar approach, and VLIW architectures utilize multiple functional units to exploit instruction-level parallelism from a single thread of control flow. Such microprocessors rely on the compiler or the hardware to extract instruction-level parallelism from programs, and to schedule independent instructions on multiple functional units in each machine cycle. As the issue rate of future microprocessors increases, the compiler or the hardware will have to extract more instruction-level parallelism from programs by analyzing a larger instruction window. However, it is known to be very difficult to extract enough parallelism with a single thread of control even for a small number of functional units [5, 23, 49].

A single-threaded sequencing mechanism has several major limitations. For example, independent instructions from different basic blocks need to be grouped together in a single instruction stream. As a larger instruction window size is used, several basic blocks, which may be control dependent on different branch conditions, need to be examined together. Instruction-level speculative execution with branch prediction is needed to move independent instructions across basic block boundaries. However, a large instruction window size requires the compiler or hardware to perform more levels of branch prediction and speculation. The accuracy of branch prediction at deeper levels will suffer quickly [16]. Also, in VLIW architectures, the code size will expand exponentially because a single-threaded code needs to include all possible combinations of branch conditions. This problem is especially serious when a compiler attempts to software pipeline a loop with many conditional branches [50].

In superscalar architectures, the processor needs to perform run-time dependence checking for both register and memory accesses. The hardware overhead for such dependence checking is very high and can grow quadratically as the size of the instruction window increases [16]. In VLIW architectures, the sub-operations of each long-word are executed in a lock-step fashion to enforce the dependences between instructions. Any asynchronous event, such as a cache miss, caused by any one of the sub-operations will stall all other sub-operations in the same long-word instruction even if they are independent. These unnecessary stalls will happen more frequently as the issue rate of the VLIW processor increases.

With the above mentioned limitations in the single-threaded execution model, it is important to consider other possible models for future microprocessors which will have more functional units and higher issue rates. In this thesis, we present a concurrent multiple-threaded model, called *superthreading* [43], as an alternative. The superthreaded architecture integrates compilation techniques and run-time hardware support to exploit both thread-level and instruction-level parallelism in programs. It uses a thread pipelining execution model to enhance the overlapping between threads. It also provides compiler-directed thread-level control and data speculation and run-time data dependence checking. These features allow the compiler to exploit more potential thread-level parallelism in general-purpose applications.

1.1 Thesis Organization

This thesis is organized as follows. Chapter 2 gives the background of concurrent multi-threaded models. In Chapter 3, we introduce the superthreaded execution model [43, 39]. Chapter 4 describes the microarchitecture of a superthreaded processor. The compilation techniques [41, 42, 26] or the superthreaded model is presented in Chapter 5. Chapter 6 evaluates the performance of the superthreaded architecture by using hand compiled codes [40]. Chapter 7 presents the framework of a superthreading compiler prototype based on the SUIF compiler, and shows the speedups of superthreaded codes generated by the compiler. In chapter 8, we present an extension of the superthreaded architecture, called *two-dimensional superthreading* [44], which can support memory latency hiding and improve instruction throughput. Finally, we give our concluding remarks and the direction for future work in Chapter 9.

Chapter 2

Background

As the single-threaded model faces difficulty in exploiting instruction-level parallelism from a large instruction window, it is natural to consider using multiple threads to examine multiple instruction windows simultaneously. With multiple control flows, *i.e.*, using multiple program counters, a processor can effectively exploit loop-level, or thread-level, parallelism. This approach is similar to having a very tightly-coupled multiprocessor on a single chip. When compared with superscalar or VLIW architectures, the concurrent multiple-threaded architectures have more flexibility in exploiting thread-level parallelism. The independent instruction streams need not be bound and stalled together, making it simpler and more efficient to support speculative execution at the thread level. When compared with multiprocessor architectures, the concurrent multiple-threaded architectures can support more aggressive data communication and synchronization mechanisms between threads to exploit more fine-grained parallelism. In addition, multiple functional units can be shared among threads for better utilization in some multiple-threaded architectures.

Many concurrent multiple-threaded processor architectures have been proposed and studied [2, 8, 9, 10, 11, 14, 15, 17, 18, 19, 31, 36, 47, 46, 52, 32, 38, 28, 29, 20, 21]. Some of them

[9, 14, 18, 31, 47, 46] are primarily for increasing system throughput by allowing multiple programs (e.g. one program on each thread) to be run concurrently. In this thesis, we focus on models that are primarily for speeding up the execution of one single program. In this chapter, we review those concurrent multithreaded execution models and their architectural support for inter-thread data communication and thread-level speculation.

2.1 Concurrent Multithreaded Execution Models

Concurrent multithreaded architectures speed up the execution of a program by fetching and executing instructions from different program locations concurrently with multiple program counters. To facilitate concurrent execution of multiple threads from a single program, all concurrent multithreaded architectures provide some sort of parallel execution model for creating and executing concurrent threads at runtime. Most of the parallel execution models allow thread generation and optimization to be done at compile time. The parallel execution models are usually incorporated in the instruction set architecture to reduce the run-time overhead for thread creation. Those parallel execution models can be classified into four categories as described below:

Variable instruction streams: This model is proposed by Wolfe and Shenused in their XIMD (Variable Instruction Stream, Multiple Data Stream Processor) architecture [52]. The XIMD architecture, which is proposed as an extension to the VLIW model, can split its functional units among one or more instruction streams. Furthermore, the number of instruction streams can vary dynamically at run time based on a scenario generated by the compiler. Like a VLIW instruction, an XIMD instruction is partitioned into several field, one for each functional unit. Each field contains a data operation to be executed on the functional unit, and a control operation that specifies the address (which can be represented as a conditional branch) of the next instruction to execute on the funational unit. To allow instructions from different addresses to be executed on the multiple functional units at the same time, each functional unit has a separate sequencer to fetch instructions specified by its control operations. If the control operations in each instruction are set to be the same for all functional units, the XIMD acts like a VLIW processor with single instruction stream. To support a multiple instruction streams mode, the compiler can vary the number of instruction streams and split them among the functional units with different control operations in each instruction.

Master/Slave threads: This model is similar to the multiprocessing or multithreading programming model in the operating system environment. In this model, a program starts with a master thread. The master thread can fork one or more slave threads. A slave thread inherits the state of the master thread, and executes a piece of sequential code which is independent of the master thread. The master thread and the slave threads execute in parallel on different processing elements or funational units. After a slave thread completes its execution, it is merged into the master thread and then retires. Concurrent multithreaded architectures that fall into this category include M-machine [10, 19] and SPSM (Single-Program Speculative Multithreading) architecture [8].

In the M-machine, a home H-thread can create concurrent H-threads by executing a special fork operation. Each H-thread executes on a cluster (processing element) and has its own register file. While running, H-threads communicate through registers and can write data to the register file of other H-threads. An H-thread suspends by executing an exit operation.

In the SPSM architecture, a main thread can initiate future threads also by means of a fork instruction. A future thread executes a sequential instruction trace which is logically ahead of the main thread. When a future thread is started, a copy of the current state of the main thread becomes the initial state of the future thread. The main threads and the future thread then fetch and execute in parallel. At some point the main thread catches up, i.e., its flow reaches the starting address of the future thread. The future thread's updates are merged into the main thread's state. One important feature of this execution model is that the fork instruction merely provides a compile-time *hint* regarding the forking potential. The hardware can choose to or not to fork a future thread depending on the available machine resources at run time.

Predecessor/Successor threads: In this model, the control flow graph of a program is partitioned into a sequence of threads or tasks by the compiler. At run time, the processor starts the execution of the program by assigning the entry thread of the program on one of its processing elements. The entry thread can fork one of its successor threads in the control flow graph on another processing element. The successor thread can further fork its own successor thread on another processing element, and so on. When multiple threads execute in parallel, the oldest thread in the program sequential order is called *head thread*. Logically, the head thread represents the current state in the program sequential order. Unlike the master/slave threads model, where slave threads are retired and merged into the master thread after completion, this execution model retires the head thread when the head thread completes its execution. After the head thread is retired, its immediately successor thread becomes the new head thread and the state of the successor threads become the new program sequential state.

There are two major advantages with this execution model. First, it can naturally support thread-level speculation, since the program sequential state is always represented by the head thread, and if the speculative execution of successor threads is incorrect, the processor only need to discard the states of the incorrect successor threads and restart them. Second, the sequential thread execution order allows it to use a very simple and fast unidirectional ring to connect all processing elements, since data always flow from predecessor threads to successor threads.

Concurrent multithreaded architectures falling into this category include Multiscalar [12, 36] and the superthreaded architecture presented in this thesis. In the multiscalar paradigm, the control flow graph of a program is transformed into a task graph by the compiler. At run time, a multiscalar processor uses a sequencer to walk through the task graph and assign tasks (with task prediction) to multiple processing elements one by one in the program sequential order. Tasks execute on the multiple processing elements in parallel but complete and retire in the program sequential order. Each processing element has it own register file and own stage in the shared *Address Resolution Buffer (ARB)* [13] to maintain its speculation state corresponding to register and memory values. When a task is assigned to a processing element, the live register values from its predecessor tasks will be forwarded to its register file through the unidirectional ring.

Dynamic threading: Recently, a couple of concurrent multithreaded execution models [32, 34, 45, 29] have been proposed to identify concurrent threads or traces at run time. These models do not rely on the compiler to identify concurrent threads from a sequential program and to generate multithreaded code. Instead, they take the original sequential binary as input program and generate concurrent threads by dynamically analyzing the control flow and data flow information from instruction traces. This approach allows a concurrent multithreaded processor to exploit thread-level parallelism from existing codes without recompiling them. However, the hardware cost for performing control flow and data flow analyses at run time could be very high. Furthermore, the scope of dynamic concurrent thread detection cannot be too large due to the hardware resource constraints. This will limit their ability to exploit thread-level parallelism in larger loops or parallel sections.

2.2 Inter-thread Data Communication

Most of concurrent multithreaded architectures allow threads to communicate register data or memory data with each other to exploit fine-grained thread-level parallelism. Because a concurrent multithreaded processor is built on a single chip, it can be more tightly coupled than a multiprocessor, and can provide more aggressive mechanisms for data communication and synchronization between threads. This section describes the data communication mechanisms used in some concurrent multithreaded architectures.

2.2.1 Register Data Communication

Some concurrent multithreaded architectures such as XIMD [52], Elementary Multithreading [15], M-machine [10, 19], Multiscalar [11, 36], Trace Processor [32, 34] allow concurrent threads to communicate or share register data with each other. In the XIMD architecture, all threads (instruction streams) share a global register file. This approach allows data communication between threads to be very straightforward. However, sharing register file requires concurrent threads to use different registers for their own local register values. Register allocation may become very complicated because two concurrent threads cannot use the same registers for their local values. Moreover, sharing register file requires the register file to provide more read/write ports, which may in turn increase the processor cycle time.

To avoid the disadvantages of sharing register file, most of concurrent multithreaded architectures provide a local register file for each thread. To support data communication between register files of different threads, the M-machine allows an H-thread to write the result of an instruction directly to the register file of another H-thread. Each register also has a *full/empty* bit to support data synchronization between H-threads. The *full/empty* bit is reset to *empty* by an *empty* instruction executed on the current thread and is set to *full* when another thread writes a result to the register.

Elementary Multithreading use a queue register file to communicate register data between two neighboring threads. The queue registers are mapped into some registers of the neighboring threads. The reference to such registers is interpreted as the reference to queue registers. Queue registers are also attached with *full/empty* bits to support data synchronization.

Multiscalar allows a thread (task) to forward register data to its successor task through the unidirectional ring connection between processing units. To facilitate register data forwarding and synchronization, the compiler of multiscalar generates a *create mask* [3] for each task which indicates the register values the task may produce. In addition, the last instruction in a task that produces the value is attached with a *forward* bit to inform the processing element to forward the result of the instruction to its successor tasks. On the other hand, when a task reads a register which is in the *create masks* of its predecessor tasks and has not yet been received, the task will wait until its predecessor tasks forward the value.

Traces processors use a local register file per trace for holding values produced and consumed solely within a trace, and use a global register file for holding values that live between traces. In order to obtain register renaming information at run time, trace processors perform trace preprocessing before storing a trace into the trace cache. In the trace preprocessing, register operands are marked as local or global, and locals are pre-renamed to the local register file. During execution, a trace accesses global register file for live-in register values. If a live-in register value is not available, the trace will wait for the value produced by predecessor traces by monitoring the global result bus, or use the predicted value [27] of the register based on the history of the previous traces. On the other hand, when an instruction produces a live-out value, the result will be written to the global register file via the global result bus.

2.2.2 Memory Data Communication

For memory data dependences between threads, most of concurrent multithreaded architectures communicate and synchronize memory data through the shared data cache or the main memory. To enforce data dependences between threads, most models rely on the compiler to detect dependences between threads, and to insert explicit data synchronization and communication commands in a program. Without any run-time support for memory disambiguation, the compiler must be very conservative so that data dependences that are unknown at compile time (called *maybe* data dependences) will need to be enforced. For programs with a lot of *maybe* data dependences between threads, the performance will suffer if those ambiguous data dependences do not actually occur at run-time.

Some models [11, 36, 8, 38, 20] allow the compiler to speculate on those *maybe* data dependences with run-time data speculation support. We will discuss this approach in the next section in more detail. Another approach is to perform run-time data synchronization based on the history of previous execution [30, 48]. However, this approach may require a lot of storage to keep all previous data dependence history. In this thesis, we propose a run-time data dependence checking mechanism to detect and synchronize inter-thread memory data dependences at run time with the help of the compiler. We will present the run-time data dependence checking mechanism in the next chapter.

2.3 Thread-level Speculative Execution

Speculative execution at the instruction level is a very important technique used in superscalar processors to exploit more potential instruction-level parallelism. It allows the compiler or a processor to speculate on unknown control or data dependences between instructions and enables instructions with potential dependences to execute in parallel. For concurrent multiple-threaded architectures, speculative execution at the thread level is also very important in order to exploit more potential thread-level parallelism in programs. In this section, we review the thread-level speculative execution support for control and data dependences.

2.3.1 Control Speculation

Some concurrent multiple-threaded models such as Elementary Multithreading [15], Multiscalar [11, 36], SPSM [8], Traces processors [32, 34], and others [20, 28, 29] support thread-level control speculation. A thread which is control dependent on some conditions determined by earlier threads can be initiated and speculatively executed before the conditions are known. After the earlier threads have determined the conditions, the speculative thread is committed or squashed, accordingly. Thread-level control speculation is very useful in exploiting parallelism from WHILE loops where a later iteration is control dependent on the exit condition in an earlier iteration.

To support thread-level control speculation, a concurrent multithreaded processor must buffer the results of speculative threads before they are committed. Buffering register results for speculative threads is very straightforward, since most of concurrent multiple-threaded architectures provide a local register file for each thread. The results of a speculative thread can be saved in the local register file before it is committed or merged into the master thread. For memory data, a concurrent multithreaded processor can use a store buffer in each thread to save the store data of the speculative threads. For example, in the address resolution buffer (ARB) of a Multiscalar processor, each active task has its own value field in every entry to save its store data before it is committed.

2.3.2 Data Speculation

Multiscalar [11, 36], SPSM [8], Traces processors [32, 34], and others [38, 20, 28, 29] also support speculation on the *maybe* data dependences with the help of run-time address disambiguation hardware. In these models, the compiler or the processor speculates on *maybe* data dependences between threads without generating explicit data synchronization instructions to enforce them. At run time, the address disambiguation hardware keeps track of the memory operations generated by all active threads, and checks if there are any violations of the memory dependences. A dependence violation is detected if a thread writes to a memory location after a later thread reads from the same location. When this happens, the later thread, and all of the threads following it, have to be squashed and then re-execute to preserve the correct memory access order. The run-time support for memory speculation and dependence checking can improve the performance because the number of data synchronizations can be reduced. However, the cost of incorrect memory speculation can be very high, since any dependence violations will force the later threads to discard all computed results and to re-execute from the beginning. In addition, the hardware for run-time memory disambiguation can be very expensive, since it needs to save and compare all of the load and store addresses from all active threads.

Chapter 3

The Superthreaded Architectural Model

The superthreaded architecture, like other concurrent multiple-threaded architectures, exploits task-level parallelism with multiple threads of control. In its general form, a superthreaded processor comprises a number of thread processing elements , which are connected to each other with a unidirectional ring as shown in Figure 3.1. The multiple thread processing elements share the instruction cache and the data cache. There is also a shared register file which includes some global registers and a lock register. At run-time, the multiple thread processing elements, each with its own program counter and instruction execution data path, can fetch and execute instructions from multiple program locations simultaneously.

The compiler statically partitions the control flow graph of a program into threads. Each thread corresponds to a portion of the control flow graph. It performs a round of computation on a set of data which, has no, or only a few, dependences with other concurrent threads. The compiler can

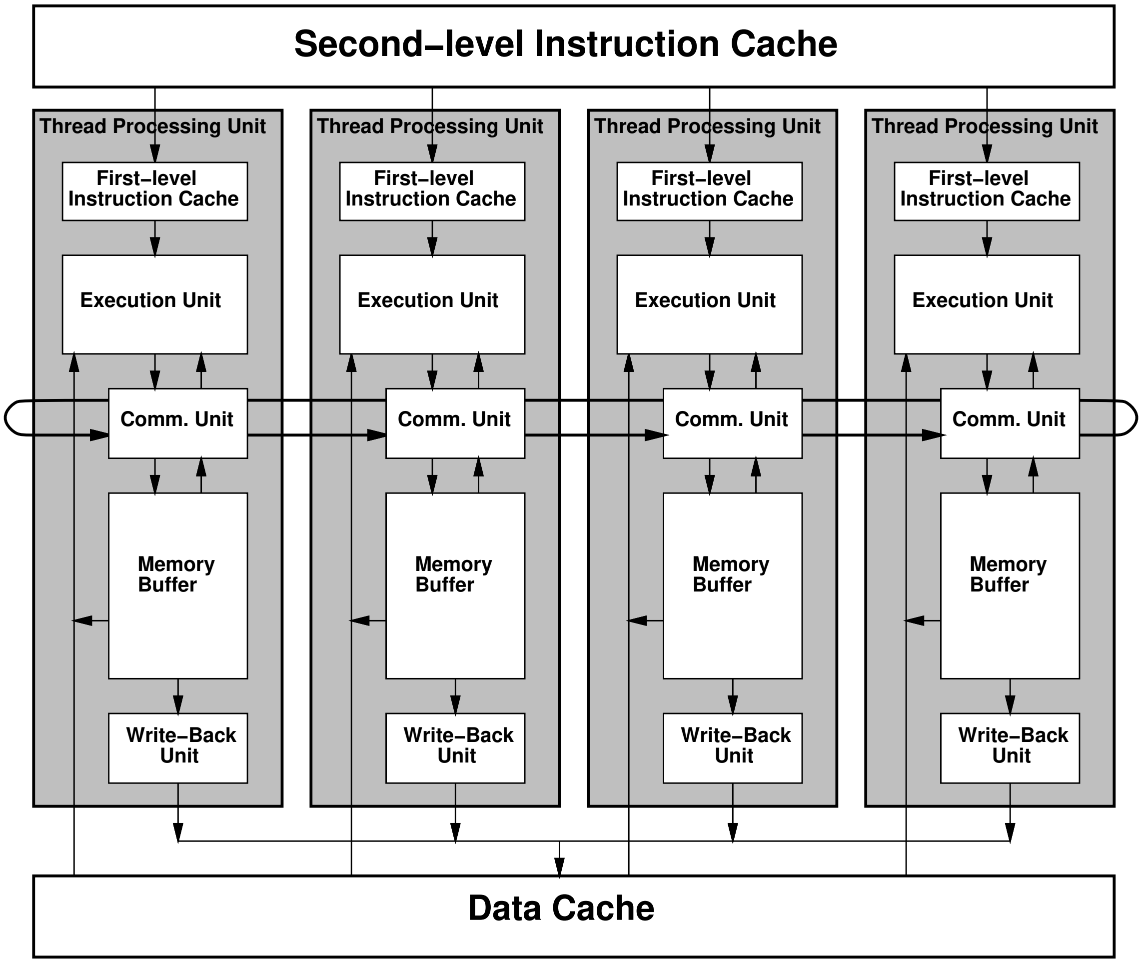

Figure 3.1: The organization of the superthreaded processor

decide the granularity of the threads. In general, a thread could be one or several iterations of a loop.

The execution of a program starts from its entry thread. It can fork a successor thread on another thread processing element. The successor thread can further fork its own successor thread. This process continues until all thread processing elements are busy.

When multiple threads are executed on a superthreaded processor, the oldest thread in the sequential order is referred to as the *head thread*. All of the other threads derived from it are called *successor threads*. After the head thread completes its computation, it will retire and release the thread processing element. Its successor thread becomes the new head thread. The completion and retirement of the threads must follow the original sequential execution order.

In the superthreaded execution model, a thread can fork one of its successor threads with or without control speculation. When forking a successor thread without control speculation, it must ensure that all of the control dependences of the successor thread have been satisfied. If a thread forks a successor thread with control speculation, it must later verify all of the speculated control dependences. If any of the speculated control dependences are false, the thread must issue a command to kill the successor thread and all of its following threads.

3.1 Thread Pipelining Execution Model

The superthreaded architecture uses a thread pipelining model to enforce data dependences between concurrent threads. Unlike the instruction pipelining mechanism in a superscalar processor, where instruction sequencing and data dependence checking and forwarding are performed by processor hardware automatically, the superthreaded architecture performs thread initiation and data forwarding through explicit thread management and communication instructions. The execution of a thread is partitioned into several stages, each of them performing a specific function, as described below. Figure 3.2 shows the pipeline stages of a thread and the relation between concurrent threads.

