The Superthreaded Architecture: Thread Pipelining with Run-time Data Dependence Checking and Control Speculation *

Jenn-Yuan Tsai

Center for Supercomputing R & D
University of Illinois
Urbana, IL 61801-1351
jytsai@csrd.uiuc.edu

Pen-Chung Yew

Department of Computer Science
University of Minnesota
Minneapolis, MN 55455-0519
yew@cs.umn.edu

Abstract

This paper presents a new concurrent multiple-threaded architectural model, called superthreading, for exploiting 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 architectural model can exploit loop-level parallelism from a broad range of applications.

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 [2, 13, 17].

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 also 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 [9]. Also, in VLIW architectures, the code size will expand exponentially because a single-threaded code that includes all possible combinations of branch conditions needs to be generated. This problem is especially serious when a compiler attempts to software pipeline a loop with conditional branches [18].

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 [9]. In VLIW architectures, the sub-operations of each long-word are executed in a lock-step fash-

*This work is supported in part by the National Science Foundation under Grant No. MIP 93-07910, MIP 94-96320, and CDA 9502979; by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order No. D 346; and by a gift from Intel Corporation.

ion 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 of the stalled sub-operation. 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 paper, we propose a concurrent multiple-threaded model, called *superthreading*, as an alternative. It uses a thread pipelining execution model to initiate and execute threads on the fly. It also supports run-time dependence checking and data forwarding to efficiently enforce data dependences between threads.

The remainder of this paper is organized as follows. Section 2 gives the background of the concurrent multiple-threaded models. In Section 3, we present the superthreaded microarchitecture and its execution model. Section 4 describes some important compilation techniques for the superthreaded model. We conclude the paper in Section 5 with a discussion of future work.

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 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.

Many concurrent multiple-threaded processor architectures have been proposed and studied [1, 3, 4, 5, 6, 7, 8, 10, 11, 12, 14, 15, 16, 19]. Some of them [4, 7, 11, 14] are primarily for increasing system throughput by allowing multiple programs (one program for each thread) to be run concurrently. In this paper, we focus on models that are primarily for speeding up the execution of one single program. Among them, models such as *Simultaneous Multithreading* [16] and *SPSM* [3] allow tasks that are independent, such as the iterations of a do-all loop, to be executed in parallel. This restriction can simplify the design, but limits the exploitable parallelism.

Models such as *HEP* [10], *Tera* [1], *XIMD* [19], *Elementary Multithreading* [8], and *M-machine* [5, 12] allow data synchronization and communication between threads. These models rely on compilers to detect dependences between threads, and to insert explicit data synchronization and communication commands in a program. They do not support run-time dependence checking. Hence, the compilers must be very conservative so that data dependences that are unknown at compile time (called *maybe* data dependences in this paper) will need to be enforced. For programs with a lot of *maybe* data dependences between threads, the performance will suffer if those *maybe* data dependences do not actually occur at run-time.

Some concurrent multiple-threaded models such as *Elementary Multithreading* [8], *Multiscalar* [6, 15], and *SPSM* [3] support 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 from an earlier iteration.

Multiscalar [6, 15] and *SPSM* [3] also support speculation on the *maybe* data dependences with the help of run-time address disambiguation hardware. In these models, the compiler 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 reexecuted 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. 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.

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 units with a pair of shared instruction and data caches, as shown in Figure 1. At run-time, the multiple thread processing units, each with its own program counter and instruction execution data path, can fetch and execute instructions from multiple program locations simultaneously.