3.2 Continuation Stage

After a thread is initiated by its predecessor thread, it starts with the continuation stage. The major function of the continuation stage is to compute recurrence variables, such as loop index variables, needed to fork the next thread. These variables are called *continuation variables.* The values of

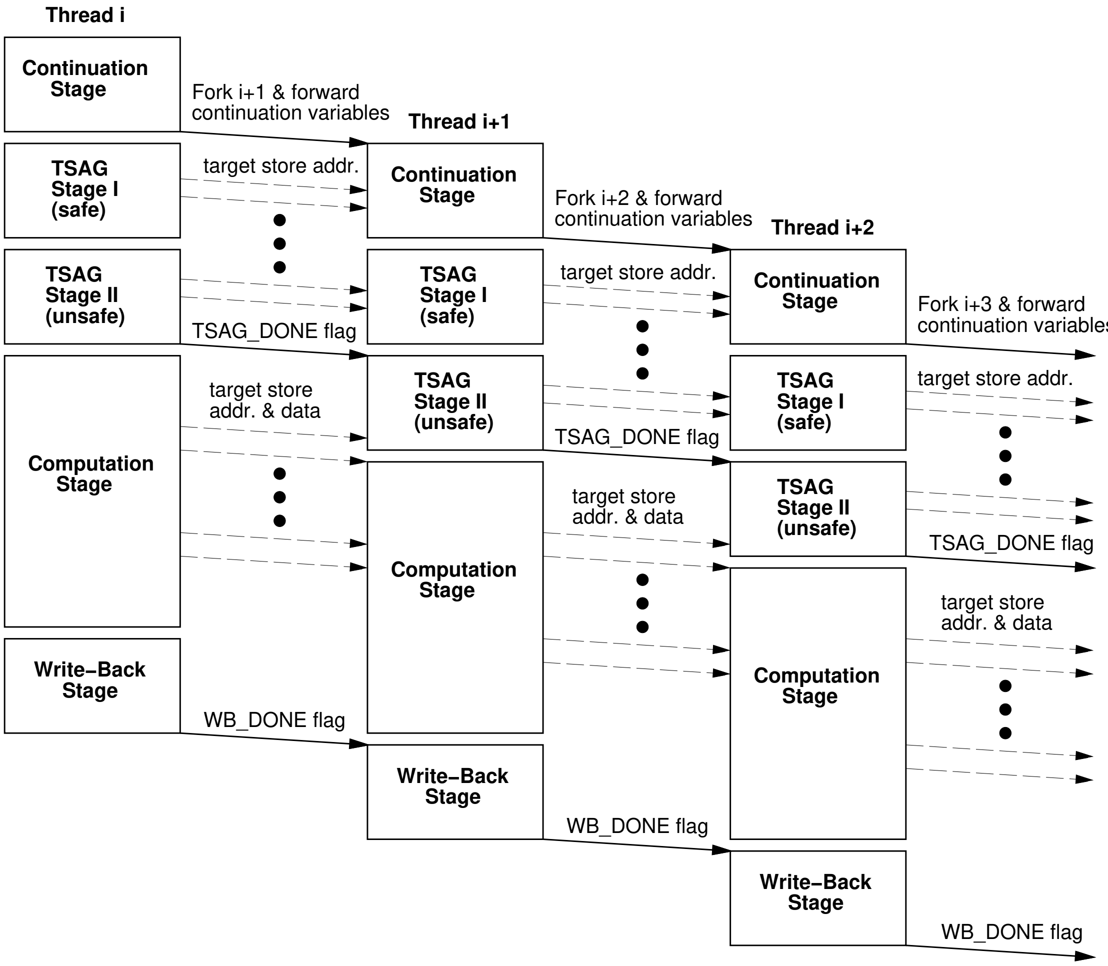

Figure 3.2: The pipelined execution of contiguous threads

the continuation variables will be forwarded to the next thread processing element before the next thread is activated. In the case of a DO loop, the continuation variables, such as i = i+1 or p = p->next, will be computed and forwarded to the next thread processing element. The continuation stage of a thread is ended with a fork instruction. The next thread will start after the execution of the fork instruction. As shown in Figure 3.2, the continuation stages of two adjacent threads will never overlap. The serialization of continuation stages is necessary because the computation of the continuation variables is dependent on the results from the continuation stage of the predecessor thread.

3.2.1 Target-Store-Address-Generation Stage

A thread may perform store operations on which later concurrent threads could be data dependent. We refer to these store operations as *target stores (TS)*. To reduce hardware complexity, the superthreaded model will not allow speculation on data dependences. To facilitate run-time data dependence checking, the addresses of these target stores need to be calculated as soon as possible. The target-store-address-generation (TSAG) stage performs the address computation for these target stores. These target store addresses will be stored in the memory buffer of each thread processing element (see Figure 4.1). They also need to be forwarded to the memory buffers of all succeeding concurrent threads for their run-time dependence checking.

After a thread completes the TSAG stage and all of the target store addresses are forwarded, it will send a *tsag_done* flag to the successor thread. The *tsag_done* flag informs the next thread that it can start computation that may be dependent on the predecessor threads. Before receiving the *tsag_done* flag, a thread can only perform computation that will not depend on any target stores (except those previously computed continuation variables) of its active predecessor threads.

3.2.2 Computation Stage

The computation stage performs the main computation of a thread. During the computation stage, if the address of a load operation matches that of a target store entry in its memory buffer, the thread will either read the data from the entry if it is available or wait until the data is forwarded from an earlier concurrent thread. On the other hand, if the value of a target store is computed during this stage, the thread needs to forward the address and the data to the memory buffer of all its concurrent successor threads. If a target store only occurs in one of the branches in a conditional statement, and that branch is not taken during the execution of the thread, a null data needs to be forwarded to all concurrent successor threads so that they will not be suspended indefinitely. The computation stage of a thread, if not killed by a predecessor thread, will end with a stop command.

3.2.3 Write-Back Stage

If the control dependences are cleared after the computation stage, which then allows the thread to complete its execution, the thread processing element will write all of the data from its store operations in its memory buffer, including data from both target stores and non-target stores, to memory. Data from store operations need to be kept in the memory buffer before write-back stage to avoid the memory state being altered by a speculative thread which is prematurely aborted by an earlier concurrent thread due to incorrect control speculation. To maintain the correct memory state, concurrent threads must perform the write-back stages in their original order. To simplify the control, a thread must wait for a *wb_done* flag from its predecessor thread before it can perform its write-back stage. It also needs to forward a *wb_done* flag to the next thread after it completes its write-back stage.

Because all of the stores are committed thread-by-thread, write-after-read (anti-dependence) and write-after-write (output-dependence) hazards cannot occur during the run-time. Besides, a target store address need not be forwarded further after the data has been stored back to the memory. A target store needs to be forwarded to no more than $N-1$ concurrent successor threads, where N is the total number of thread processing elements, because each thread is initiated and retired in order. The Nth successor thread will be initiated on the same thread processing element after the current thread completes its write-back stage and retires. Since all of the data from the store operations in the memory buffer have been written-back to the memory, the Nth successor thread will never need a target store address from the currently retired thread.

3.2.4 More on TSAG Stage

To allow more overlap between threads, we can further partition the target-store-address-generation stage into two parts. The first part is for target store address generations that do not have any data dependence on earlier threads. These target store addresses can be quickly computed and forwarded to the next thread. The second part is for target store address computations that are data dependent on the earlier thread. These computations must wait for the *tsag_done* flag from the predecessor thread (see Figure 3.2).

3.3 Superthreading Instructions

To facilitate thread pipelining and data forwarding, a set of thread management and communication instructions are provided as described below.

3.3.1 Fork

The *fork <address>* instruction starts a new thread on the next thread processing element. When a *fork* instruction is executed, the thread processing element activates the next thread processing element, and copies part of the memory buffer to the next thread processing element. The part copied includes the continuation variables computed in the continuation stage, and the target store entries from the current thread and all its predecessor threads that have not yet retired. After the copying, the thread starting address is sent to the next thread processing element. The next thread processing element then initializes its program counter with the given address and starts the execution of the new thread.

If the next thread processing element is still busy, the thread processing element will delay the *fork* instruction until the next thread processing element completes its execution and becomes idle. The current thread processing element will continue executing instructions following the *fork* instruction without being stalled. The target store addresses and data generated are kept in the memory buffer, and will be forwarded to the next thread processing element when the *fork* instruction is actually executed.

3.3.2 Target Store Instructions

The superthreaded model provides three special instructions to facilitate the generation and forwarding of target store addresses and data.

- *allocate_ts <address>:* This instruction allocates a target store entry in the memory buffer, and forwards the address to the next thread processing element.

- *store_ts <address, data>:* This instruction stores the data in the memory buffer entry corresponding to the target store address, and forwards the address and the data to the next thread processing element. If a corresponding target store entry cannot be found in the memory buffer, a new target store entry is allocated. This instruction is used to perform the last write to a target store entry in a thread. For all other writes to a target store address, regular store instructions are used.

- *release_ts <address>:* This instruction forwards the specified target store entry, including both address and data, to the successor thread processing element. This instruction is used to release a target store entry that will not be written after this point because the control path which executes the corresponding *store_ts* instruction is not taken. If the data field of the entry is empty, null data will be forwarded.

Every target store address allocated at the TSAG stage has exact one corresponding *store_ts* or *release_ts* instruction along any control path to release the last store value. It is possible, however, that a target store entry is allocated to two target stores which have the same address (*i.e.*, they are alias) and will be released twice by their corresponding *store_ts* or *release_ts* instructions. For this case, we use an alias count, which will be described in Section 4.3, to determine when a target store data can be released.

When a thread processing element forwards a target store entry to the next thread processing element, it will attach a *target-store distance count (TSDC)* with an initial value of 1 to the entry. After the next thread processing element receives the entry, it will increment the TSDC by one before it forwards the entry to its next thread processing element. When a thread processing element receives a target store entry with the TSDC equal to $N-1$, it will stop its propagation as described earlier.

3.3.3 Abort and Stop

When a thread determines that a control speculation is incorrect, it must kill all of the successor threads. The *abort_future* instruction is used to abort the concurrent successor threads. When a thread processing element receives an abort command it will propagate the command to the next thread and discard all of the data in its register file and memory buffer. It then changes its state to idle. After a thread aborts its successor thread, it can fork a new thread with a *fork* instruction.

The *stop* instruction is used by a thread to terminate itself normally. When the thread processing element executes a *stop* instruction, it waits for the *wb_done* flag from the previous thread processing element and then performs the write-back stage automatically. After completing the write-back stage and issuing a *wb_done* flag to the next thread, the thread retires and becomes idle.

3.3.4 Thread Synchronization

There are two flags used to synchronize thread pipelining: the *tsag_done* flag and the *wb_done* flag. The *tsag_done* flag prevents a thread from performing any unsafe loads that may be dependent on the target stores of the previous threads before they finish their TSAG stages. To perform the synchronization, a *release_tsag_done* instruction is placed after the TSAG stage and a *wait_tsag_done* instruction is placed before the first unsafe load operation in a thread.

The *wb_done* flag is used by thread processing elements to serialize the write-back stages. However, the compiler can place a *wait_wb_done* instruction in a thread to force the thread processing element to wait for the *wb_done* flag and to perform the write-back stage without terminating this thread. After executing the *wait_wb_done* instruction, the thread will become the head thread and will start execution from the continuation stage. A thread which executes the last iteration of a loop needs the *wait_wb_done* instruction to wait for the completion of the loop before it can start the thread after the loop.

The superthreaded architecture also provides two explicit synchronization instructions defined as follows:

- *acquire_lock <lock_bit>:* This instruction is used to acquire a lock for entering a critical section guarded by the lock. When executing an *ACQUIRE_LOCK* instruction, the thread processing element will set the specified bit in the shared lock register if it is zero. Otherwise, the thread processing element will wait until the bit is reset by the thread processing element which holds the lock, and then set the bit.

- *release_lock <lock>:* This instruction resets the specified bit in the global lock register. This instruction is used to release the lock acquired before and exit the corresponding critical section.

3.3.5 Example Program

The code segment shown in Figure 3.3 is one of the most time-consuming loops in the SPEC95 integer benchmark *124.m88ksim*. This is a while loop with exit conditions in the loop head as well as in the loop body. There is a potential read-after-write data dependence across loop iterations caused by the variable `minclk`.

```
while ( funct_units[i].class != ILLEGAL_CLASS ) {
    if( f->class == funct_units[i].class ) {
        if ( minclk > funct_units[i].busy ) {
            minclk = funct_units[i].busy;
            j = i;
            if ( minclk == 0 ) break;
        }
    }
    i++;
}
```

Figure 3.3: An example code segment from 124.m88ksim

Figure 3.4 shows the superthreaded code for the loop. In this code, each thread corresponds to a loop iteration. In the continuation stage, each thread increments the recurrence variable `i` and forwards its new value to the next thread processing element using a *store_ts* instruction. The

```
/* Continuation Stage */
L1:
    i_1 = i;
    store_ts(&i,i_1+1);
    fork L1;

/* Target-Store-Address-Generation Stage */
    allocate_ts(&minclk);
    wait_tsag_done;
    release_tsag_done;

/* Computation Stage */
    if (funct_units[i_1].class == ILLEGAL_CLASS ) {
        abort_future;
        i = i_1;
        goto L2;
    }

    if ( f->class == funct_units[i_1].class ) {
        if ( minclk > funct_units[i_1].busy ) {
            store_ts(&minclk, funct_units[i_1].busy);
            j = i_1;

            /* if minclk is zero, break to terminate search */
            if ( minclk == 0 ) {
                abort_future;
                i = i_1;
                goto L2;
            }
        } else
            release_ts(&minclk);
    } else
        release_ts(&minclk);

    stop;

/* Write-back Stage */
/* -> performed automatically after stop */
/* End of thread pipelining */

L2:
```

Figure 3.4: The superthreaded code for the example shown in Figure 3.3

original value of `i` is saved in `i_1` for later use. The continuation stage ends with a *fork* instruction to initiate the successor thread.

In each thread, there is only one target store corresponding to the update of the variable `minclk`. The address of the variable `minclk` is forwarded to the next thread in the TSAG stage. Since the TSAG is not dependent on predecessor threads, it can proceed immediately after the continuation stage. However, the computation stage needs to wait until it receives the *tsag_done* flag from the predecessor thread. This is enforced by the *wait_tsag_done* instruction.

In the computation stage, a thread first checks if the first exit condition is true. If it is true, the thread will abort the successor threads by using an *abort_future* instruction, and then jump out of the loop. Otherwise, the thread will perform the computation of the loop body. In the computation, the update to the variable `minclk` is performed by a *store_ts* instruction, which will forward the result to the successor threads. If the control path that executes the *store_ts* is not taken, the thread will execute a *release_ts* instruction to release the target store entry so that the successor threads will not wait for the target store data. If both exit conditions are false, the thread will execute a *stop* instruction, and then start the write-back, which is performed by the hardware automatically.

Chapter 4

Superthreaded Processor Microarchitecture

The previous chapter introduced the execution model of the superthreaded architecture. In the execution model, threads are initiated and retired in their original order, and that data also flows in the same direction as the thread order. Hence, the thread processing element can be connected by a unidirectional wrap-around ring to simplify the hardware complexity, as shown in Figure 3.1. Each thread processing element has a communication link to its next thread processing element along the ring. The commands and data are always forwarded by a thread to its successor thread in the same direction.

Figure 4.1 shows the detailed block diagram of a thread processing element. A thread processing element consists of four major parts: an execution unit, a communication unit, a memory buffer, and a write back unit. This chapter describes the functions and some design issues of these units.

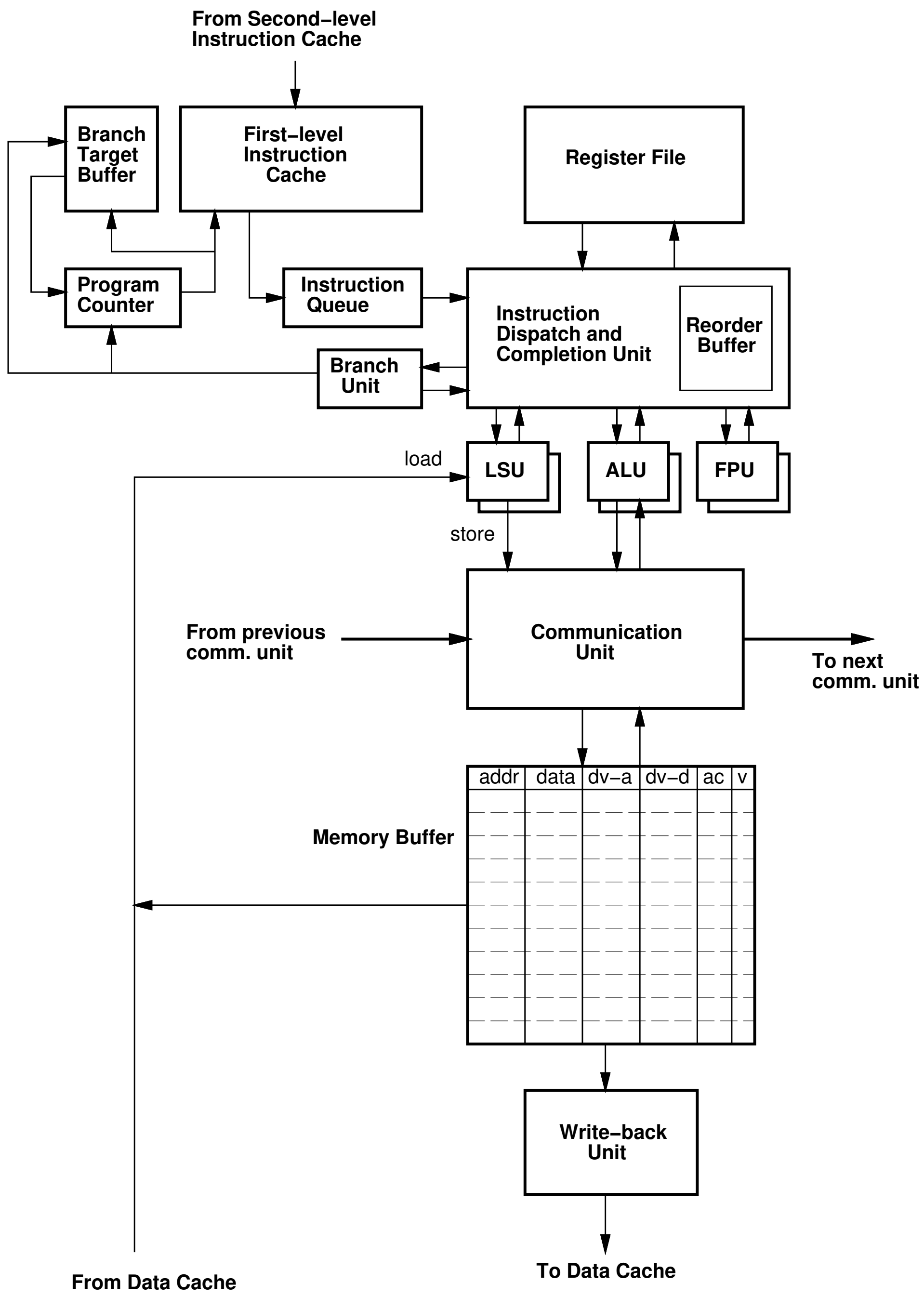

Figure 4.1: The block diagram of the thread processing element

4.1 Execution Unit

The execution unit, like a single-threaded microprocessor, has a program counter, instruction fetch and issue logic, a complete set of registers, and a small number of functional units. To further exploit instruction-level parallelism within a thread, the execution unit can be made to issue multiple instructions per machine cycle. The execution units of different thread processing elements may share some expensive functional units. However, the communication overhead for sharing should be less than the cost of the functional units.

The superthreaded architecture does not support direct data transfer between the local register files of thread processing units. All data transfers between threads must be carried out by means of target store forwarding. However, a thread may need to access some global registers defined by previous threads, such as loop invariants and global pointers. For this purpose, the superthreaded architecture has a shared global register file which can be accessed by all of the thread processing elements. The global register file contains several global registers for holding the global program context and lock register for explicit synchronization between threads. To support speculative execution, we only allow the head thread to update the shared registers except the lock register, which can be updated by all of the thread processing elements as long as they are not speculative. This restriction also enables us to use a single write port for the shared register file. The number of read ports for the shared register file is also flexible, because each thread can copy the values of global registers to its own local register files, and thus reduce the bandwidth requirement for the shared register file.

To provide sufficient instruction bandwidth, each thread processing element has its own first-level (L1) instruction cache. The instruction fetch logic is also equipped with a branch target buffer [24] to eliminate branch delay and to support branch prediction and instruction-level speculative execution. Instructions fetched from the instruction cache are stored in the instruction queue before they are decoded and dispatched.

The instruction dispatch and completion unit decodes instructions from the instruction queue and dispatch them to the reservation stations of the appropriate functional units in order. Instructions can then be executed out of order when their operands are available. To support speculative execution and in-order instruction completion, the instruction dispatch and completion unit uses an reorder buffer [33] to buffer instruction results before they are committed. The reorder buffer also serves as an rename buffer to provide later instructions with uncommitted results which they are flow (read-after-write) dependent on.

In the superscalar execution model, instructions can be executed out of order provided their operands are available. When these dynamic scheduling techniques are applied to a thread processing element, the execution order of instructions assumed by the thread pipelining execution model for run-time data dependence checking and synchronization between threads must be respected. In particular, the following execution order of the superthreaded instructions must be enforced.

- *fork* cannot bypass any *store_ts* instructions in the continuation stage. This is because the continuation stage of the successor thread, which is started immediately after the fork is executed, may depend on those *store_ts* instructions.

- *relase_tsag_done (signal the successor thread the completion of the TSAG stage)* cannot bypass previous *allocate_ts* and *store_ts* instructions. Because the successor thread may perform loads that are dependent on target stores of the current thread after receiving this signal, all target store address should be forwarded to the successor threads for their run-time data dependence checking before this signal is released.

- *store_ts* and *release_ts* cannot bypass its corresponding *allocate_ts* to the same target store address. This order is necessary because the successor threads rely on this order to perform run-time data dependence checking and synchronization.

- load cannot bypass *wait_tsag_done (wait for the predecessor thread completing the TSAG stage)*, and *wait_wb_done (wait for the predecessor thread completing the write-back stage)* instructions. Because a load following a *wait_tsag_done* or *wait_wb_done* may be dependent on stores of the predecessor thread, it must wait until the predecessor thread completes the TSAG stage or writes all store data back to the data cache, respectively.

All of above instructions except load will be forwarded to the communication unit for further processing. For these instructions, above instruction scheduling constraints define the order that must be followed by the dynamic scheduling scheme when forwarding instructions to the communication unit. After instructions are forwarded to the communication unit, the processing order is not a concern because the communication unit processes instructions from the execution units in sequential order. For loads that cannot bypass *wait_tsag_done* or *wait_wb_done*, the load unit will hold the loads until the communication unit signals the completion of these synchronization instructions.

4.2 Communication Unit

The communication unit has an input link from and an output link to the communication units of its two neighboring processing units. It receives and processes commands and data from the execution unit and the up-stream communication unit, and forwards commands and data to the down-stream communication unit. Functions performed by the communication unit include:

- Store target store entries from the up-stream communication unit to the memory buffer, and forward the entries to the down-stream communication unit, if necessary.

- Perform *allocate_ts*, *store_ts*, and *release_ts* commands from the execution unit, and forward the corresponding target store entries to the down-stream communication unit.

- Carry out the *fork* command from the execution unit, and forward all valid target store entries in the memory buffer to the down-stream communication unit before activating the next thread.

- Process the *abort_future* command from the execution unit or the up-stream communication unit, and forward the command to the down-stream communication unit, if necessary.

- Execute the synchronization instructions and the *stop* instruction from the execution unit.

4.3 Memory Buffer

The memory buffer plays an important role in the run-time data dependence checking. All of the target store entries from current and the predecessor threads are stored in the memory buffer. The memory buffer is also used as a write-back buffer for storing the non-target store data to support speculative execution. The memory buffer can be organized as a set-associative or a fully-associative cache. Besides the address tag and the data field, each memory buffer entry includes the following additional fields:

- *Valid bit (V):* a bit that indicates whether this entry is valid or not.

- *Alias Count (AC):* the number of aliases that have the same target store address in the thread.

- *Target Store Distance Vector for Address (TSDV-A):* A vector of N bits for recording which threads have generated a target store address corresponding to this entry, where N is the number of thread processing elements.

- *Target Store Distance Vector for Data (TSDV-D):* A vector of N bits for recording which threads have released a target store data corresponding to this entry.

The TSDV-A and TSDV-D fields are used for handling the output dependences between threads. The memory buffer uses them to keep track of the threads which have generated the target store address, and threads which have already released the data. The initial values of TSDV-A and TSDV-D are all 0's. When a memory buffer entry is allocated to a target store address with a distance count of i, bit i of the TSDV-A field is set to 1.

When the memory buffer receives the data for a target store entry from a previous thread, care must be taken to avoid the target store data from an earlier thread overwriting the target store data from a later thread which is received earlier. In addition, when the execution unit performs a load whose address matches a target store address, the load operation must wait for the data from the most recent store. To ensure the correctness of the memory accesses, the following actions must be taken to receive target store data and to execute load operations.

When the memory buffer receives a target store data entry with a distance count of i:

- If it is a non-null data and the least-significant non-zero bit of the TSDV-D is at bit j ($j = N$ if the TSDV-D is all 0's):
  - If $i < j$, store the target store data to the data field and set bit i of the TSDV-D to 1.
  - if $i >= j$, discard the target store data.

- If it is null data, reset bit i of the TSDV-A to zero, and reset the valid bit if the TSDV-A becomes all 0's.

When the execution unit performs a load operation:

- If the load address matches a memory buffer entry, and its valid bit is set:
  - If bits 1 to $N-1$ of the TSDV-A are all 0's: If the data field is non-null, read the data (it is a store data generated by this thread). Otherwise, access the data cache (this is a read before the first write to the target store entry allocated by this thread).
  - If the least-significant non-zero bit of the TSDV-A$(1 : N-1)$ is bit j: If bit j of the TSDV-D field is also 1, read the data. Otherwise, wait for the target store data from the jth predecessor thread.
- If the load address does not match any valid memory buffer entry, then access the data cache.

The alias count (AC) field is used for handling possible aliases within a thread that cannot be detected at compile time. In the TSAG stage, it is possible that two or more target store addresses generated are the same (*i.e.*, are aliased). When the execution unit executes an *allocate_ts* instruction, the memory buffer will allocate an entry and set the alias count to 1. Later, if the executing unit executes another *allocate_ts* instruction with the same target store address, the memory buffer will increase the alias count by one. When the executing unit performs the *release_ts* or the *store_ts* instruction to release a target store data, the memory buffer will decrease the alias count by one and check if it is equal to zero. If the alias count is greater than zero, which means that this entry is still held by other target stores, the communication unit will not forward the target store entry to the next thread. A target store entry can be forwarded to the next thread only after the alias count becomes zero.

Another potential problem is that the memory buffer may overflow at run-time. When the memory buffer is full and cannot allocate a new entry for a target store or a non-target store, the execution unit will suspend the execution of the thread, and will perform a *wait_wb_done* operation automatically. The *wait_wb_done* operation will cause the thread to wait until it receives the *wb_done* flag from the predecessor thread, when it then becomes the head thread. As the new head thread, it is no longer in speculative mode. Hence, it can write the non-target store data in the memory buffer to the memory, and can remove all target store entries from the predecessor threads, which are now retired. With the memory buffer space released by the non-target stores and the previous target stores, the thread processing element can resume the execution of the thread.

4.4 Write-Back Unit

The write-back unit performs the write-back stage by scanning the memory buffer and writing all store data generated by the thread back to data cache. A memory buffer entry will be written back to the data cache if its valid bit is set and its TSDV-A is all 0's (non-target store) or bit 0 of the TSDV-A is 1 (local target store).

Besides preserving the sequential memory state and supporting the thread-level speculation, performing the thread-level memory write-back in order has the following benefits:

- Output-dependences and anti-dependences between threads caused by non-target stores are eliminated.

- As only one thread processing element can perform memory write-back stage at a time, the shared data cache can eliminate the arbitration hardware of the write port, which improves the throughput of memory write-back.

Chapter 5

Compilation for the Superthreaded Architecture

To fully utilize the hardware support for thread-level speculative execution and run-time data dependence checking, the superthreaded architecture relies on the compiler to extract thread-level parallelism and to generate superthreaded codes. Given a sequential program, the compiler first partitions the execution of the program into threads for concurrent execution. The compiler then performs thread pipelining to facilitate run-time data dependence checking. Both tasks require powerful program analysis and transformation techniques. Fortunately, many of these techniques have been developed in traditional parallelizing compilers. We can leverage those techniques for superthreaded processors.

In addition to generating superthreaded codes, the compiler can further enhance parallelism between threads and reduce run-time data dependence checking overhead by applying some specific program transformations for superthreaded processors.

In this chapter, we first present the program analysis and transformation techniques for program partitioning and thread pipelining, and then describe the advanced program transformation techniques to enhance the performance of superthreaded processors.

5.1 Basic Compiler Techniques for Superthreading

The compiler requires extensive data flow and dependence information to partitions a program at the appropriate level of granularity and to generate efficient superthreaded codes. The compiler can use the following traditional program analysis and transformation techniques to extract the desired information from programs, and to reduce the run-time overhead for data dependence checking.

- *Function inlining:* If a thread includes function calls that may cause data dependences between threads, the compiler can use function inlining to allow data flow analysis and instruction movement across function boundaries for thread pipelining.

- *Induction variable substitution:* A loop may have several induction variables, which are variables incremented or decremented by a constant value in each iteration. They will cause loop-carried data dependences. The compiler needs to generate target store instructions for those induction variables. To reduce the number of target stores, the compiler can select one induction variable as the primary induction variable, and replace other induction variables with linear functions of the primary induction variable. By applying such induction variable substitution, the compiler needs to generate only one target store for the primary induction variable. The overhead for data forwarding and synchronization can be greatly reduced.

- *Alias analysis:* Programs written in C or C++ usually use many pointers and dynamic data structures. Without any alias analysis, the compiler must assume all pointers are possible aliases when it analyzes data dependences and identifies target stores. Although the superthreaded architecture can handle aliases by performing run-time data dependence checking, more precise alias information will result in fewer target stores generated.

- *Data dependence analysis:* Data dependence analysis is a very important technique used in parallelizing compilers to determine if a loop should be executed in parallel. For superthreaded processors, data dependence analysis can help to identify target stores for run time data dependence checking and synchronization.

- *Variable privatization:* A private variable is a variable whose updated value does not reach a later thread, even though the variable may be written and then read in a later thread. The compiler need not generate target store instructions for the update of the private variables because they will not cause read-after-write data dependence across threads. For this purpose, the compiler can perform analysis (on both scalars and arrays) to identify privatizable variables [53, 25].

- *Loop unrolling/interchange:* Loop unrolling and loop interchange are common techniques used in parallelizing compilers to increase the granularity of parallel loops. In the thread pipelining execution model, by applying loop unrolling or loop interchange, we can increase the size of the computation stage, which will be executed in parallel, and thus enable a larger portion of threads to be overlapped.

5.2 Program Partitioning

To achieve the best performance gain from superthreading, the compiler should partition a program at a level where sufficient thread-level parallelism is available. On the other hand, since each thread processing unit can also be superscalar, there should be sufficient instruction-level parallelism left in each thread. Therefore, the compiler must generate threads at the appropriate level where the maximum combined performance gains can be achieved from both superthreading and superscalar. In addition, the size of a thread cannot be too large in order to avoid overflowing the memory buffer [43].

To generate threads at the right level, the compiler examines data dependences between contiguous execution portions of a program and estimates the amount of thread-level parallelism at each level with respect to the data dependences. If the program has more than one level of loops that have a good amount of parallelism, the compiler will schedule the outer loop that would not overflow the memory buffer for thread-level execution, and leave the inner loop for each thread processing unit to exploit instruction-level parallelism. In case that only inner-most loops are suitable for thread-level execution, the compiler can perform loop unrolling or loop blocking to increase the size of each thread, and to provide each thread processing unit with sufficient workload to exploit instruction-level parallelism. Another technique to optimize program partitioning is loop interchange, which can be used either to interchange an outer parallel loop that will overflow the memory buffer with a inner sequential loop to allow the new inner loop to be executed in parallel, or to interchange a small inner parallel loop with an outer sequential loop to increase the size of each thread.

5.3 Thread Pipelining

After partitioning a program into threads, the compiler then further partitions each thread into pipelined stages to facilitate thread continuation and run-time data dependence checking. To facilitate thread pipelining, the compiler first analyzes data dependences between contiguous threads and marks store operations in each thread that could be data dependent by its concurrent successor threads as target stores. The compiler then moves instructions that are needed to compute the target store addresses, and their dependent instructions to the TSAG (target-store-address-generation) stage. For each target store address computed in the TSAG stage, the compiler places an *allocate_ts* instruction in the TSAG stage to forward the target store address to the concurrent successor threads. The target store addresses computed in the TSAG stage can be saved in registers and used in the following computation stage.

After the TSAG stage is formed, the remaining computation in a thread is its computation stage. The compiler replaces each last-store operation to a target store address with a *store_ts* instruction. In addition, if a control path taken at run-time will prevent some target store entries from being updated, the compiler needs to insert *release_ts* instructions in the control path to release these target store entries.

For threads that have many possible control paths, there may be many target store addresses that need to be generated at the TSAG stage, but only a few of them will actually be used in the computation stage. In this case, the overhead for the target store address generation might overshadow the performance gain from the parallel execution of multiple threads. To reduce the overhead in target store address generation, the compiler can move the condition tests to the TSAG stage and generate only target store addresses that will be used in the chosen control path. Like the target store addresses, the branch conditions precomputed at the TSAG stage can be saved in registers for later use in the computation stage.

The final step for thread pipelining is to extract the continuation stage from the TSAG stage. The continuation stage contains the computation that is essential to start the next thread. The data generated in the continuation stage will be forwarded directly to its successor threads before they are activated. Therefore, a thread can immediately load and use the data generated in the continuation stage of the previous thread without waiting for the previous thread to complete the TSAG stage. However, because the continuation stages of the contiguous threads are executed sequentially, the compiler must keep the continuation stage as small as possible. With these considerations, there are two criteria for instructions to be moved to the continuation stage. First, they only depend on the continuation stages of the predecessor threads. Second, they are needed in the TSAG stages of the current thread and the successor threads. In general, the continuation stage contains instructions to update the value of recurrent variables such as the loop index, which are usually needed for target store address computation.

After the continuation stage of a thread is extracted from its TSAG stage, there still may be instructions in the TSAG stage which are data dependent on its concurrent predecessor threads. In this case, the compiler needs to place a *wait_tsag_done* instructions before those instructions to delay their execution until the predecessor threads complete their TSAG stages and forward all of their target store addresses.

In the thread pipelining, the compiler will try to increase the execution overlap of concurrent threads by minimizing the stall caused by data dependences between threads. To do this, we can perform statement reordering [6] and schedule target stores as early as possible, or schedule load instructions that may be data dependent on the target stores of some predecessor threads as late as possible.

5.4 Advanced Compiler Techniques for Superthreading

5.4.1 Conversion of Data Speculation to Control Speculation

Some concurrent multiple-threaded architectures, such as *Multiscalar* [11, 36] and *SPSM* [8], provide hardware support for data speculation. With such hardware support, the compiler can speculate on potential data dependences between threads by assuming that they would not be violated, and rely on the hardware to detect violations at run-time. A dependence violation is detected if a thread writes to a memory location after a later thread has read from the same location. When this happens, the hardware will squash the later thread and all of its successor threads.

Hardware support for data speculation can be very expensive, because it needs a buffer (called *Address Resolution Buffer* in multiscalar) to keep all *load* and *store* addresses from all active threads in order to detect data dependence violations. To avoid using expensive hardware for data speculation such as the address resolution buffer, and to avoid wasting useful computation when a thread is squashed due to data dependence violation, the superthreaded architecture *enforces*, rather than *speculates*, data dependences between threads. It uses the memory buffer to perform both run-time data dependence checking and implicit data synchronization, and requires only store addresses and data to be buffered. Since there are far fewer store addresses than load addresses, the size of the memory buffer can be much smaller than the the address resolution buffer in multiscalar.

However, data speculation can be useful for some programs. For example, data dependences may occur only in certain control paths within each thread. By using data speculation, we can exploit the potential parallelism if the control paths that contain the data dependences are not taken at runtime.

For threads that that can benefit from this kind of data speculation, the compiler can perform the data speculation by converting it to control speculation, which is supported by the su-

```
while ((c = getbyte()) != EOF ) {
    in_count++;
    fcode = (long)(((long) c << maxbits)+ent);
    i = ((c << hshift) ^ ent);

    if ( htabof (i) == fcode ) {
        ent = codetabof (i);
        continue;
    } else if ((long)htabof (i) < 0)
        goto nomatch;
probe:
    . . .
    if ( htabof (i) == fcode ) {
        ent = codetabof (i);
        continue;
    }
    if ( (long)htabof (i) > 0 )
        goto probe;
nomatch:
    /* computation on which the successor
       threads are dependent  */
     . . .
}
```

Figure 5.1: An example code from 129.compress

```
/* Continuation Stage */
loop_continue:
    FORK(loop_continue);

/* TSAG Stage - generate only ts-addresses
    in speculated paths */
    ALLOCATE_TS(&ent);
    ALLOCATE_TS(&in_count);
    RELEASE_TSAG_DONE;
    WAIT_TSAG_DONE;

/* Computation Stage */
    c = getbyte();
    if ((c = getbyte()) == EOF ) {
        /* control speculation incorrect */
        ABORT_FUTURE;
        goto loop_exit;
    }
    STORE_TS(&in_count, in_count+1);
    fcode = (long) (((long) c << maxbits) + ent);
    i = ((c << hshift) ^ ent);
    if (htab[i] == fcode) {
        STORE_TS(&ent,codetab[i]);
        STOP;    /* thread retires as expected */
    } else if ((long)htabof (i) < 0)
        goto nomatch;
probe:
    . . .
nomatch:
    /* data speculation incorrect */
    ABORT_FUTURE;
    . . .
    goto loop_continue;
loop_exit:
    . . .
```

Figure 5.2: The superthreaded code of Figure 5.1 using conversion of data speculation into control speculation

perthreaded architecture. To speculate on a read-after-write data dependence that will occur only if the predecessor thread takes certain control paths that execute the write, the compiler does not generate any target store addresses for the write. Instead, the compiler inserts an *abort_future* instruction in the control paths that execute the write. If the predecessor thread does take that control path, the *abort_future* instruction will be executed and the successor thread that depends on the write will be squashed and re-executed.

The WHILE loop shown in Figure 5.1 is from the SPEC95 integer benchmark program *129.compress*. In this loop, each iteration computes the value of `fcode` and check if it matches the value of the corresponding entry in the hash table (`htabof`). If it matches, the loop continues. Otherwise, the loop iteration will update the hash table and introduce a lot of data dependences to successor iterations. At run time, most of loop iterations will not update the hash table. To generate superthreaded code for this loop, we can speculate on the data dependences caused by the update of the hash table with control speculation. The corresponding superthreaded code is shown in igure 5.2. In the superthreaded code, we only generate target store instructions for `ent` and `in_count` and ignore the potential data dependences caused by the update of the hash table. However, if at run time the control flow of a thread actually goes to `nomatch` and updates the hash table, the thread will execute an ABORT_FUTURE instruction to squash the successors which may data depend on the updated hash table.

5.4.2 Distributed Heap Memory Management

Programs written in C often need to allocate dynamic memory space at runtime. Dynamic memory space is allocated and deallocated through the heap memory manager, such as *malloc* and *free* functions provided by the standard C library. The heap memory manager usually uses a free list to keep track of the free memory blocks available for allocation. The allocation or deallocation of a memory block will modify the free list and the memory blocks. Such heap memory management can cause potential data dependences between threads and create a bottleneck for thread parallelization.

To eliminate such data dependences, we can distribute the management of the dynamic heap memory to each thread processing element; that is, each thread processing element maintains its own free list to keep track of the free memory blocks owned by the thread. With the heap memory management distributed to each thread processing element, the allocation and deallocation of memory blocks on concurrent threads can be made to be independent of each other and can be executed in parallel.

To support distributed heap memory management, we need to modify the heap memory management routines. The new heap memory management routines will maintain a free memory block list for each thread processing element. When a heap memory management routine is called, it must be given the ID of the thread processing element. The routine will then allocate or deallocate a memory block from or to the free list associated with the thread processing element. Note that the distributed heap memory management still uses a single heap memory space. A memory block allocated by one thread processing element can be used and deallocated by another thread processing element. The multiple free lists may become unbalanced after the program has run for a while. If any free list runs out of its free memory blocks, the heap memory manager will invoke a re-distribution routine to balance the number of free memory blocks among the thread processing elements.

The FOR loop shown in Figure 5.3 is from the SPEC95 integer benchmark program *130.li*. This loop traverses the link list `p` and frees any memory space pointed by `p`. The deallocation of heap memory space will cause data dependences between loop iterations on the system free list. To generate superthreaded code for this loop, we can use the modified heap memory management routines described above to deallocate heap memory space as shown in Figure 5.4. In the superthreaded code, we deallocate heap memory space by using function call to the new version of free library `st_free`, which maintains a free list for each thread processing element. The `st_free` has an additional parameter which indicates the the ID of the current thread processing element. The `st_free` will free the memory space to the free list associated with the thread processing element.

5.4.3 Using Critical Section for Order-independent Operations

Data dependences between threads are often caused by order-independent operations on a shared variable or data structure. Examples of order-independent operations include adding a value to a variable (reduction operations) and inserting a node to a list in which the order is irrelevant. Using target stores to enforce order-independent operations will serialize the concurrent execution of the multiple threads. To avoid such a problem, we can place order-independent operations in critical sections.

There are two requirements to ensure the correct execution of critical sections for order-independent operations. First, the results of the order-independent operations should bypass the memory buffer and be written directly into the data cache so that the other threads can get the updated data. Second, only non-speculative threads (threads which will not be aborted by its predecessor threads) can enter the critical section and perform order-independent operations out of order. This is to prevent a speculative thread from performing an order-independent operation and writing uncommitted results to the data cache.

```
for (n = seg->sg_size; n--; p++) {
    if (!(p->n_flags & MARK)) {
        switch (ntype(p)) {
          case STR:
            if (p->n_strtype == DYNAMIC &&
                p->n_str != NULL){
                total -= strlen(p->n_str)+1;
                free(p->n_str);
            }
            break;
          case FPTR:
            if (p->n_fp)
                fclose(p->n_fp);
            break;
          case VECT:
            if (p->n_vsize) {
                total -= p->n_vsize*sizeof(NODE **);
                free(p->n_vdata);
            }
            break;
        }
        p->n_type = FREE;
        p->n_flags = 0;
        rplaca(p,NIL);
        rplacd(p,fnodes);
        fnodes = p;
        nfree++;
    }
    else
        p->n_flags &= ~(MARK | LEFT);
}
```

Figure 5.3: An example code from 130.li

```
/* Continuation stage  */
loop_continue:
    if (!(n+1)
        goto loop_exit;
    STORE_TS(&p,p+1);
    STORE_TS(&n,n-1);

    FORK(loop_continue);

/* TSAG stage */
     . . .

/* Computation stage */
    tpe_id = get_thrd_unit_id();
    if (!(p->n_flags & MARK)) {
        switch (ntype(p)) {
        case STR:
          if (p->n_strtype == DYNAMIC && p->n_str != NULL) {
              STORE_TS(&total, total - strlen(p->n_str) - 1);
            st_free(tpe_id, p->n_str);
          }
          break;
        case FPTR:
          . . .
        case VECT:
          if (p->n_vsize) {
              STORE_TS(&total, total - p->n_vsize*sizeof(NODE **));
              st_free(tpe_id, p->n_vdata);
          }
          break;
        }
        . . .
    } else
        . . .
    STOP;
loop_exit:
    . . .
```

Figure 5.4: The superthreaded code of Figure 5.3 using distributed heap memory management

5.4.4 Memory Buffering in the Main Memory

We use memory buffers to save target store addresses and data for run-time data dependence checking, and to buffer uncommitted store data generated during speculative execution. The memory buffer can also be used to store private variables and to eliminate anti- and output-dependences caused by private variables between threads. Due to the limitation of hardware resources, the memory buffer will have a fixed size. The compiler can estimate the usage of the memory buffer and partition threads accordingly to avoid memory buffer overflow. When a memory buffer overflows, the thread must be stalled until all of its predecessor threads are completed before it can resume execution.

To exploit parallelism from a large outer loop and to avoid memory buffer overflow, the compiler can buffer the uncommitted store data as well as the private store data in the main memory. For this purpose, a special store instruction is provided which can bypass the memory buffer in the thread processing unit and directly write the data to the data cache.

While a thread may use the cache or the main memory to buffer uncommitted and private store data, it still needs the memory buffer in the thread processing unit to store target store addresses and data for run-time data dependence checking. When a thread becomes the head thread and has completed its computation stage, it has to perform the write-back for the store data buffered in the main memory.

Chapter 6

Performance Evaluation

6.1 Methodology

We evaluate the performance of the superthreaded architecture using a trace-driven simulator. The simulation environment is shown in Figure 6.1 We transform benchmark programs into their corresponding superthreaded programs at the source level. In the transformed superthreaded programs, special superthreading instructions, such as *fork* and *store_ts*, are represented as function calls to specific subroutines. The transformed superthreaded program is compiled by the SGI C compiler. The program is then instrumented by *Pixie* [35] to generate instruction and memory reference traces.

The simulator executes the instrumented program on the host SGI machine and collects the traces generated by the program. During the trace collection phase, the function calls which represent the superthreading instructions will be converted to the actual superthreading instructions for simulation. The simulator can also recognize the starting of a new thread, which is marked by a special function call inserted in the transformed program. After detecting the starting mark of a

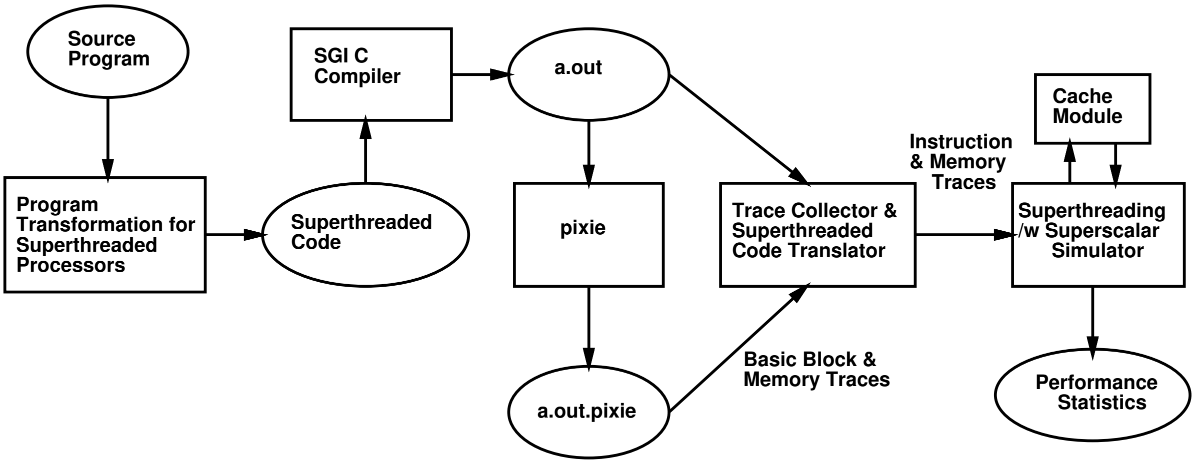

Figure 6.1: Simulation environment for the superthreaded architecture

new thread, the simulator will assign the instruction traces following the mark to the next thread processing element. Using this mechanism, we can generate multiple-threaded instruction trace streams from a single-threaded instruction trace stream and simulate their execution on multiple thread processing elements.

After the trace collection phase, the converted instruction traces are fed into the corresponding thread processing elements in the simulator. The instruction fetch and execution portion of a thread processing element. is organized similar to a superscalar processor with a branch target buffer for branch prediction and a reorder buffer for out-of-order execution. Functional units are fully pipelined and have the same latencies as those of MIPS R10000.

The interconnection between thread processing elements is modeled as a unidirectional ring. Each communication unit can forward a configurable number of commands or target store entries per cycle to the down-stream communication unit. The memory buffer is fully-associative and has a configurable number of ports. The memory buffer uses a two-stage pipeline structure, so every read and write to the memory buffer will take two cycles.

The simulator also includes data cache module, which is shared by all thread processing elements. To provide sufficient cache access bandwidth, the cache is non-blocking and can be made of multiple-port, multiple-bank, or both. In the following performance evaluation results, the basic configuration for the data cache is given as follows: 64 kbytes, write back, 4-way set associative, 16-byte block, two read ports and one write port. A memory access hit in the cache takes two cycles, while the cache miss penalty is 12 cycles. The number of interleaved banks is scaled up with the total issue rate (number of thread elements $\times$ issue rate per thread element) of the processor.

6.2 Benchmarks

We use two GNU utilities (*wc* and *cmp*), two SPEC92 floating pointer programs (*alvinn* and *ear*) and four SPEC95 integer programs (*m88ksim*, *gcc*, *compress*, and *ijpeg*) for our performance study. All of these programs are written in C. Since the compiler development for the superthreaded processors requires both a front end parallelizing compiler and a back end optimizing compiler and is still under way (see next chapter), we manually transformed the most time-consuming routines of those programs into the superthreaded codes at the source level. This approach allows us to evaluate and study various architectural issues, design parameters, and the feasibility of the superthreaded execution model before its compiler is fully functional.

To faithfully mimick the action of the superthreaded compiler, we limited ourselves to use only those compiler techniques described in Chapter 5 during manual transformation. Table 6.1 summarizes the program transformation techniques that we actually employed for the benchmarks.

Even though we try to mimick compiler algorithms as closely as possible, the transformed programs should still be interpreted as the results of a very good compiler with profiling information. On the other hand, the manual transformations are applied only at the source code level, and the

Transformations	wc	cmp	alvinn	ear	m88ksim	gcc	compress	ijpeg
Function inlining							√	
Induction variable substitution	√	√	√		√			√
Loop unrolling	√	√	√	√	√			
Loop interchanging			√					
Statement reordering to increase overlap	√				√	√	√	
Converting data speculation to control speculation						√	√	
Distributed heap memory management						√		
Critical section for order-irrelevant operation	√					√		
Memory buffering in the main memory						√		

Table 6.1: Program transformations used in manually transformed programs

SGI back end optimizing compiler is used to exploit instruction level parallelism without any specific consideration for the superthreaded execution model. Some performance degradation could be resulted from such omission. We thus restrict our performance comparison in varying architectural and design parameters, and not against the other CMA models.

Table 6.2 shows the input sets used for the simulation, the dynamic instruction counts of the original programs, and the dynamic instruction counts of those most time-consuming routines before and after they are transformed into the superthreaded form. Table 6.2 also shows the IPCs of the original programs run a single-threaded, single-issued processor model, which is used as our base model for computing speedup.

Program	Input set	Insn. Count Total (original)	Insn. Count Transformed (before)	Insn. Count Transformed (after)	Percent Increase	Base Model IPC
wc	expr.c from gcc	3.98M	3.97M (99.7%)	6.15M	54.9%	0.69
cmp	expr.c from gcc	5.13M	5.12M (99.8%)	5.27M	10.9%	0.78
052.alvinn	ref(20 iterations)	613.6M	543.5M (88.6%)	559.4M	2.9%	0.68
056.ear	short	812.7M	802.0M (98.6%)	846.9M	5.6%	0.89
124.m88ksim	train	195.2M	148.0M (75.8%)	156.7M	5.9%	0.71
126.gcc	jump.i	218.0M	64.7M (29.8%)	64.8M	0.2%	0.64
129.compress	train	50.6M	46.4M (90.2%)	52.1M	12.3%	0.75
132.ijpeg	test	856.1M	598.1M (69.9%)	711.8M	19.0%	0.81

Table 6.2: Benchmark programs and their dynamic instruction counts

6.3 Performance Comparison for Single Issue Models

Table 6.3 shows the speedups of a superthreaded processor, with 4 or 8 thread processing elements, over a single-threaded, single-issue processor for the most time-consuming routines in each of the benchmark programs. The superthreaded processor and the single-threaded processor use the same hardware configurations in instruction execution. However, the single-threaded processor simulator executes the original benchmark programs, while the superthreaded processor simulator executes the transformed superthreaded codes, which have more run-time overhead than the original programs as shown in Table 6.2.

Wc and *cmp* spend most of their execution time in one loop. Both loops are very small and have some loop-carried data dependences. For these two loops, we performed loop unrolling and used target store instructions as well as critical sections to enforce data dependences. Even with the loop-carried data dependences, these two programs can achieve good speedups in the superthreaded execution model.

The two SPEC92 floating pointer programs *052.alvinn* and *056.ear* spend most of their execution time in several parallel loops. By executing iterations of these loops concurrently on multiple thread processing elements, we can obtain very good speedups for these two programs. To increase the granularity of threads, we perform loop unrolling for some of these loops. We also perform loop interchanging for Routine (4) *hidden_input* in *052.alvinn* to avoid memory buffer overflow. In *124.m88ksim*, the routines that have been transformed are Routine (1) *killtime*, Routine (2) *ckbrkpts*, Routine (3) *alignd*, Routine (4) *test_issue*, and Routine (5) *loadmem*. Both Routine (2) *ckbrkpts* and Routine (4) *test_issue* have a small WHILE loop with exit conditions. These loops have a small number of iterations before they exit, and therefore their speedups are very limited. Routine (1) *Killtime*, Routine (3) *alignd*, and Routine (5) *loadmem* have more speedups because their loops have more iterations. However, there are cross-iteration data dependences in the loops of Routine (3) *alignd*, which cause the speedup to be saturated when more than 4 thread processing elements are used.

For *126.gcc*, we generated the superthreaded codes for the two most time-consuming passes: Routine (1) *cse* (common subexpression elimination) and Routine (2) *regclass* (register class). For Routine (1) *cse*, we exploit the thread-level parallelism for the outermost loop, which processes an

Program	wc		cmp		alvinn		ear	
# of TPEs	4	8	4	8	4	8	4	8
Routine(1)	1.95	2.71	3.34	6.15	3.74	7.48	3.02	4.53
Routine(2)					3.70	6.62	3.70	5.57
Routine(3)					3.69	6.62	3.54	5.31
Routine(4)					3.40	6.90	3.66	5.47
Routine(5)					1.47	1.47	3.32	4.98
Routine(6)					3.89	7.76	3.38	5.09
(1) - (6)	1.95	2.71	3.34	6.15	3.61	7.17	3.53	5.29

Program	m88ksim		gcc		compress		ijpeg	
# of TPEs	4	8	4	8	4	8	4	8
Routine(1)	3.23	5.03	1.31	1.39	1.04	1.04	3.83	6.87
Routine(2)	1.25	1.28	2.05	3.27	1.26	1.26	3.48	5.59
Routine(3)	2.06	2.01					1.60	1.64
Routine(4)	1.15	1.15					3.73	5.24
Routine(5)	3.77	7.53					2.69	4.22
Routine(6)							2.35	4.21
Routine(7)							2.86	4.81
(1) - (7)	2.71	3.52	1.37	1.50	1.13	1.13	2.91	4.35

Table 6.3: Speedups of a superthreaded processor over a single-thread processor

extended basic block in each iteration. To exploit parallelism between iterations, data speculation is necessary because basic block boundaries could be modified during the process. We converted the data speculation to control speculation to exploit its potential parallelism. However, at the run-time about one-third of the basic blocks are modified and need to be re-processed. Hence, only a small speedup is obtained. In Routine (2) *regclass*, there is a major loop which scans all instructions and computes the cost for pseudo-registers. This loop only has a few data dependences between loop iterations. We enforce the data dependences by using target store instructions and critical sections. The speedup for this loop turns out to be reasonably good. One notable observation is that most of the innermost loops in *gcc* are inherently sequential due to the algorithm used in *gcc*. The speedup obtained from software pipelining on these innermost loops is known to be be very small.

In *129.compress*, most of the execution time is spent in two loops, one is in Routine (1) *compress* and the other one is in Routine (2) *decompress*. These two loops have very limited loop-level parallelism because each iteration may be data-dependent on the results computed at the very end of the previous iteration. Although we can speculate on the data dependences with control speculation, only limited speedups can be obtained from these loops.

132.ijpeg is an image compression program that has a lot of parallel loops or loops with only a few loop-carried data dependences. By using the target store mechanism, we can successfully partition the execution of those loops into concurrent threads and achieve very good speedups.

6.4 Performance Comparison for Multiple-issue Models

To compare models with different issue rates and different number of thread processing elements, we run the benchmark programs on the simulator with the number of thread processing elements

# of TPEs ×	1	1	1	1	2	2	2	2	4	4	4	4	8	8	8	8	1	16
Issue rate	1	2	4	8	1	2	4	8	1	2	4	8	1	2	4	8	16	1
Reorder buffer size	8	16	32	64	8	16	32	64	8	16	32	64	8	16	32	64	128	8
Pending cond. branches	1	1	2	4	1	1	2	4	1	1	2	4	1	1	2	4	8	1
ALUs	1	2	4	8	1	2	4	8	1	2	4	8	1	2	4	8	16	1
FPUs	1	1	2	4	1	1	2	4	1	1	2	4	1	1	2	4	8	1
LSUs	1	1	2	4	1	1	2	4	1	1	2	4	1	1	2	4	8	1
Comm. bandwidth	-	-	-	-	1	1	2	2	1	2	2	3	2	2	3	4	-	2
Write back bandwidth	1	1	1	2	1	1	2	4	1	2	4	8	2	4	8	16	4	4
Data cache banks	1	1	1	2	1	1	2	4	1	2	4	8	2	4	8	16	4	4

Table 6.4: Configurations for thread processing elements of various processor models

ranging from 1 to 8, and the issue rate within each thread processing element ranging from 1 to 8. Models with a single-threaded processing element are equivalent to a comparable superscalar processor. Note that we run the original benchmark programs on single-threaded models, and run the transformed superthreaded programs, which have more run-time overhead than the original programs, on multiple-threaded models.

In every model, each thread processing element uses a 1024-entry, 4-way associative branch target buffer and a 96-entry fully-associative memory buffer. The detailed simulation configuration for each model is given in Table 6.4. In general, the reorder buffer (instruction window) size, the level of branch speculation over unresolved conditional branches, and the number of functional units are scaled up with the issue rate of each thread processing element, while the communication bandwidth, write back bandwidth (for store data), and the number of interleaved cache banks are scaled up with the total issue rate of each combination.

To compare the relative performance (speedup) among different superthreaded configurations, and between superthreaded processors and superscalars, we use a single-threaded, single-issue processor as our baseline. This allows us to see the improvement due to superscalar approach, and the additional improvement obtained through superthreading. We do not use IPC for the performance comparison because multiple-thread models have relatively higher IPCs than single-thread models (due to the instruction overhead in the superthreaded programs). Figures 6.2 and 6.3 show the speedup of different configurations. For each benchmark program, we show both the speedup of the entire program and the speedup of the transformed portion of the program.

From Figures 6.2 and 6.3, we can see that the superthreaded model can further improve the performance of a single-threaded (1 TPE) superscalar architectural model for most of the benchmark programs. For programs with high thread-level parallelism such as 052.alvinn and 056.ear, the combined speedup from superscalar and superthreading can exceed 10. On the other hand, programs with intensive loop-carried data dependences such as 129.compress cannot benefit much from the superthreaded execution model.

Overall, while the single-threaded model with 4-issue superscalar achieves an average speedup (entire program) of 2.34, the superthreaded model can further improve the average speedup to 4.04 (73% improvement) with 4 TPEs and to 4.6 (97% improvement) with 8 TPEs. For models with 8-issue per TPE, the superthreaded models can improve the average speedup from 3.18 to 4.97 (56% improvement) with 4 TPEs and to 5.5 (73% improvement) with 8 TPEs. If we consider only the transformed portion of the programs, the superthreaded models with 8 TPEs can improve the average speedup from 2.36 to 5.63 (139% improvement) for 4-issue per TPE and from 3.18 to 6.43 (102% improvement) for 8-issue per TPE.

Figure 6.4 compares speedup (transformed portion) of different models with a total issue rate of 16. We can see that models combining superthreading and superscalar (4-TPE×4-Issue or 8-TPE×2-Issue) can achieve better speedup than the pure superscalar model (1-TPE×16-Issue). In

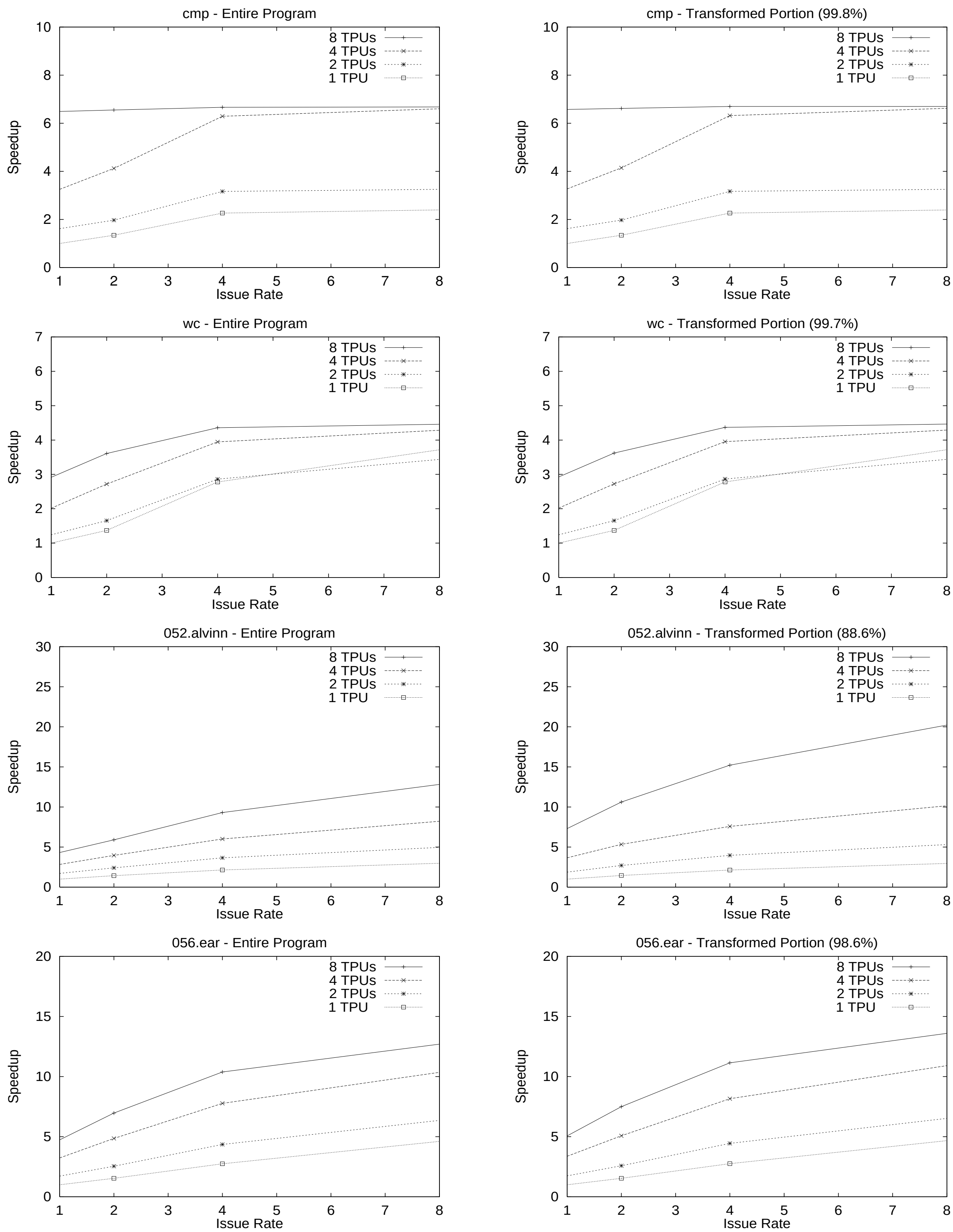

Figure 6.2: Performance comparison for wc, cmp, 052.alvinn and 056.ear

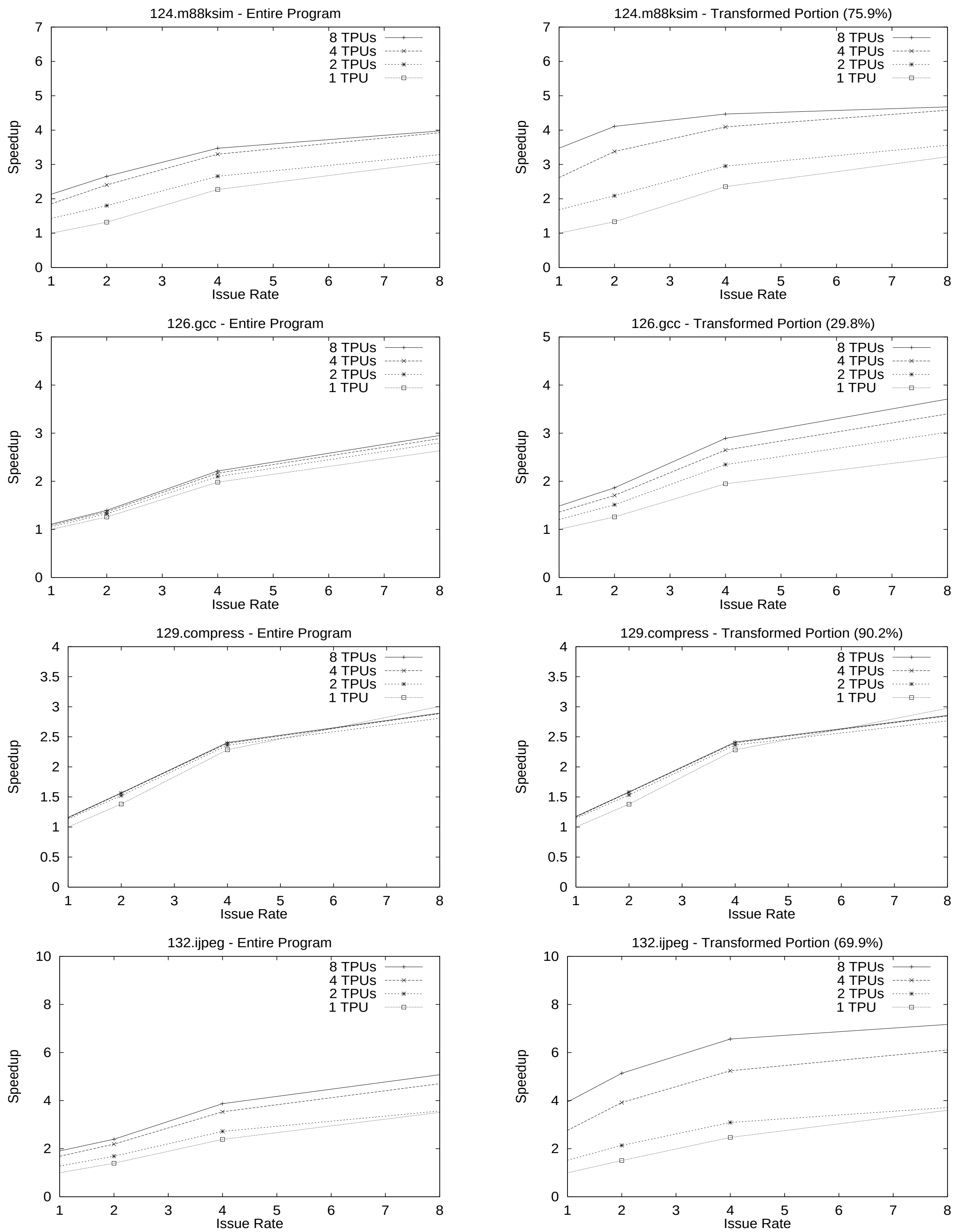

Figure 6.3: Performance comparison for 124.m88ksim, 126.gcc, 129.compress and 132.ijpeg

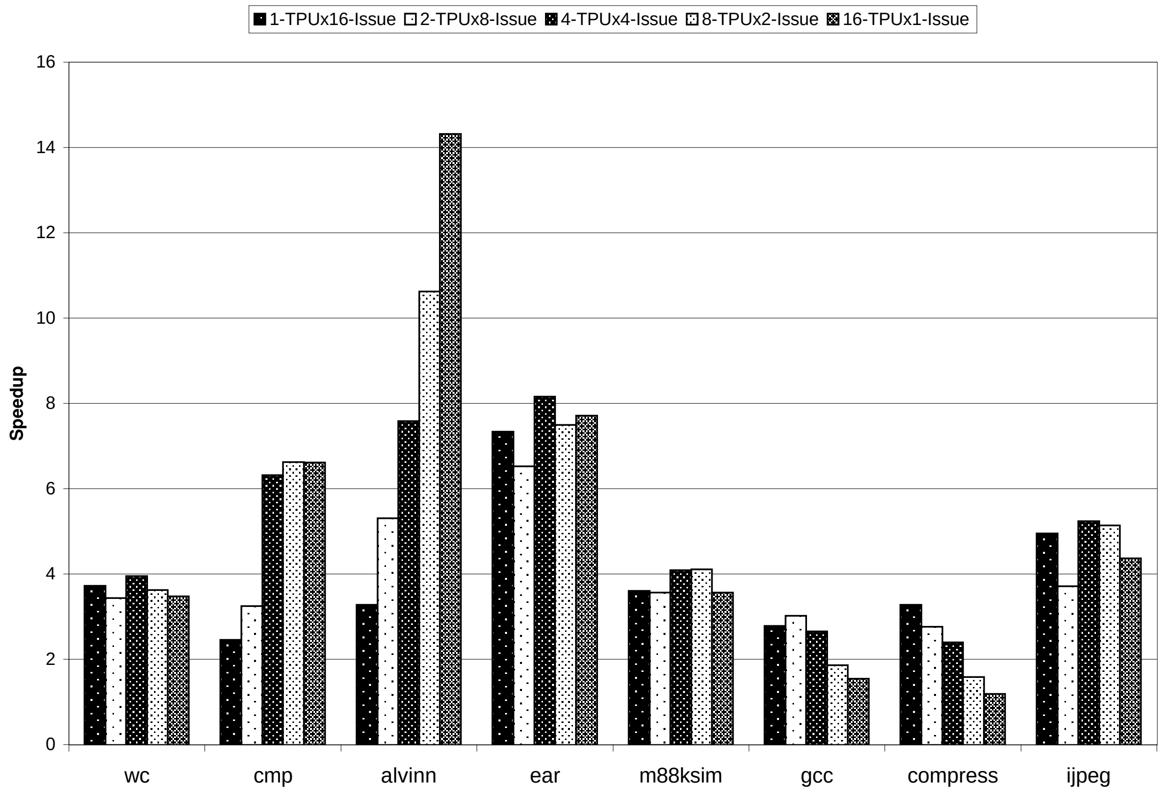

Figure 6.4: Performance comparison of models with a total issue rate of 16

addition, the clock cycle time of the pure superscalar model could be longer than that of the combined models, because the 16-issue superscalar model uses a large (128 entries) instruction reorder buffer for dynamic instruction scheduling and supports higher level (8 branches) of branch prediction and speculation.

6.5 Memory Buffer Usage

The memory buffer of each thread processing element could be very expensive because it needs to use associative hardware for run-time data dependence checking. For this reason, the memory buffer cannot be too large. Table 6.5 shows the maximum number of memory buffer entries used in each benchmark program executed in a superthreaded processor with different numbers of thread processing elements. Note that the memory buffer may collect more target store entries from the predecessor threads as the number of thread processing elements increases. For *gcc*, speculative and private data is stored in the main memory. The numbers shown in Table 6.5 do not include such entries in the main memory. The results show that using a small number of memory buffer entries (no more than 100) to buffer target stores and local stores can be sufficient and effective for most programs.

6.6 Communication Bandwidth Requirement

The communication units and the unidirectional ring allow a thread to receive commands and target stores from its predecessor threads, and to forward commands and target stores to its successor threads for thread initiation/synchronization and run-time data dependence checking. To keep the thread initiation and the run-time data dependence checking from becoming the critical path in the concurrent multithreaded execution, a communication unit may need to process and forward more

Program	Number of TPEs		
	2	4	8
wc	4	4	4
cmp	4	4	4
052.alvinn	72	73	73
056.ear	25	25	25
124.m88ksim	26	26	26
126.gcc	3	3	3
129.compress	20	24	32
132.ijpeg	26	26	26

Table 6.5: Maximum number of memory buffer entries used in each program

than one request per cycle as the issue rate and the number of thread processing elements increase. However, increasing the bandwidth of communication units could be very expensive. Therefore, it is very important to know the bandwidth requirement of the communication unit for the benchmark programs.

In the communication bandwidth study, we use a superthreaded processor model consisting of four 4-issue thread processing elements, which we think is a reasonable configuration for the next generation of microprocessors, as our base model. We run the benchmark programs on the base processor model with different communication bandwidths ranging from 1 request per cycle to 4 requests per cycle. The simulation result are shown in Table 6.6. We see that for most of the programs, the speedup increases a noticeable amount as the communication bandwidth increases from 1 request per cycle to 2 requests per cycle. However, the speedup increase saturates beyond

Program	Communication Bandwidth			
	1	2	3	4
wc	3.56	3.95	4.00	4.00
cmp	5.66	6.29	6.41	6.34
052.alvinn	5.90	6.01	6.04	6.06
056.ear	7.43	7.77	7.83	7.85
125.m88ksim	3.07	3.30	3.34	3.34
126.gcc	2.17	2.17	2.17	2.17
129.compress	2.32	2.39	2.39	2.39
132.ijpeg	3.39	3.54	3.56	3.57

Table 6.6: Speedup of a 4-TPE, 4-issue superthreaded processor with different communication bandwidths

3 and 4 requests per cycle. We conclude that a communication bandwidth of 2 requests per cycle would be sufficient for a 4-TPE, 4-issue superthreaded processor.

6.7 Data Cache Bandwidth Requirement

As the total issue rate increases in the superthreaded architectural models, it requires higher data cache bandwidth to support multiple loads and stores per cycle. Techniques to improve cache bandwidth include non-blocking, multi-port, and interleaved multi-bank [37]. Non-blocking improves the cache performance by reducing the bandwidth degradation due to misses. Multi-port (duplicated ports) and multi-bank caches can serve multiple requests per cycle. In general, a multi-port cache is more effective than a multi-bank cache because it has less port contention. However, it is much more expensive than a multi-bank cache.

read ports	write ports	banks	wc			cmp			052.alvinn			056.ear		
			contention % r	contention % w	speed up	contention % r	contention % w	speed up	contention % r	contention % w	speed up	contention % r	contention % w	speed up
1	1	2	3.20	34.56	3.95	20.40	15.99	6.19	51.81	14.73	5.60	64.40	27.36	4.28
1	1	4	2.38	16.33	3.95	19.43	16.12	6.29	33.57	16.48	5.91	47.76	34.88	5.70
1	1	8	2.27	16.32	3.95	18.54	16.12	6.29	22.10	8.73	5.96	36.02	30.20	6.55
2	1	2	0.04	34.57	3.95	0.00	16.00	6.19	11.16	13.90	6.00	25.18	26.11	6.87
2	1	4	0.04	16.32	3.95	0.00	16.13	6.29	6.11	16.46	6.01	10.20	32.59	7.77
2	1	8	0.04	16.31	3.95	0.00	16.12	6.29	3.52	8.99	6.01	4.65	28.66	7.89
3	2	2	0.00	14.53	3.95	0.00	0.00	6.29	3.45	6.35	6.01	6.25	16.45	7.84
3	2	4	0.00	3.82	3.95	0.00	0.00	6.29	1.43	1.71	6.02	1.16	5.59	7.96
3	2	8	0.00	3.82	3.95	0.00	0.00	6.29	0.69	0.81	6.02	0.43	3.94	7.96
Cache hit ration %			99.95	99.98		99.72	99.98		92.25	99.88		99.99	99.98	

read ports	write ports	banks	124.m88ksim			126.gcc			129.compress			132.ijpeg		
			contention % r	contention % w	speed up	contention % r	contention % w	speed up	contention % r	contention % w	speed up	contention % r	contention % w	speed up
1	1	2	34.52	17.07	3.21	18.38	22.07	2.13	27.37	11.03	2.24	53.41	20.40	2.97
1	1	4	20.87	24.38	3.27	13.56	31.01	2.14	22.70	12.10	2.25	37.72	29.24	3.28
1	1	8	14.92	17.42	3.27	11.62	25.66	2.15	20.90	7.19	2.26	29.50	23.30	3.38
2	1	2	2.03	16.43	3.29	0.90	23.19	2.16	0.65	11.48	2.39	15.93	20.25	3.45
2	1	4	0.96	24.93	3.30	0.27	32.34	2.17	0.20	13.26	2.39	6.36	28.64	3.54
2	1	8	0.60	17.63	3.30	0.18	27.29	2.17	0.11	8.86	2.39	3.36	23.29	3.55
3	2	2	0.10	7.88	3.29	0.13	9.69	2.17	0.02	2.58	2.39	2.76	10.34	3.55
3	2	4	0.05	2.10	3.30	0.04	7.21	2.17	0.00	2.06	2.39	0.98	5.42	3.56
3	2	8	0.04	0.87	3.30	0.02	5.05	2.17	0.00	0.00	2.39	0.30	4.03	3.56
Cache hit ration %			99.61	98.24		99.41	99.37		99.11	98.52		99.81	99.50	

Table 6.7: Speedups and percentages of request blocked due to port contention

To determine the cache bandwidth requirement for a superthreaded processor, we run the benchmark programs through the simulator configured with different combinations of multi-port and multi-bank. Again, we use a 4-TPE, 4-issue superthreaded processor as our base model. For multi-port cache, we use three different configurations: one read port/one write port, two read ports/one write port, and three read ports/two write ports. For multi-bank cache, the number of interleaved banks ranges from 2 to 8. Table 6.7 shows the simulation results. It shows the percentage of requests blocked due to port contention, which happens when more requests than the available ports are send to the same bank in the same cycle. The results also show the cache hit ratio for reads and writes. Note that the cache hit ratios would not be affected by the number of ports or banks.

We can see that most of the request contention ratio decreases as the number of ports or banks increases. For some programs, however, the contention ratio for the single write port increases as the number of banks increase from 2 to 4. This is because the write-back (from the memory buffer to the data cache) bandwidth will be limited to 2 writes per cycle when the cache module has only two banks and one write port in each bank. With fewer write-back requests per cycle, the contention ratio for the write port could be reduced, but eventually the total execution time may increase due to the slower write-back.

It is also very interesting to see that when the contention ratio is less than 30%, further reducing the contention ratio by adding banks or ports would not improve the speedup much. This is because when the contention ratio is not very high, a blocked request has a good chance to be served in the next cycle. As a result, delaying a few cache accesses for one or two cycles would not degrade the performance much. On the other hand, for programs with high instruction throughput and with a contention ratio higher than 50% such as 056.ear, increasing the number of cache ports and banks can improve their speedup significantly. Overall, a 4-way interleaved cache module with two read ports and one write would provide sufficient bandwidth for the benchmark programs studied.

Chapter 7

Generating Superthreaded Codes with the SUIF Compiler

A complete implementation of the superthreading compiler is currently under development at University of Minnesota as part of the Agassiz compiler project. To test the ability of an existing parallelizing compiler for generating superthreaded code and to study new compiler techniques for the complete implementation of the superthreading compiler, we have built a superthreading compiler prototype based on the SUIF parallelizing compiler (baseparsuif-1.0.0.beta.1) [51]. In this chapter, we describe the framework of the compiler prototype and show the speedups of the compiler generated superthreaded codes running on the superthreaded processor simulator.

7.1 Compiler Framework

We built the superthreading compiler prototype by modifying the existing passes of the SUIF parallelizing compiler and adding our own passes to generate superthreaded code. Figure 7.1 shows the compilation passes we use in the superthreading compiler prototype. Passes we add to generate superthreaded code are shown in shaded boxes. All the other passes are from the SUIF parallelizing compiler.

Although the SUIF compiler system can handle both C and Fortran programs, the released SUIF parallelizing compiler can only parallelize Fortran program effectively. Due to this limitation, the superthreading compiler prototype only targets for Fortran programs. The complete implementation of the superthreading compiler in the Agassiz compiler system will provide more program analysis and transformation support for C programs and generate superthreaded codes for both C and Fortran programs.

In the superthreading compiler prototype, a fortran program is first converted into C code by using a modified version of f2c (call sf2c in SUIF). The C code is then converted into SUIF intermediate representation by the C-parser. All of the remaining passes then work on the intermediate representation to perform program analysis or transformation. Before running the parallelization passes, the compiler first performs some classic optimizations such as constant folding, constant propagation, copy propagation, common sub-expression elimination, and dead code elimination. Next the compiler calls the induction variable substitution pass to detect induction variables and replace them by expressions of the loop index variable.

The compiler then performs scalar privatization analysis and reduction analysis to identify loop private variables and reduction variables. The analyzed results are kept in annotations to be used by later passes for parallelism analysis and thread pipelining. Next the compiler run the parallelism analysis pass to analyze and optimize loop-level parallelism for each loop. This pass first builds a data dependence graph for each loop nest, then analyzes the loop-level parallelism according to the data dependence graph and the private variables and reduction variables information produced by previous passes. This pass may also performs loop interchange, loop skewing, and loop reversal to

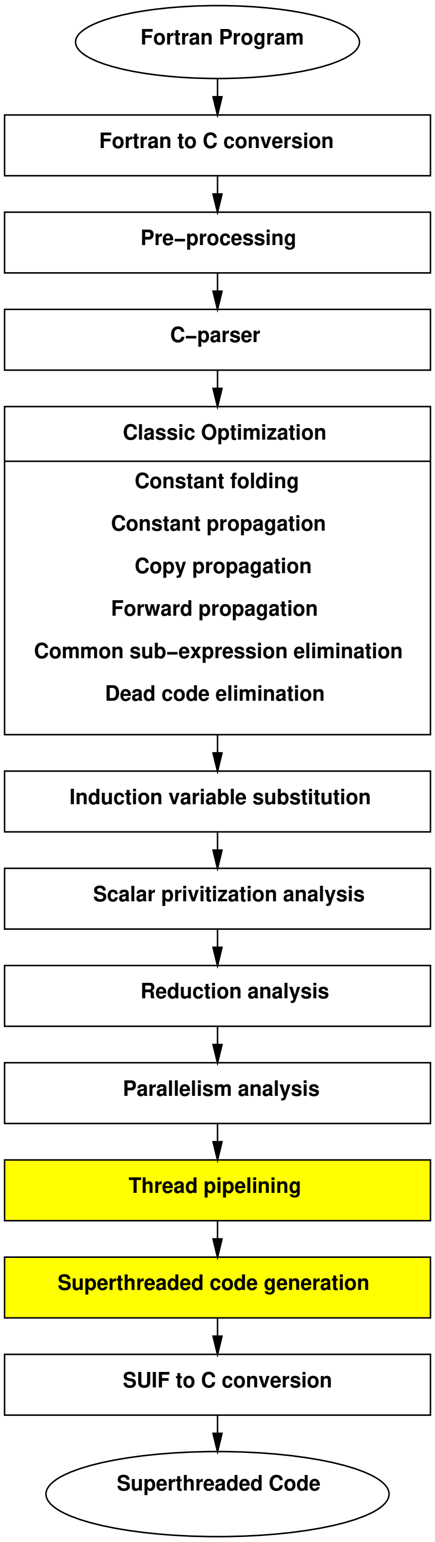

Figure 7.1: Compilation passes in SUIF for generating superthreaded code

improve loop-level parallelism. After the loop-level parallelism analysis and transformation, this passes will put a *doall* annotation to a loop if its loop iterations can execute in parallel, and put a *doacross* annotation to a loop if it only has loop-carried data dependences with constant distances.

After the parallelism analysis pass is done, the compiler performs thread pipelining for loops annotated as *doall* or *doacross* using the algorithm described in 5.3. Since the superthreaded architecture supports only one-level of parallel thread execution, once a loop is thread pipelined, we will not further perform thread pipelining for its inner loops even if they are parallel loops. Following the thread pipelining passes is the code generation passes. In this pass, we replace special superthreading operations, such as *fork* and *store_ts*, by function calls to specific subroutines for the superthreaded processor simulator as described in 6.1.

Finally, the compiler converts the transformed SUIF intermediate representation back to C code. The C code is then fed into our superthreaded processor simulation environment as described in 6.1 for performance evaluation.

7.2 Results

We use the superthreading compiler prototype to compile 8 SPEC95 floating pointer programs (*101.tomcatv*, *102.swim*, *103.su2cor*, *104.hydro2d*, *107.mgrid*, *125.turb3d*, *141.apsi*, and *146.wave5*)[1] and generate their superthreaded codes. We run the compiler generated superthreaded code on the superthreaded processor simulator with the same procedure and configuration as described in Chapter 6. We use the *test* input sets provided by SPEC95 for the simulation. However, some input sets are modified to further reduce the simulation time.

[1]The other two SPEC95 floating pointer programs (*110.applu* and *145.fpppp*) fail in the sf2c pass.

The simulation results for those benchmark programs are shown in Figure 7.2. Again, we run the benchmark programs on the simulator with the number of thread processing elements ranging from 1 to 8, and the issue rate within each thread processing element ranging from 1 to 8. The speedup numbers are obtained by comparing to the single-threaded, single-issue model. Note that we run the original benchmark programs on single-threaded models, and run the compiler generated superthreaded code on on multiple-threaded models.

The results show that the superthreading compiler prototype, which leverages existing parallelizing compiler techniques and use two simple passes for thread pipelining and code generation, can effectively parallelize the floating pointer benchmark programs for the superthreaded architecture. For *102.swim* and *107.mgrid*, the superthreaded execution model can achieves an speedup (entire program) of 4.4 and 6.3 with 8 TPEs. For *103.su2cor* and *104.hydro2d* the speedup from superthreading is about 3 for models with 8 TPEs. For *141.apsi* and *146.wave5*, the speedup is about 2 for models with 8 TPEs. The speedup of *101.tomcatv* and *125.turb3d* from superthreading is not significant. It is because only one third of *101.tomcatv* and half of *125.turb3d* dynamic executed codes are parallelized by the compiler.

In *101.tomcatv*, the core computation of the program is successfully parallelized by the compiler. However, becuase we use the *test* input set to reduce the simulation time, only one third of dynamic instructions are spent in the core computation and the remaining dynamic instructions are spent in input and output which cannot be executed in parallel. If we use the regular *ref* input set, the core computation will accumulate more then 98% of the sequential execution time and the overall speedup would be very good. To verify this, we compute the speedup of the core computation of *101.tomcatv* by excluding the input/output portion. The result is shown in Figure 7.3 and we can see the speedup of the core computation with 8 TPEs is about 6.

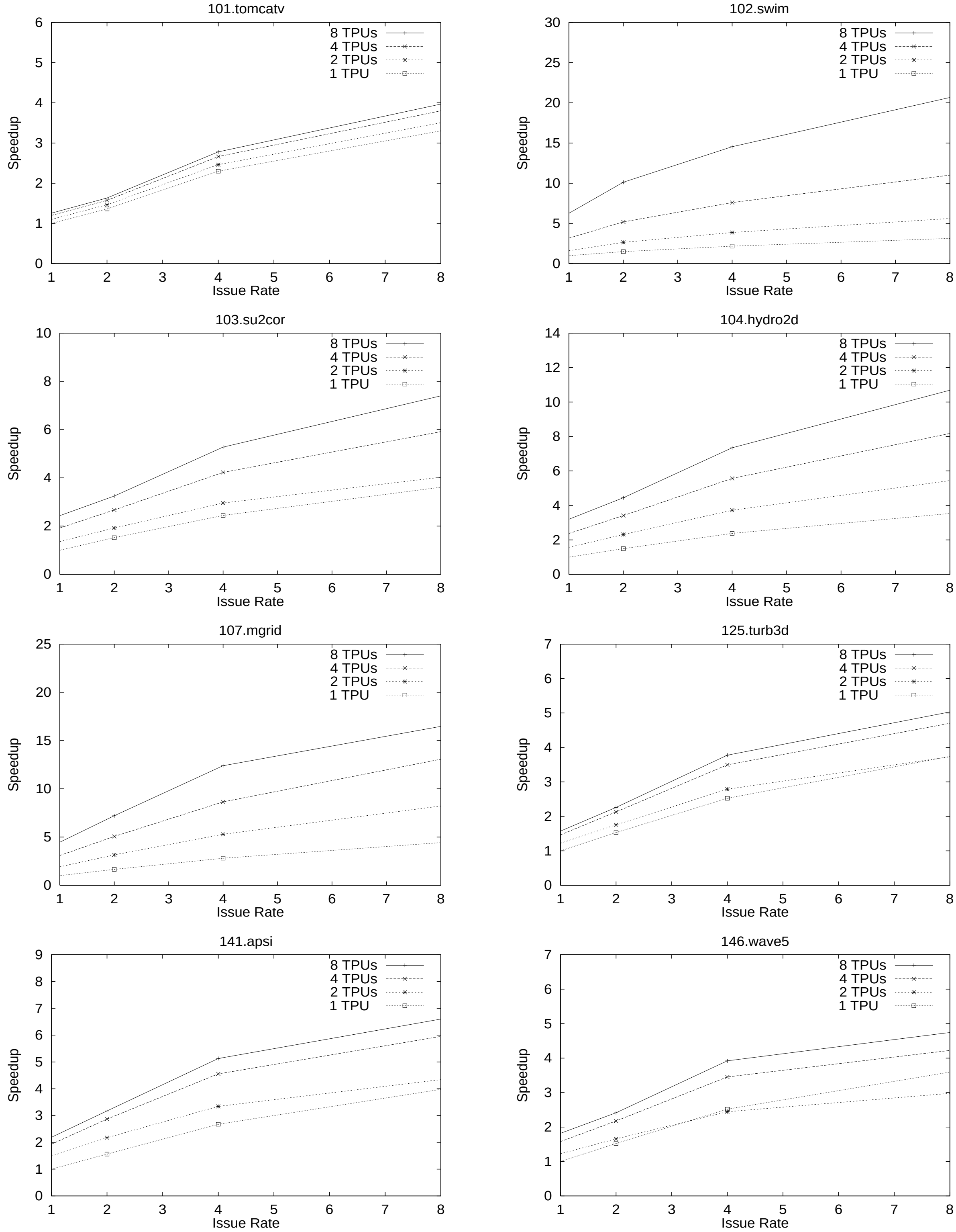

Figure 7.2: Speedups of compiler generated superthreaded codes for SPEC95 floating pointer programs

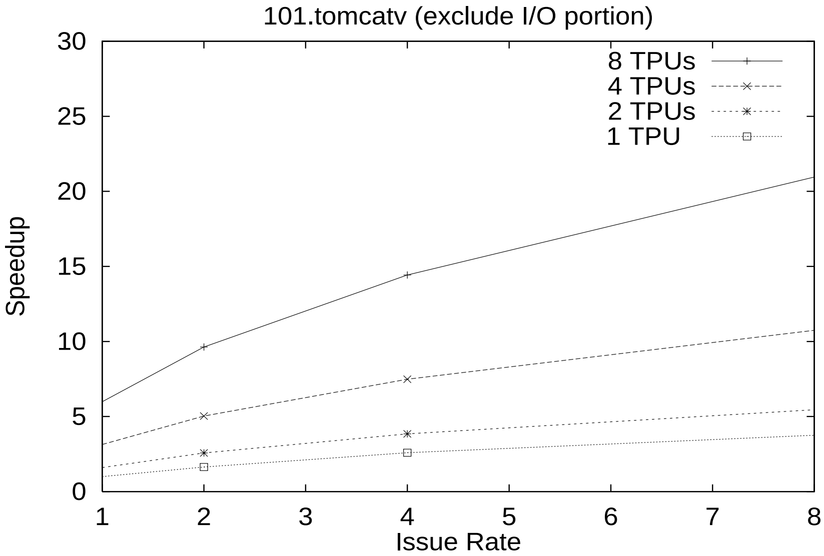

Figure 7.3: Speedup of 101.tomcatv excluding the input/output portion

Chapter 8

Two-Dimensional Superthreading

8.1 Introduction

As the speed gap between processors and memories exacerbated each year by the large discrepancy in their rate of increase, dealing with ever worsening memory latency has become a dominant issue in achieving high overall system performance. *Program locality*, including temporal and spatial locality, has been exploited successfully in sophisticated memory hierarchy, including multi-level high-speed cache memories. However, cache memories have many well-known problems, and the speed gap has grown to such an extent that techniques beyond exploiting program locality are needed. Recently, *program parallelism*, in addition to the program locality, has been increasingly used to combat this problem. Techniques, such as data prefetching and multi-threading, use program parallelism to overlap memory operations, and thus "hide" the memory latency. Key to the success of this approach is to find sufficient program parallelism to hide the long memory latency.

The SIA road map for semiconductors proposes a schedule that will allow the fabrication of VLSI chips containing a billion of transistors in the next few years. Such technologies will allow a tremendous amount of functionality and memory capacity to be included on a single chip for very high performance. However, it has been known that using such technologies to increase the capacity and the level of hierarchy in cache memories will meet diminishing return. Exploiting program parallelism, using fast context switching among multiple light threads or data prefetching, has shown increasing effectiveness in hiding memory latency, in addition to the cache memories. However, as the memory latency increases, more parallelism needs to be found in order to be expended for this purpose.

There are several major obstacles which limit the amount of parallelism that can be exploited. Most state-of-the-art microprocessors use single-threaded superscalar architecture to exploit instruction-level parallelism. Its parallelism is limited by the size of the single instruction window. Multithreaded architectures, such as Tera[2] and M-Machine [10], rely totally on parallelizing compilers to partition a program into multiple threads for context switching. Program structures such as pointer aliases, loops with early exits, or complicated dependences across loop iterations cannot be analyzed at compile time, which often resulting in a huge loss in "exploitable" program parallelism.

In this chapter, we describe a two-dimensional superthreaded architecture, which combines aggressive thread-level and instruction-level speculation on control dependences, runtime data dependence checking both within and between threads, compiler-controlled data speculation, and a thread pipelining execution model to exploit more program parallelism than traditional approaches both at compile time and runtime. It can then expend such program parallelism for memory latency hiding. Such an architecture allows more parallelism to be exploited from complicated program structures such as doubly-nested WHILE loops, loops with multiple control blocks in the loop body, loops with ambiguous data dependences between loop iterations, etc. It also allows system resources to be shared more efficiently among threads.

In this chapter, we describe its program execution model in section 2. The applications of such an execution model on real programs are presented in section 3. In section 4, we describe some possible hardware implementations of such an execution model. Some related work is described in section 5. We summarize this chapter in section 6.

8.2 The Two-Dimensional Superthreaded Execution Model

8.2.1 Processor Overview

As shown in Figure 8.1, a two-dimensional superthreaded processor consists of a number of thread processing elements with a shared second-level instruction cache and a shared data cache. Each thread processing element contains a first-level instruction cache, an execution unit, a communication unit, a primary memory buffer, and a secondary memory buffer. Logically, a thread processing element consists of multiple thread slots, each thread slot corresponding to a thread context. The communication units of all the thread processing elements are linked together via a uni-direction link.

At run-time, the multiple thread processing elements can fetch and execute instructions from multiple program locations simultaneously, and communicate with each other via the uni-direction communication link. On the other hand, within each thread processing element, threads in different thread slots can communicate via the primary memory buffers.

The primary memory buffer performs run-time dependence checking for data synchronization and/or speculation. The primary memory buffer is also used to buffer uncommitted store data written by speculative threads.

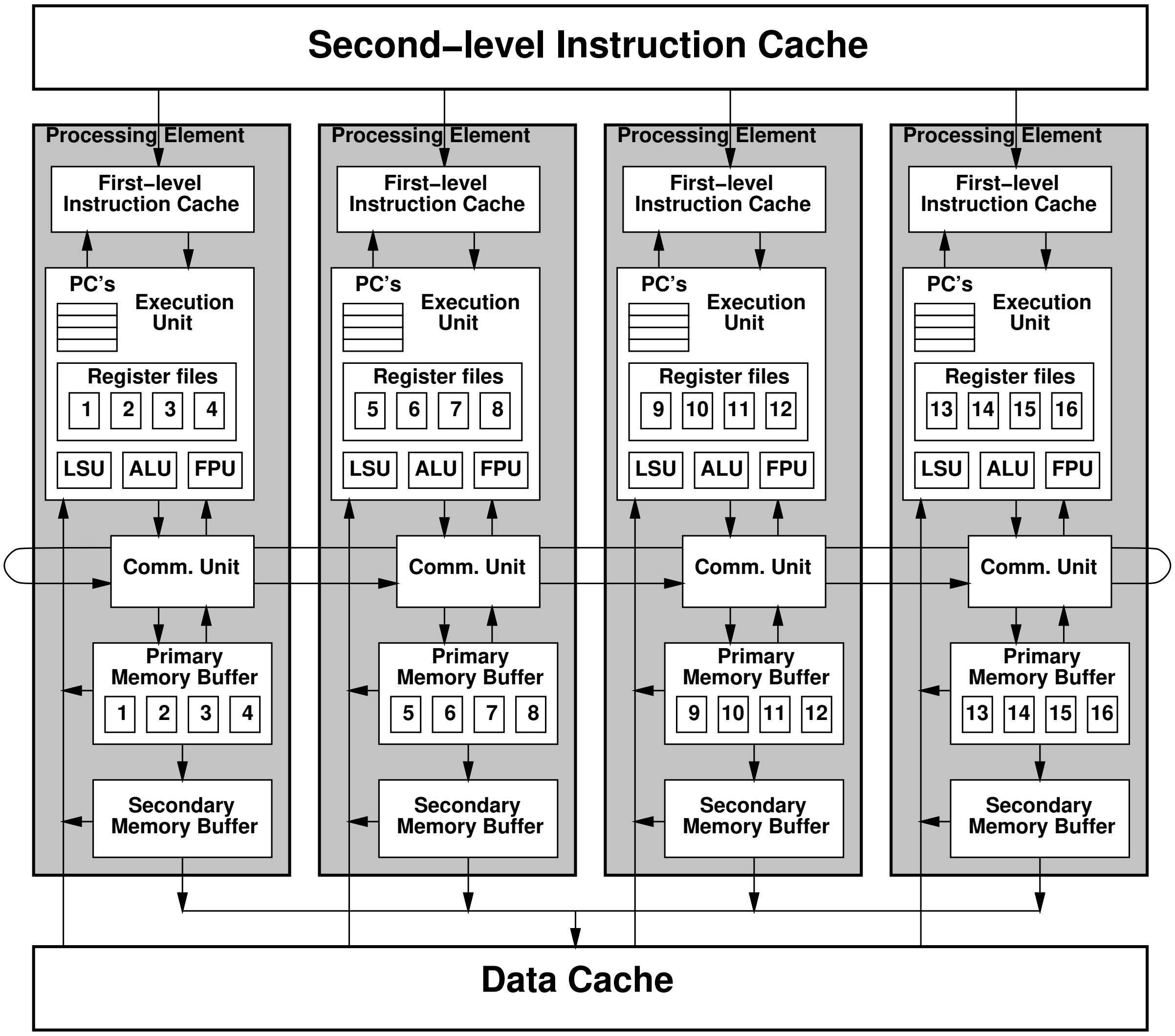

Figure 8.1: The organization of the two-dimensional superthreaded processor

8.2.2 Two-Dimensional Thread Pipelining Execution Model

The superthreaded architecture supports a thread pipelined execution model to increase the overlap among concurrent threads and to facilitate data dependence enforcement between threads.

The thread pipeline execution model uses *run-time data synchronization* and *compiler-directed data speculation* to enforce read-after-write data dependence. Run-time data synchronization can identify an address which will be written by a thread. All reads to that address in its succeeding concurrent threads will wait until the data is written to that address. On the other hand, compiler-directed data speculation identifies an address whose content may be read by a thread before its content is updated by an earlier thread. If later, an earlier thread forwards a store data to the address, the data dependence is violated and the thread and all its successor threads will be aborted (see Section 8.2.3.3). Furthermore, the thread pipeline execution model maintains an in-order write-back commitment among the concurrent threads to enforce write-after-write data dependence.

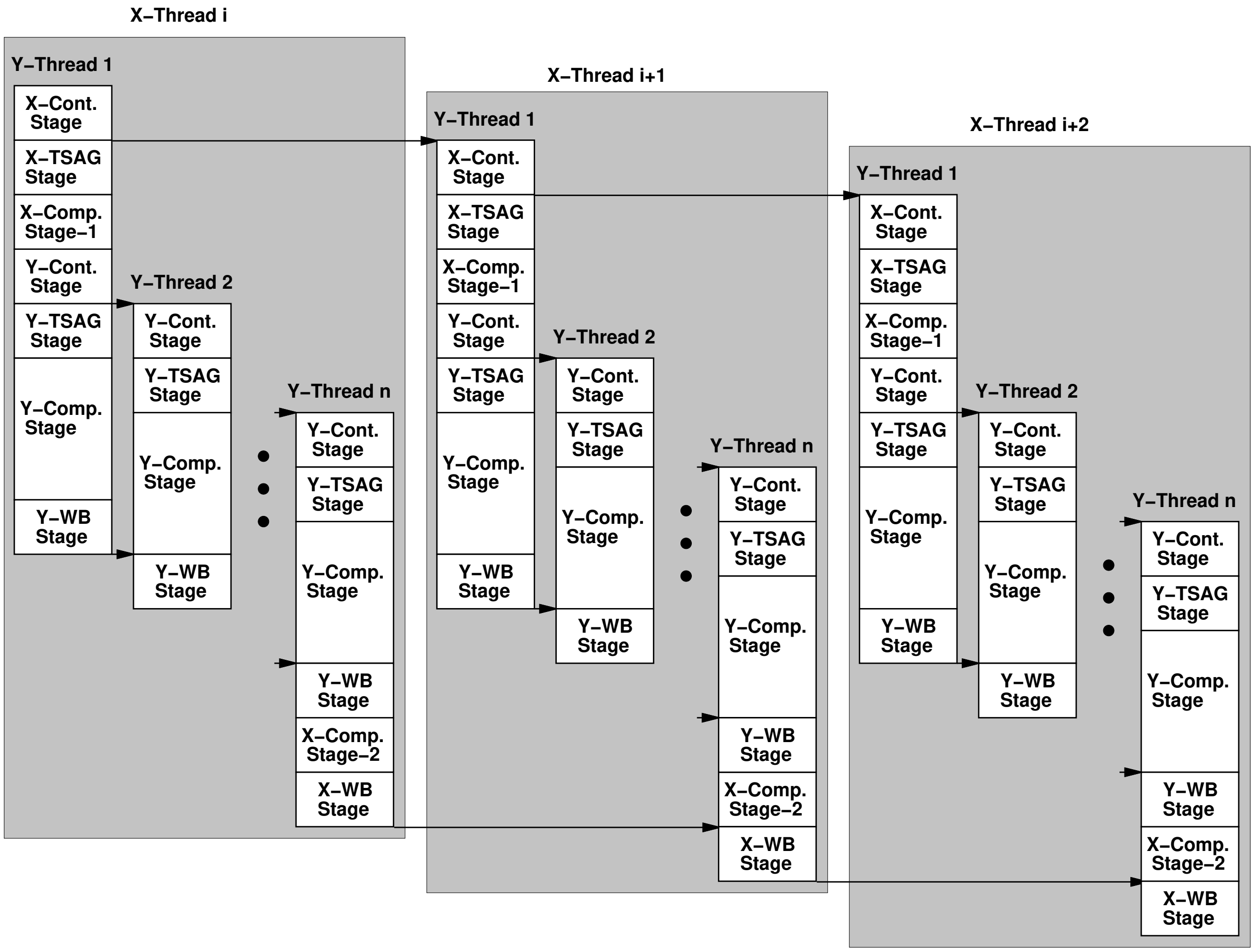

Figure 8.2: The pipelined execution of x-threads and y-threads

Threads can be scheduled along two possible dimensions, one along the thread processing elements and the other within each thread processing element. We will identify them as x-dimension and y-dimension, respectively. The threads among the thread processing element (i.e. along the x-dimension) will be called x-threads. The threads within each thread processing element (i.e. along the y-dimension) will be called y-threads (see Figure 8.2).

The superthreaded architecture performs thread initiation and data forwarding through explicit thread management and communication instructions. The execution of a thread is partitioned into several stages. Figure 8.2 shows the thread pipeline stages and the relation among concurrent threads.

An x-thread is initiated by its preceding x-thread, and starts with the *continuation* stage. The continuation stage computes the recurrent variables, such as outer loop index variables, needed to initiate the next x-thread. The values of these variables will be forwarded to the next thread processing element before the next x-thread is activated. The continuation stage ends with an *x_fork* instruction, which starts the next x-thread.

The x-thread enters the *target store address generation (X_TSAG)* stage after the continuation stage. The TSAG stage calculates the addresses for the *x-target stores*, which perform store operations that later concurrent x-threads could data depend on. The thread will send a *x_tsag_done* flag and forward all the target store addresses to its successor x-threads when it completes its X_TSAG stage.

Next to the X_TSAG stage is the *x-computation* stage. It performs the computation before entering the y-thread codes, e.g. performing some computation in the outer loop nest before entering the inner loop nest in a doubly-nested loop.

The *y-continuation* stage after the *x-computation* stage computes the y-recurrent variables, such as inner loop index variables, needed to fork a y-thread within the same thread processing element. The values of these variables will be written in the corresponding primary memory buffer entries for that y-thread. The y-continuation stage ends with a *y_fork* instruction to fork a new y-thread in the same thread processing element.

After initiates a new y-thread in the next available thread slot of the same thread processing element, the current y-thread enters the *y-target store address generation(Y_TSAG)* stage, and calculates the addresses for the *y-target stores*, which perform store operations that later y-threads could data depend on. When the thread completes its Y_TSAG stage, it will send a *y_tsag_done* flag to its successive y-thread.

The *y-computation* stage after the Y_TSAG stage performs the main computation of a y-thread. During the computation stage, if the address of a load operation matches that of a target store entry in its primary memory buffer, the thread will either read the data from the entry if it is available, or wait until it is available. A data becomes available either after it is forwarded from earlier x-threads or after it is written by earlier y-threads in the same thread processing element. On the other hand, if the value of an x-target store is computed during this stage, the thread must forward the address and the data to the primary memory buffer of all its concurrent successor x-threads; if the value of a y-target store is computed during this stage, the thread writes the data to the corresponding primary memory buffer slot in the thread processing element.

The y-computation stage of a thread, if not aborted by a preceding thread, will end with a *y_stop* command. After the *y_stop* command, the y-thread will enter the *y-write back* stage after receiving the *y_wb_done* flag from its predecessor y-thread.

Because the data from the store operations of a y-thread is actually kept in the primary memory buffer before the y-write back stage to ensure an in-order commitment to the secondary memory buffer, the task of the y-write back stage is to move the data from the primary memory buffer entries to the secondary memory buffer, release the occupied thread slot, and send a *y_wb_done* flag to its successive y-thread in the same thread processing element.

A y-thread, except the last one in an x-thread, ends with its y-write-back stage. But for the last y-thread within an x-thread, there could be some computation needed between the y-write-back and the x-write-back stage, for example, when some more computation is needed at the end of the outer loop after the inner loop is completed. When the thread executes the *x_stop* command, it waits for the *x_wb_done* flag from its preceding x-thread and enters *x-write-back* stage. The x-thread will write its data in the secondary memory buffer to the data cache, send an *x_wb_done* flag to its successor x-thread, and release the thread processing element.

8.2.3 Instruction Set Extension

To facilitate thread pipelining, run-time data dependence checking, and compiler-directed data speculation, a set of thread management and communication instructions are provided as described below.

8.2.3.1 Instructions for Thread Management

The *x_fork* *<starting_address>* and *y_fork* *<starting_address>* instructions will start a new thread context on the next thread slot, or the next thread processing element, respectively. For *x_fork*, part of the memory buffer will be copied to the next thread processing element. The part copied includes the x-continuation variables, and the x-target store entries from the current x-thread and all its predecessor x-threads that have not yet retired.

When a thread determines that a control speculation is incorrect, it must kill all of its successor threads. The *x_abort_future* and *y_abort_future* instructions are used to abort the successor x-threads and y-threads, respectively.

The *x_stop* or *y_stop* instructions are used by an x-thread or a y-thread to terminate itself normally. It waits for the *x_wb_done* or the *y_wb_done* flag from its preceding thread and then performs the *x-write-back* or *y-write-back* stage automatically. After completing the write-back stage and issuing the *wb_done* flag to the next thread, the thread slot or the processing element retires and becomes *idle*.

8.2.3.2 Instructions for Run-Time Data Dependence Checking

The *x_allocate_ts* <*address*> instruction allocates a target store entry in the primary memory buffer, and forwards the address to the next thread processing element. The *y_allocate_ts* <*address*> instruction allocates a target store entry in the primary memory buffer and sets a flag indicating that the entry will be written, but does not forward the address.

The *x_store_ts* <*address, data*> instruction stores the data in the primary memory buffer entry corresponding to the target store address, and forwards the address and the data to its successor x-threads. If a corresponding target store entry cannot be found in the primary memory buffer, a new target store entry is allocated. This instruction is used to perform the last write to an x-target store entry.

The *y_store_ts* <*address, data*> instruction stores the data in the primary memory buffer slot corresponding to the target store address and sets a flag indicating that the data is available for successor y-threads.

The *x_release_ts* <*address*> and *y_release_ts* <*address*> instructions release the specified target store entry to the next x-thread or y-thread. These instructions are used to release a target store entry that will not be written after this point because the control path which executes the corresponding *x_store_ts* or *y_store_ts* instruction is not taken.

The *x_wait_tsag_done* or *y_wait_tsag_done* instruction requires a thread to wait for the *x_tsag_done* or *y_tsag_done* flag of its preceding thread. These instructions prevent a thread from performing any unsafe loads that may depend on the target stores of its preceding threads. On the other hand, the *x_release_tsag_done* or *y_release_tsag_done* instruction forwards *x_tsag_done* or *y_tsag_done* flag to the successor x-thread or y-thread. These instructions are placed after the TSAG stage of a thread.

8.2.3.3 Instruction For Compiler-Directed Data Speculation

The *load_tl* <*address*> instruction allows the compiler to perform a speculative load which may data depend on the target stores of its predecessor threads. This instruction is used when the compiler cannot determine if a load executed by a thread is dependent on a store of an earlier thread, and it requires a lot of overhead to generate the target store address for the store in the TSAG stage. To speculate on the potential data dependence, the compiler uses the *load_tl* instruction to load the data speculatively in a later thread instead of generating the target store address in the TSAG stage of an earlier thread. The data dependence will be checked later when the earlier thread performs the store with a target store.

When a thread executes a *load_tl* instruction, it will first check the primary memory buffer to see if there is a target store entry for the *address*. If YES, the thread will wait for the latest thread that allocated the target store entry to forward the data. Otherwise, the thread will read the data from the data cache and keep the load address in an primary memory buffer entry.

It is possible that the *load_tl* instruction reads a data from an address before one of its predecessor thread performs a target store to the address. In this case, the store address generated by the predecessor threads will match the load address in the primary memory buffer and a data dependence violation is detected. When a data dependence violation is detected, the thread which executed the *load_tl* and its successor threads will be squashed and re-executed.

8.2.3.4 An Example Program

The example in Figure 8.3 shows a doubly-nested Do-while loop from SPICE in the SPEC FP92 suite, and its corresponding superthreaded code. Because each inner iteration writes or reads some elements of the array VALUE, there are potential read-after-write data dependences among both the inner loop iterations and the outer loop iterations. The superthreaded code uses the run-time data synchronization via *x_allocate_ts* and *y_allocate_ts* to enforce the potential data dependence at run-time. This allows thread-level speculation, and thus parallel execution of the doubly-nested Do-while loop. In this example, each thread is one iteration of the loop nest. An x-thread corresponds to an outer iteration, while a y-thread corresponds to an inner iteration. The *x_allocate_ts* instruction allocates a target store entry in the primary memory buffer to facilitate the run-time dependence checking among outer iterations. Similarly, the *y_allocate_ts* instruction enables run-time data dependence checking among inner iterations.

```
C
C  FORWARD SUBSTITUTION
C
      DO 20 I=2,NSTOP
      LOC=I
      IORD=NODPLC(IRSWPF+I)
   10 LOC=NODPLC(JCPT+LOC)
      IF (NODPLC(JCOLNO+LOC).GE.I) GO TO 20
      J=NODPLC(JCOLNO+LOC)
      JORD=NODPLC(IRSWPF+J)
      VALUE(LVN+IORD)=VALUE(LVN+IORD)-
     1     VALUE(LVN+LOC)*VALUE(LVN+JORD)
      GO TO 10
   20 CONTINUE
```

```
      I1=2
   5  IF (I1.GT.NSTOP) GO TO 30
C  Fork the next outer-loop iteration on
C  the next thread processing element
C  x-thread: continuation stage
      I=I1
      x_store_ts(I1, I1+1)
      x_fork 5
C  x-thread: target store stage
      IORD=NODPLC(IRSWPF+I)
      x_allocate_ts(VALUE(LVN+IORD))
      x_release_tsag_done
C  x-thread: computation stage
      x_wait_tsag_done
      LOC = I
C  Fork the next inner-loop iteration on
C  the current thread processing element
C  y-thread: continue stage
   10 y_store_ts(LOC,NODPLC(JCPT+LOC))
      y_fork 10
C  y-thread: target store stage
      y_allocate_ts(VALUE(LVN+IORD))
      y_release_tsag_done
C  y-thread : computation stage
      y_wait_tsag_done
      IF (NODPLC(JCOLNO+LOC).GE.I) GO TO 20
      J=NODPLC(JCOLNO+LOC)
      JORD=NODPLC(IRSWPF+J)
      V1=VALUE(LVN+IORD)-
     1   VALUE(LVN+LOC)*VALUE(LVN+JORD)
      y_store_ts(VALUE(LVN+IORD),V1)
      y_stop
C  y-thread : write back stage
   20 y_abort_future
      x_store_ts(VALUE(LVN+IORD),V1)
      x_stop
C  x-thread : write back stage
   30
```

Figure 8.3: An example program from SPICE and its superthreaded code

8.3 Applications of Two-Dimensional Superthreading

With the hardware support for speculative execution and run-time data dependence checking, the two-dimensional superthreaded architecture can exploit thread-level parallelism from a broad range of program structures.

Doubly-nested WHILE loops: In a WHILE loop, the execution of a later iteration is control dependent on the exit condition determined by an earlier iteration. Due to the control dependences between loop iterations, most existing multiprocessor and multithreaded architectures cannot execute a WHILE loop in parallel. With the hardware support for thread-level speculation, the two-dimensional superthreaded architecture can execute a WHILE loop in its x-dimension or y-dimension, or execute a doubly-nested WHILE loop in both of its x-dimension and y-dimension. Figure 8.4 shows an example of a doubly-nested WHILE loop from the TRACK benchmark in the Perfect Club suite and its corresponding superthreading version. The loop, which accounts for about 30% of the total sequential execution time, is qualified as doubly-nested WHILE loop because there are a couple of exit conditions in its loop body. By storing the speculative execution results in the memory buffers, we can schedule the outer KD loop iterations as x-threads on different thread processing elements, and the inner KT loop as y-threads within the same thread processing elements. If a thread encounters any one of the exit conditions, it will abort its successor y-threads and x-thread threads. Note that the two statements in label 250 are rarely executed but could cause data dependences across loop iterations. We enforce the data dependences through control speculation with an assumption that those statements would rarely be executed at run time.

Doubly-nested loops with ambiguous data dependences: A loop with ambiguous data dependences between its iterations that cannot be analyzed at compiler time is very difficult to

```
      DO 400 KD = 1, LSTDAT
      ...
      IFDATA = LSTTRK+1
      DO 300 KT = 1,LSTOLD
C
C  Crude association tests
C
      ...
      IF (IDDSIT(KT).NE.ISITE) GO TO 300
      IF (Y0.GT.DELX.OR.Y0.LT.DELXN) GO TO 300
      IF (Y1.GT.DELX.OR.Y1.LT.DELXN) GO TO 300
      IF (ZDAT.LT.ZZMIN) GO TO 300
C
C  Nominal gating test
C
      ...
      GATE = PMKAL(1,IG) + SIGMS2
      IF (GATE.LE.0.0) THEN
         ISTAT = -7
         RETURN
      ENDIF
      ...
      IF ( G2RAT.GT.GATEF ) GO TO 300
C
C  Velocity gate test
C
      ...
      IF (G2.GT.VMAG) GO TO 300
C
C  Create provisional track
C
      ...
      NEXTRK = LSTTRK+1
C
C  TEST: storage overflow
C
      IF (NEXTRK.LE.MAXTRK) THEN
         ISTAT = -6
         RETURN
      ENDIF
      ...
      IF (IOK.GE.0) THEN
         ISTAT=-9
         RETURN
      ENDIF
      ...
C
C  Begin uniqueness tests
C
      ...
      IF (ISSAME.EQ.0) GO TO 250
C
C  Choose equivalent track
C
      ...
      GO TO 300
C
C  Extend the track file
C
  250 NUSED(KD) = NUSED(KD)+1
      LSTTRK = NEXTRK
  300 CONTINUE
  400 CONTINUE
```

```
      KD1=1
100   IF (KD1.GT.LSTDAT) GO TO 410
C  Fork the next outer-loop iteration on
C  the next thread processing unit
      KD=KD1
      x_store_ts(KD1,KD1+1)
      x_fork 100
      ...=LSTTRK+1
      ...
      KT1=1
110   IF (KT1.GT.LSTOLD) GO TO 400
C  Fork the next inner-loop iteration on
C  the current thread processing unit
      KT=KT1
      y_store_ts(KT1,KT1+1)
      y_fork 110
C  Crude association tests
      ...
      IF (...) GO TO 300
C  Nominal gating test
      ...
      IF(GATE.LE.0.0) THEN
        y_abort_future
        x_abort_future
        ISTAT=-7
        RETURN
      ENDIF
      ...
      IF ( G2RAT.GT.GATEF ) GO TO 300
C  Velocity gate test
      ...
      IF ( G2.GT.VMAG ) GO TO 300
C  Create provisional track
      ...
      NEXTRK=LSTTRK+1
C  TEST: storage overflow
      IF ( NEXTRK.GT.MAXTRK ) THEN
        y_abort_future
        x_abort_future
        ISTAT=-6
        RETURN
      ENDIF
      ...
      IF ( IOK.GE.0 ) THEN
        y_abort_future
        x_abort_future
        ISTAT=-9
        RETURN
      ENDIF
C  Begin uniqueness tests
      ...
      IF ( ISSAME.EQ.0 ) GO TO 250
C  Choose equivalent track
      ...
      GO TO 300
C  Extend the track file -- Abort future threads
C  because of unexpected data dependences
250   y_abort_future
      x_abort_future
      y_store_ts(NUSED(KD),NUSED(KD)+1)
      x_store_ts(NUSED(KD),NUSED(KD)+1)
      y_store_ts(LSTTRK,NEXTRK)
      x_store_ts(LSTTRK,NEXTRK)
      x_fork 100
      GO TO 110
300   y_stop
400   x_stop
410   ...
```

Figure 8.4: An example program from TRACK and its superthreaded code

be parallelized. The two-dimensional superthreaded architecture provides the compiler with two kinds of run-time data dependence checking support: run-time data synchronization and compiler-directed data speculation. In the doubly-nested loop shown in Figure 8.3, it is very difficult for the compiler to analyze the data dependences in the subscripted-subscripted array VALUE among outer-loop iterations. In the superthreading version of the loop shown in Figure 8.3, we use the run-time data synchronization mechanism to enforce the potential data dependences. We can also enforce the potential data dependences among outer-loop iterations by using the compiler-directed data speculation mechanism. The superthreaded code for the loop using compiler-directed data speculation is shown in Figure 8.5, where we use *load_tl* instructions to read the VALUE array and eliminate the *x_allocate_ts* instruction in the X_TSAG stage.

Singly-nested loops with multiple conditional blocks in loop bodies: The control speculation support of the two-dimensional superthreaded architecture also allows the compiler to speculate on a sequence of conditional instruction blocks and execute them in parallel. The code segment shown in Figure 8.6 is one of the most time-consuming loops in 034.mdljdp2 from the SPEC95 benchmark. The loop body consists of multiple conditional instruction blocks whose execution are control dependent on some conditional branches. In this loop, we can execute each loop iteration as an x-thread on a thread processing element and partition the multiple conditional instruction blocks into y-threads with control speculation. The corresponding superthreading code is shown on the right of Figure 8.6. This program execution model allows us to exploit both inter-iteration thread-level parallelism and intra-iteration thread-level parallelism from a single loop.

Doubly-nested DO loops with/without data dependences: A DO loop has a loop index variable and explicit loop bounds, which determine the number of the loop iterations. This is

```
      I1=2
   5  IF (I1.GT.NSTOP) GO TO 30
C  Fork the next outer-loop iteration on
C  the next thread processing element
C  x-thread: continuation stage
      I=I1
      x_store_ts(I1, I1+1)
      x_fork 5
C  x-thread: target store stage
      x_release_tsag_done
C  x-thread: computation stage
      x_wait_tsag_done
      IORD=NODPLC(IRSWPF+I)
      LOC = I
C  Fork the next inner-loop iteration on
C  the current thread processing element
C  y-thread: continue stage
   10 y_store_ts(LOC,NODPLC(JCPT+LOC))
      y_fork 10
C  y-thread: target store stage
      y_allocate_ts(VALUE(LVN+IORD))
      y_release_tsag_done
C  y-thread : computation stage
      y_wait_tsag_done
      IF (NODPLC(JCOLNO+LOC).GE.I) GO TO 20
      J=NODPLC(JCOLNO+LOC)
      JORD=NODPLC(IRSWPF+J)
      V1=load_tl(VALUE(LVN+IORD))-
     1   load_tl(VALUE(LVN+LOC))*load_tl(VALUE(LVN+JORD))
      y_store_ts(VALUE(LVN+IORD),V1)
      y_stop
C  y-thread : write back stage
   20 y_abort_future
C  Forward VALUE(LVN+IORD) to successor x-threads for
C  data dependence violation checking
      x_store_ts(VALUE(LVN+IORD),V1)
      x_stop
C  x-thread : write back stage
   30
```

Figure 8.5: A superthreaded code for the example code from SPICE using compiler-directed data speculation

```
      DO 20 J = JBEG, JEND
C
C  Particle-particle centre separation.
C
      XDIFF    = X0(I) - X0(J)
      IF (XDIFF.LT.-SP1) THEN
         XDIFF = XDIFF + SP2
      ELSE IF (XDIFF.GT.SP1) THEN
         XDIFF = XDIFF - SP2
      END IF
      RSQ = XDIFF*XDIFF
      IF (RSQ.GT.RCUTSQ) GO TO 20

      YDIFF    = Y0(I) - Y0(J)
      IF (YDIFF.LT.-SP1) THEN
         YDIFF = YDIFF + SP2
      ELSE IF (YDIFF.GT.SP1) THEN
         YDIFF = YDIFF - SP2
      END IF
      RSQ = RSQ + YDIFF*YDIFF
      IF (RSQ.GT.RCUTSQ) GO TO 20

      ZDIFF    = Z0(I) - Z0(J)
      IF (ZDIFF.LT.-SP1) THEN
         ZDIFF = ZDIFF + SP2
      ELSE IF (ZDIFF.GT.SP1) THEN
         ZDIFF = ZDIFF - SP2
      END IF
      RSQ = RSQ + ZDIFF*ZDIFF
      IF (RSQ.GT.RCUTSQ) GO TO 20
C
C  Update number of pairs within cutoff.
C

      NUMPRS = NUMPRS + 1
C
C  Calculate LJ part of interaction.
C
      RECIR2 = SP1/RSQ
      RPOW06 = SIGMA6*RECIR2*RECIR2*RECIR2
      RPOW12 = RPOW06*RPOW06
      UCONF  = UCONF+(RPOW12-RPOW06-ULJCUT)
      FOVERR = RECIR2*(RPOW12+RPOW12-RPOW06)
C
C  Update force accumulators.
C
      FIJX  = XDIFF*FOVERR
      FIJY  = YDIFF*FOVERR
      FIJZ  = ZDIFF*FOVERR

      XFORCE(I) = XFORCE(I) + FIJX
      YFORCE(I) = YFORCE(I) + FIJY
      ZFORCE(I) = ZFORCE(I) + FIJZ

      XFORCE(J) = XFORCE(J) - FIJX
      YFORCE(J) = YFORCE(J) - FIJY
      ZFORCE(J) = ZFORCE(J) - FIJZ
C
C  Update the virial.
C
      VIRIAL = VIRIAL + RSQ*FOVERR
20    CONTINUE
```

```
      J1 = JBEG
10    IF (J1.GE.JEND) GO TO 20
C  Fork the next iteration on the next thread
C  processing unit
      x_store_ts(J1,J1+1)
      x_fork 10
C  XDIFF: fork the thread starting from 12
C         on the current thread processing unit
      y_store_ts(J,J1)
      y_fork 12
      y_allocate_ts(XDIFF)
      y_allocate_ts(RSQ)
      y_release_tsag_done
      XDIFF    = X0(I) - X0(J)
      IF (XDIFF.LT.-SP1) THEN
         XDIFF = XDIFF + SP2
      ELSE IF (XDIFF.GT.SP1) THEN
         XDIFF = XDIFF - SP2
      END IF
      y_release_ts(XDIFF)
      y_store_ts(RSQ, RSQ + XDIFF*XDIFF)
      IF (RSQ.GT.RCUTSQ) THEN
         y_abort_future
         x_stop
      ENDIF
      y_stop
C  YDIFF: fork the thread starting from 14
C         on the current thread processing unit
12    y_fork 14
      ...
      IF (RSQ.GT.RCUTSQ) THEN
         y_abort_future
         x_stop
      ENDIF
      y_stop
C  ZDIFF: fork the thread starting from 16
C         on the current thread processing unit
14    y_fork 16
      ...
      IF (RSQ.GT.RCUTSQ) THEN
         y_abort_future
         x_stop
      ENDIF
      y_stop
C  Update number of pairs within cutoff
C  and force accumulators
16    x_allocate_ts(NUMPRS)
      x_allocate_ts(XFORCE(I))
      ...
      y_wait_tsag_done
      x_wait_tsag_done
      x_release_tsag_done
      ...
      x_store_ts(NUMPRS, NUMPRS + 1)
      ...
      FIJX = XDIFF*FOVERR
      ...
      XFORCE(J) = XFORCE(J) - FIJX
      ...
      x_store_ts(XFORCE(I),XFORCE(I)+FIJX)
      ...
      x_stop
C-------------------------------------------
20    CONTINUE
```

Figure 8.6: An example program from 034.mdljdp2 and its superthreaded code

the kind of loops that most existing multiprocessors and multithreaded architectures can do well. With the two-dimensional multithreading support, we can exploit parallelism not only from singly-nested DO loops but also from doubly-nested DO loops by executing the outer-loop iterations as x-threads and the inner-loop iterations as y-threads. Data dependences between loop iterations can be enforced efficiently using run-time data synchronization.

8.4 Hardware Support for Two-dimensional Superthreading

As shown in Figure 8.1, a two-dimensional superthreaded processor consists of multiple thread processing elements to support the concurrent execution of x-threads. To support concurrent y-threads in each x-thread, each thread processing element provides multiple thread slots to hold the contexts of the resident y-threads. In addition, each thread processing element incorporates two levels of memory buffers to support run-time data dependence checking and to buffer speculative results for both y-threads and x-threads. The following two subsections describe the hardware mechanism of the thread slots and the memory buffers in each thread processing element.

8.4.1 Thread Slots and Instruction Execution

In the two-dimensional superthreaded architecture, each thread processing element can support multiple y-threads to hide long memory latencies caused by cache misses or data synchronization. When a y-thread executed on a thread processing element encounters a long latency operation, the thread processing element can keep its functional units busy by executing instructions from other concurrent y-threads. For this purpose, each thread processing element replicates some of its storage units to hold the context of the multiple y-threads.

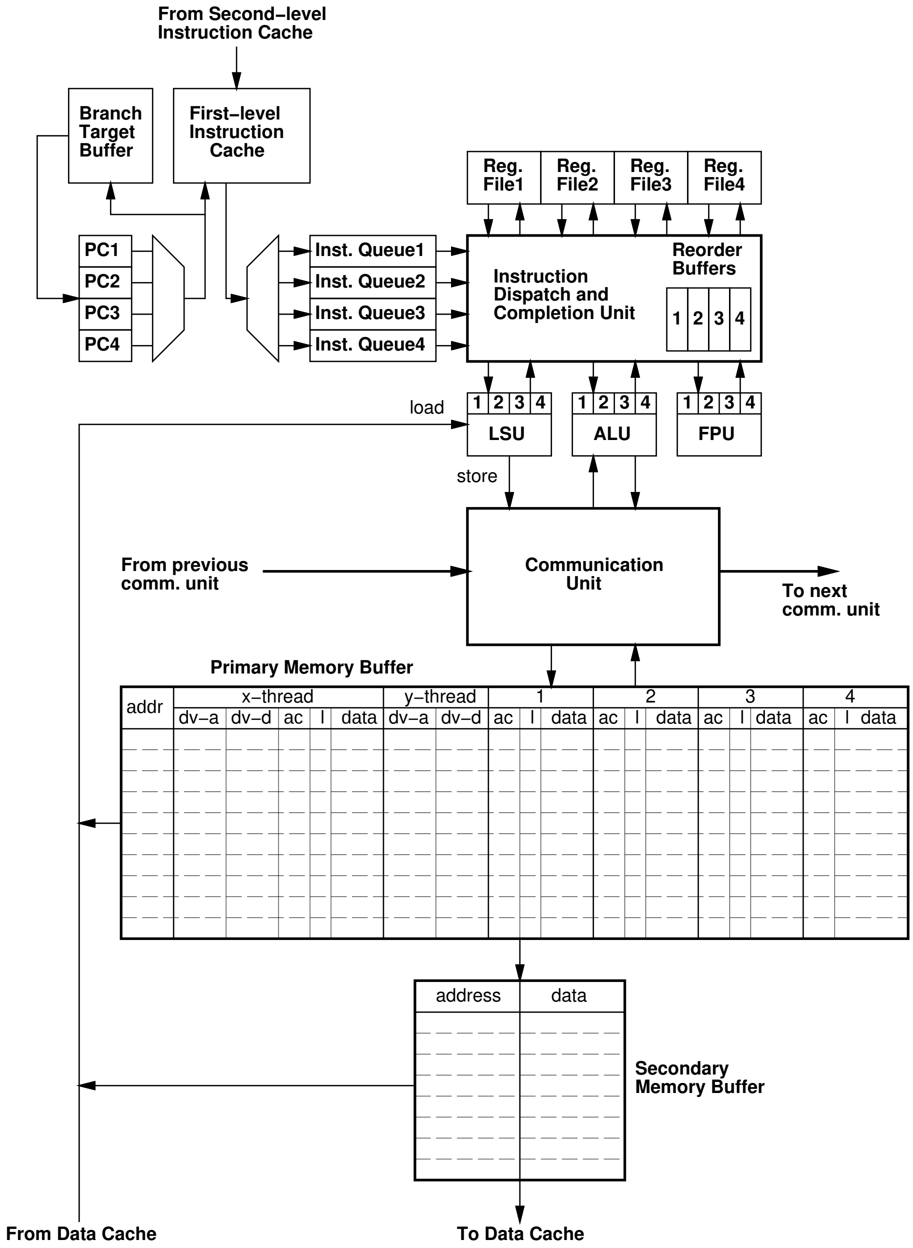

Figure 8.7: The block diagram of a thread processing element

Figure 8.7 shows the detailed block diagram of a thread processing element. To hold the context of a y-thread, each thread slot in the thread processing element includes a program counter, an instruction queue, a register file, and corresponding fields in the primary memory buffer. While these storage units are replicated for each thread slot, the instruction cache, the functional units, the communication units, and the secondary memory buffer are shared. At run time, y-threads are assigned to the thread slots in a round-robin order.

The instruction cache is shared by all thread slots in a cycle-by-cycle interleaved fashion. That is, the instruction fetch logic selects a thread slot and sends its program counter to the instruction cache in each cycle. The instructions fetched from the instruction cache will be fed into the corresponding instruction queue. In general, thread slots with fewer instructions in their instruction queues will be given higher priority for instruction fetch. In case of a tie,the instruction fetch logic will choose the oldest y-thread.

To allow instructions fetch beyond a conditional branch, the instruction fetch logic can use a conventional branch target buffer [24] to support instruction-level control speculation within each y-thread.

In every cycle, the instruction dispatch unit examines the top instructions in the instruction queues of all thread slots. If the required resources are available, the dispatch unit will send the instructions to the reservation stations of the appropriate functional units. Instructions of a thread slot are dispatched in order. To prevent instructions of a stalled threads filling the reservation station of a functional unit and blocking the execution of other other threads, each thread slot has its own reservation station in every functional unit.

Once instructions are dispatched to the reservation stations of a functional unit, they can be executed by the functional unit out of program order when their source operands are available. If two or more instructions in the reservation stations of different threads are ready to be executed, priority is given to the instruction from the oldest thread. Our scheduling policy is to execute instructions from earlier threads as soon as possible, since they may produce results needed by later threads.

Although instructions of a thread can be executed out of order, their execution results are completed in-order by using a reorder buffer[33] as in a conventional superscalar processor.

8.4.2 Memory Buffers

The primary and the secondary memory buffers of a thread processing element have four major functions:

- Performing run-time data dependence checking for data synchronization and data speculation,
- Providing a thread with dependent store data from predecessor y-threads or x-threads,
- Buffering uncommitted store data during thread-level speculative execution, and
- Buffering private store data of each thread to eliminate anti (read-after-write) and outpout (write-after-write) data dependencies between threads.

Both the primary and the secondary memory buffers can be organized and accessed as a set-associative or fully-associative cache. As shown in Figure 8.7, the primary memory buffer has multiple data fileds and status bits in each entry to keep track of the different copies of the store data from different threads. On the other hand, each entry of the secondary memory needs only one data field to store the most recent store data from the retired y-threads within the thread processing element.

Each primary memory buffer entry has an address tag and two portions of data fields, one for the x-thread and the other for the y-threads. The x-thread portion has only one data field to keep the most up-to-date x-target store data with respect to the current x-thread. However, each y-thread in the thread processing element has its own data field in the y-thread portion to holds the store data of a y-thread before it is retired, and to provide its successor y-threads with store data which they may depend on.

In the x-thread portion of each primary memory buffer entry, there are two *distance vectors*, one for address (*dv-a*) and the other for data (*dv-d*), to enforce read-after-write and write-after-write data dependences caused by x-target stores between x-threads. The initial values of the *dv-a* and the *dv-d* are all 0. When a primary memory buffer entry is assigned to an x-target store address generated by a x-thread with a distance of i (i.e. i x-threads away), it will record the address in its address field and mark bit i of its *dv-a* to 1.

Later, when the primary memory buffer receives an x-target store data from an x-thread, it will set the corresponding *dv-d* bit of the matched entry, and store the data only if it is from the nearest x-thread by checking the bit set in the *dv-d* to avoid the violation of the write-after-write dependences. To enforce read-after-write data dependences, a load whose address matches an x-target store entry must wait until the least-significant non-zero bit of the *dv-d* matches that of the *dv-a*.

The y-thread portion of a primary memory buffer entry also has a *dv-a* and a *dv-d* to enforce read-after-write data dependences between y-threads. Each bit of the y-thread portion *dv-a* and *dv-d* corresponds to an active y-thread within the current thread processing element. The corresponding bit of the *dv-a* or the *dv-d* is set when a y-thread performs a *y_allocate_ts* or a *y_store_ts* to the entry. When a y-thread performs a load whose address matches the entry, it must wait until the nearest preceding non-zero bit of the dv-d with respect to the y-thread matches that of the *dv-a*.

When the load/store unit (LSU) performs a load instruction, it will forward the request to both memory buffers and the data cache to get the most up-to-date data with respect to the thread executing the load instruction. Logically, the data in the y-thread portion of the primary memory buffer (produced by currently active y-threads) are more recent than those in the secondary memory buffer (produced by retired y-threads), which in turn are more recent then those in the x-thread portion of the primary memory buffer (produced by predecessor or the current x-threads). If the load address does not match any entry in both memory buffers, the load/store unit will fetch data from the data cache.

To support thread-level data speculation, each data field of a primary memory buffer entry has a *load bit (l)*, which will be set when a *load_tl* operation is performed on the entry. When a y-thread executes a *load_tl*, if its address matches an entry in the primary memory buffer, it will perform a data synchronization as described above. If the load address does not match any entry, it will read data from the secondary memory buffer or the data cache into the primary memory buffer and then mark its the load bit and continue its execution speculatively.

When primary memory buffer receives a target store which matches an entry, it will check if any of the load bits of the entry is set. If it does, a data dependence has been violated (because the data is read before it is updated). The threads and all its successor threads have to be aborted and re-executed. It is up to the compiler to decide whether such data speculation is useful.

In addition to the load bit, every data field has an *alias count (ac)* to handle possible aliases between target stores within a thread that cannot be detected at compile time. In the TSAG stage of a x-thread or a y-thread, it is possible that two or more target store addresses generated are the same. In this case, the target store data can be released only when all of the target stores corresponding to the address are executed. To enforce this, when a thread executes an *allocate_ts* instruction, the assigned entry will set the alias count of the corresponding data field of the entry to one. Later, if the thread executes another *allocate_ts* to the same entry, the alias count will be increaseed by one. When the thread performs a corresponding *release_ts* or *store_ts* instruction to the entry, the alias count will be decreased by one until it becomes zero. The thread can release the target store data and set the corresponding *dv-d* bit only when the alias count becomes zero. For x-target stores, the stare data will be forwarded to the primary memory buffer of the next thread processing element when the alias count becomes zero.

After a y-thread completes its computation stage, it will perform the write-back stage and copy the store data from its data field in the primary memory buffer to the secondary memory buffer. During the write-back, if a primary memory buffer entry is used only by the y-thread and its x-thread portion is empty, the entry can be reclaimed and used by other y-threads. After the write-back, the y-thread can retire and release the thread slot, which can then be assigned to a new y-thread.

8.5 Related Work

Processor architectures can exploit parallelism from multiple concurrent threads to achieve high system throughput. By providing additional storage to accommodate multiple threads, a processor can perform context switching among resided threads to hide long memory latency and to maximize processor utilization. On the other hand, a processor can execute multiple threads concurrently on multiple processing units to increase the number of instructions issued per machine cycle.

Most of the previous or ongoing research in multithreaded architectures uses multiple threads for only one specific purpose: to tolerate memory latency or to increase instruction issue rate. Systems that adopt multithreading for hiding latency include HEP [17], Horizon [22], Tera [2], and Alewife [1]. To tolerate memory latency, HEP, Horizon and Tera can accommodate more than one hundred threads per processor and allow fine-grained and fast context switching at every cycle. These machines superpipeline both the memory and ALU operations to increase bandwidth, but they do not use data caches. Therefore, every memory reference incurs a long latency and must be hidden by switching to other threads. The large number of of threads helps hide the long memory latency. However, it also increases the hardware cost significantly. In the mean time, there would not be enough parallelism in a program to hide all memory latencies. The Alewife employs coarse-grained multithreading to hide latency, where instructions from a given thread are executed until the thread issues a long-latency memory access. Although its coarse-grained multithreading requires more overhead to switch from one thread to another, the Alewife seeks to avoid memory latency by using a data cache in addition to hiding it.

To achieve high instruction issue rate, the XIMD [52], Elementary Multithreading [15], M-machine[10], Simultaneous Multithreading [47], Multiscalar [11, 36], and SPSM [8] machines allow instructions from different threads to be issued and executed concurrently. The Simultaneous Multithreading and SPSM machines support only independent parallel threads, such as parallel loops without cross-iteration dependences (i.e. *do-all* loops). While architectures such as the XIMD, the Elementary Multithreading, the M-machine, and the Multiscalar do support data synchronization and communication between threads.

To exploit more thread-level parallelism in programs, the SPSM and the Multiscalar support speculation on both control and data dependences between threads. With support for control speculation, a thread that is control dependent on some conditions determined by earlier threads can be initiated and speculatively executed before the results of the conditions are known. After the conditions are determined, a speculative thread is either committed or squashed. Such thread-level control speculation allows a *do-while* loop to be executed as a parallel loop. Data speculation support, on the other hand, allows a thread to speculatively assume *no dependence* and to execute load instructions before potentially dependent store instruction from earlier threads are executed. If a data dependence violation is detected at run-time, the thread that violates the data dependence will be squashed. Hardware support for data speculation can be very expensive, since all load and store address from all active threads must be saved for run-time detection of data dependence violations.

Among these architectures, only the Elementary Multithreading and the M-machine use multiple threads to both hide memory latency and increase instruction issue rate. In the Elementary Multithreading, a processor provides a number of thread slots and a larger number of register sets to accommodate multiple threads. At run-time only a few threads can be active and be assigned to the thread slots for execution. When an active thread executed on a thread slot encounters a long-latency memory access, the processor will assign the thread slot to another ready thread residing in a register set. This approach can increase instruction issue rate as well as hide memory latency. However, it requires a complex network between the shared functional units and the multiple register sets. In M-machine, a processor contains four execution clusters and can support up to six resident V-threads. A V-threads is similar to a standard process and is composed of up to four H-threads to be concurrently executed on the four execution clusters. On each cluster, H-threads of different V-threads are interleaved on a cycle-by-cycle basis over the shared execution resources. The M-machine allows only H-threads of the same V-thread to to communicate directly with each other via the interconnection between clusters. H-threads of different V-thread can only communicate through the memory.

8.6 Summary

Exploiting as much parallelism as possible from programs not only can reduce program execution time, but also can be expended to hide memory latency. In this chapter, we describe a two-dimensional concurrent multithreaded architecture which combines aggressive thread-level and instruction-level speculation on control dependences, runtime data dependence checking both within and between threads, compiler-controlled data speculation, and a thread pipelining execution model to exploit more program parallelism from complicated program structures such as doubly-nested Do-While loops, loops with ambiguous loop-carried data dependences, loops with multiple control blocks, etc. These programs structures cannot be exploited efficiently using existing compiler techniques and machine architectures. By exploiting more parallelism from programs, it can then be used to hide memory latency and to allow more efficient usage of processor recourses.

Chapter 9

Conclusions

Exploiting more parallelism from programs is the key to improving the performance of future microprocessors. While the fine-grained instruction-level parallelism available in a basic block or a small set of basic blocks is very limited, there is far more loop-level parallelism available in most programs. The concurrent multiple-threaded architectural models, which can exploit loop-level parallelism efficiently, have a great potential to be used in future microprocessor design. However, many loops in general application programs often have control dependences and data dependences between their loop iterations, which can prevent loop iterations from being executed concurrently. Therefore, it is very important for a concurrent multiple-threaded model to provide mechanisms to handle both control dependences and data dependences between threads efficiently.

In this thesis, we present the superthreaded architectural model for exploiting thread-level parallelism with run-time support for data dependence checking and control speculation. The superthreaded model uses a thread pipelining mechanism to enforce data dependences between threads. With the thread pipelining and the target store address and data forwarding schemes, each thread can perform run-time data dependence checking and synchronization on its own thread processing element. The memory buffering and the in-order thread completion schemes allow control dependent threads to be executed concurrently with control speculation.

To effectively utilize the hardware support for thread parallelization and to achieve high performance, the superthreaded architecture relies on the compiler to extract thread-level parallelism and to generate efficient superthreaded code. In this thesis, we also studied the compiler techniques for the superthreaded architecture. Through our study, we found that a lot of existing program analysis and transformation techniques developed for multiprocessor architecture can be used to generate concurrent threads for the superthreaded architecture. We also identified and developed several new program transformation techniques to exploit more parallelism in programs, and to reduce run-time overhead for data communication and dependence checking.

This thesis also presents an extension of the superthreaded architecture, called two-dimensional superthreading. The two-dimensional superthreaded architecture utilizes multiple threads to both hide memory latency and increase instruction throughput. In the two-dimensional superthreaded architecture, an x-thread executed on a thread processing element can fork a y-thread on the same thread processing element for latency hiding, or fork another x-thread on the next thread processing element. The two-dimensional supertreaded architecture allows all threads to directly communicate with each other through the memory buffers and the interconnection between thread processing element. In addition, the run-time data dependence checking and the compiler-directed data speculation support allow threads with ambiguous data dependences to be executed concurrently.

9.1 Main Results

We evaluated the performance of the superthreaded architecture through a detailed trace-driven superthreaded processor simulator by using eight benchmark programs written in C (two GNU utilities, two SPEC92 floating pointer programs, and four SPEC95 integer programs). The simulation results show that the superthreaded architectural model can further improve the performance of a single-threaded superscalar processor. On average, the superthreaded model with 8 thread processing elements can improve the speedup of 4-issue superscalar model from 2.36 to 5.63 (139% improvement), and improve the speedup of 8-issue superscalar model from 3.18 to 6.43 (102% improvement) for 8-issue per for the transformed portion of the programs.

We also studied the size of the memory buffer and the bandwidth requirement for the communication unit and the shared data cache. Our results show that a memory buffer of fewer than 100 entries is sufficient for all of the benchmark programs studied. For a 4-TPE, 4-issue superthreaded processor, the required communication bandwidth between two connected thread processing units is about 2 requests per cycle for most of the programs. On the other hand, the required cache bandwidth varies from program to program. For programs with high instruction throughput, a 4-bank data cache with two read ports and one write port would be needed to provide sufficient cache bandwidth.

We built a superthreading compiler prototype based on the SUIF paralleling compiler to automatically parallelize sequential Fortran programs and generate superthreaded codes. Our simulation results from the compiler-generated superthreaded codes show that the superthreading compiler prototype, which leverages existing parallelizing compiler techniques and use two simple passes for thread pipelining and code generation, can effectively parallelize SPEC95 floating pointer benchmark programs for the superthreaded architecture.

9.2 Future Research Directions

A complete implementation of the superthreading compiler with full support for the superthreaded architecture is currently under development at University of Minnesota. The compiler system includes a parallelizing front-end compiler for thread partitioning and thread pipelining and a optimizing back-end compiler for instruction scheduling and register allocation. The front-end compiler and the back-end compiler are integrated with a High-Level Information (HLI) interface [7]. With more dedicated compiler support for C program analyses and transformations, we expected the complete implementation of the superthreading compiler can effectively parallelize both C and Fortran programs and generate more efficient superthreaded codes.

In the mean time, a new execution-driven superthreaded processor simulator based on the SimpleScalar tool set [4] is also under development at University of Minnesota. With the actual execution of the compiler generated superthreaded codes, we will be able to further verify the superthreaded architecture and to experiment different data communication schemes and memory buffer organization. The new simulator will also support compiler-directed data speculation and thread-level value prediction to further study their impact on the performance of the superthreaded architecture.

Bibliography

[1] A. Agarwal, D. Chaiken, G. D'Souza, K Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. A. Yeung. The mit alewife machine: A large-scale distributed-memory multiprocessor. In *Proceedings of Workshop on Multithreaded Computers, Supercomputing 91*, November 1991.

[2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Conference Proceedings, 1990 International Conference on Supercomputing*, pages 1–6, June 11–15, 1990.

[3] Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. The anatomy of the register file in a multiscalar processor. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 181–190, November 30–December 2, 1994.

[4] .C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. In *Computer Architecture News*, volume 25 (3), pages 13–25, June 1997.

[5] Michael Butler, Tse-Yu Yeh, Yale Patt, Mitch Alsup, Hunter Scales, and Michael Shebanow. Single instruction stream parallelism is greater than two. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 276–286, May 27–30, 1991.

[6] Ding-Kai Chen and Pen-Chung Yew. Statement reordering for doacross loops. In *Proceedings of International Conference on Parallel Processing*, volume Vol. II, pages 24–28, August 1994.

[7] S. Cho, J.-Y. Tsai, Y. Song, B. Zheng, S. J. Schwinn, X. Wang, Q. Zhao, Z. Li, D. J. Lilja, and P.-C. Yew. High-level information - an approach for integrating front-end and back-end compilers. Technical report, Department of Computer Science and Engineering, University of Minnesota, February 1998. Technical Report #98-008.

[8] Pradeep K. Dubey, Kevin O'Brien, Kathryn O'Brien, and Charles Barton. Single-program speculative multithreading (SPSM) architecture: Compiler-assisted fine-grained multithreading. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 109–121, June 27–29, 1995.

[9] Matthew K. Farrens and Andrew R. Pleszkun. Strategies for achieving improved processor throughput. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 362–369, May 27–30, 1991.

[10] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The m-machine multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 146–156, November 29–December 1, 1995.

[11] Manoj Franklin and Gurindar S. Sohi. The expandable split window paradigm for exploiting fine-grained parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 58–67, May 19–21, 1992.

[12] Manoj Franklin and Gurindar S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 236–245, December 1–4, 1992.

[13] Manoj Franklin and Gurindar S. Sohi. Arb: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45 (6):552–571, May 1996.

[14] Robert H. Halstead, Jr. and Tetsuya Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, May 30–June 2, 1988.

[15] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase, and Teiji Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 136–145, May 19–21, 1992.

[16] Mike Johnson. *Superscalar Microprocessor Design.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[17] Harry F. Jordan. Performance measurements on HEP — a pipelined MIMD computer. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 207–212, June 13–17, 1983.

[18] George E. Daddis Jr. and H.C. Torng. The concurrent execution of multiple instruction streams on superscalar processors. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages I:76–83, August, 1991.

[19] Stephen W. Keckler and William J. Dally. Processor coupling: Integrating compile time and runtime scheduling for parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 202–213, May 19–21, 1992.

[20] Venkata Krishnan and Josep Torrellas. Executing sequential binaries on a multithreaded architecture with speculation support. In *Proceedings of the Workshop on Multithreaded Execution Architecture and Compilation*, January 27–28, 1998.

[21] Venkata Krishnan and Josep Torrellas. A clustered approach to multithreaded processors. In *Proceedings of the International Parallel Processing Symposium*, March 1998.

[22] J. T. Kuehn and B. J. Smith. The horizon supercomputing system: Architecture and software. In *Proceedings of Supercomputing 1988*, November 1988.

[23] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, May 19–21, 1992.

[24] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17(1):6–22, January 1984.

[25] Zhiyuan Li. Array privatization for parallel execution of loops. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 313–322, July 1992.

[26] Zhiyuan Li, Jenn-Yuan Tsai, Xin Wang, Pen-Chung Yew, and Bixia Zheng. Compiler techniques for concurrent multithreading with hardware speculation support. In *Proceedings of the 9th Workshop on Languages and Compilers for Parallel Computing, LNCS #1239*, pages 175–191, August 1996.

[27] Mikko H. Lipsati and John Paul Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 226–237, December 2–4, 1996.

[28] Pedro Marcuello and Antonio Gonzales. Control and data dependence speculation in multithreaded processor. In *Proceedings of the Workshop on Multithreaded Execution Architecture and Compilation*, January 27–28, 1998.

[29] Pedro Marcuello, Antonio Gonzales, and Jordi Tubella. Speculative multithreaded processors. In *Proceedings of the 1998 International Conference on Supercomputing*, July 13 - 17, 1998.

[30] Andreas Moshovos, Scott E. Breach, T.N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24rd Annual International Symposium on Computer Architecture*, pages 181–193, June 2-4, 1997.

[31] R. Guru Prasadh and Chuang lin Wu. A benchmark evaluation of a multi-thread risc processor architecture. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages I:84–91, August, 1991.

[32] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, December 1–3, 1997.

[33] J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 36–44, June 1985.

[34] James E. Smith and Sriram Vajapeyam. Trace processors: Moving to fouth-generation microarchitectures. *IEEE Computer*, 30(9):68–74, September 1997.

[35] M. D. Smith. Tracing with pixie. Technical report, Stanford University, Stanford, California 94305, November 1991. Technical Report CSL-TR-91-497.

[36] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 22–24, 1995.

[37] Gurindar S. Sohi and Manoj Franklin. High-bandwidth data memory systems for superscalar processors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, April 8–11, 1991.

[38] J. Gregory Steffan and Todd Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the Fourth International Symposium On High-Performance Computer Architecture*, pages 2–13, January 31–February 4, 1998.

[39] J.-Y. Tsai, Z. Jiang, Z. Li, D.J. Lilja, X. Wang, P.-C. Yew, B. Zheng, and S. Schwinn. Superthreading: Integrating compilation technology and processor architecture for cost-effective concurrent multithreading. In *Journal of Information Science and Engineering*, volume Vol 12, No.1, March 1998.

[40] Jenn-Yuan Tsai, Zhenzhen Jiang, Eric Ness, and Pen-Chung Yew. Performance study of a concurrent multithreaded processor. In *Proceedings of the Fourth International Symposium On High-Performance Computer Architecture*, pages 24–35, January 31–February 4, 1998.

[41] Jenn-Yuan Tsai, Zhenzhen Jiang, and Pen-Chung Yew. Program optimization for concurrent multithreaded architecture. In *Proceedings of the 10th Workshop on Languages and Compilers for Parallel Computing*, August 1997.

[42] Jenn-Yuan Tsai, Zhenzhen Jiang, and Pen-Chung Yew. Compiler techniques for the superthreaded architectures. *To appear in International Journal of Parallel Programming - Special Issue on Languages and Compilers for Parallel Computing*, June 1998.

[43] Jenn-Yuan Tsai and Pen-Chung Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques, PACT '96*, pages 35–46, October 20–23, 1996.

[44] Jenn-Yuan Tsai, Bixia Zheng, and Pen-Chung Yew. Improving instruction throughput and memory latency using two-dimensional superthreading. *To appear in Proceedings of The Second European Parallel and Distributed Systems Conference*, July 1998.

[45] Jordi Tubella and Antonio Gonzalez. Control speculation in multithreaded processors through dynamic loop detection. In *Proceedings of the Fourth International Symposium On High-Performance Computer Architecture*, pages 14–23, January 31–February 4, 1998.

[46] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 22–24, 1996.

[47] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 22–24, 1995.

[48] Gary S. Tyson and Todd M. Austin. Improving the accuracy and performance of memory communication through renaming. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 218–227, December 1–3, 1997.

[49] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, April 8–11, 1991.

[50] Nancy J. Warter, Grant E. Haab, John W. Bockhaus, and Krishna Subramanian. Enhanced modulo scheduling for loops with conditional branches. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 170–179, December 1–4, 1992.

[51] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Samam P. Amarasinghe, Jennifer M. Anderson, Steven W.K. Tjiang, Shi-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. Suif: An infrastructure for research on parallelizing and optimizing compilers. In *ACM SIGPLAN Notices*, volume 29(12), December 1994.

[52] Andrew Wolfe and John P. Shen. A variable instruction stream extension to the VLIW architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–14, April 8–11, 1991.

[53] Michael J. Wolfe. Optimizing supercompilers for supercomputers. Technical report, University of Illinois, October 1982. Technical Report UIUCDCS-R-82-1105.

Vita

Jenn-Yuan Tsai was born on January 12, 1965 in Taichung, Taiwan, Republic of China. He entered the National Chia-Tung University in 1983 and received the B.S. in Computer Engineering in 1987. He then entered National Taiwan University for his graduate studies and received the M.S. in Electrical Engineering in 1989. He was the recipient of the 1989 Long-Term Thesis Award from the Acer group for his master thesis. After receiving his M.S., he spent two years in the R.O.C. Air Force as a Quality-Control Engineering Officer. Following this, he joined Tecom Co. Ltd. where he developed software for a wireless key telephone system. In the fall of 1992, he began his graduate studies in Computer Science at the University of Illinois at Urbana-Champaign. From 1992 to 1996, he was a research assistant in the Center for Supercomputing Research and Development at the University of Illinois. From 1996 to 1998, he was a pre-doctoral research assistant in the Department of Computer Science at the University of Minnesota. He completed the Ph.D. in Computer Science in May, 1998. After completing his Ph.D. research, he will join Hewlett-Packard Company as a software engineer.