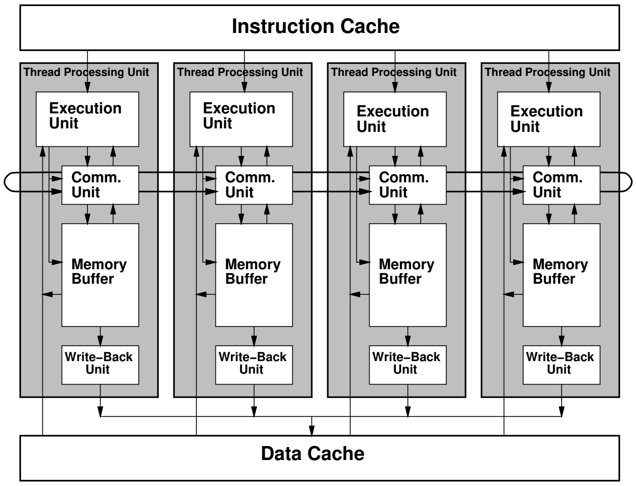

Figure 1: The organization of the superthreaded processor

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

The code segment shown in Figure 2 is one of the most frequently executed loops in the SPECint95 benchmark *130.li*. This loop walks down a nested list

```
while (TRUE) {
    if (this->n_flags & MARK)
        break;
    else {
        this->n_flags |= MARK;
        if (livecar(this)) {
            this->n_flags |= LEFT;
            tmp = prev;
            prev = this;
            this = prev->n_car;
            prev->n_car = tmp;
        } else if (livecdr(this)) {
            this->n_flags &= ~LEFT;
            tmp = prev;
            prev = this;
            this = prev->n_cdr;
            prev->n_cdr = tmp;
        } else
            break;
    }
}
```

Figure 2: An example code segment from *130.li*

pointed to by `this` as far as possible. The loop marks every node it traverses and reverses the direction of the links along the path it passes through. In the loop, there are two explicit data dependences between neighboring iterations caused by the recurrence variables `this` and `prev`. The compiler can partition the execution of the loop into threads, where each thread corresponds to a loop iteration.

The execution of a program starts from its entry thread. It can fork a successor thread on another thread processing unit. The successor thread can further fork its own successor thread. This process continues until all thread processing units are busy. In the program shown in Figure 2, a loop iteration can fork its next iteration after it updates the recurrence variable `this`. The superthreaded code for this program is shown later in Figure 4.

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 unit. 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

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 shows the pipeline stages of a thread and the relation between concurrent threads.

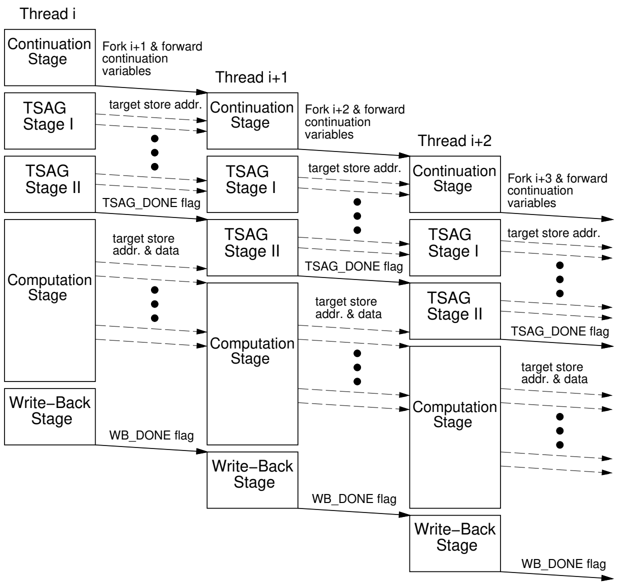

Figure 3: The pipelined execution of contiguous threads

3.1.1 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 the continuation variables will be forwarded to the next thread processing unit 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 unit. 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, 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.

3.1.2 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 unit (see Figure 5). 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.

3.1.3 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.

3.1.4 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 unit 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 units, because each thread is initiated and retired in order. The Nth successor thread will be initiated on the same thread processing unit 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.

3.1.5 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).

3.2 Thread Management and Communication Instructions

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

3.2.1 Fork

The *fork* <*address*> instruction starts a new thread on the next thread processing unit. When a *fork* instruction is executed, the thread processing unit activates the next thread processing unit, and copies part of the memory buffer to the next thread processing unit. 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 unit. The next thread processing unit then initializes its program counter with the given address and starts the execution of the new thread.

If the next thread processing unit is still busy, the thread processing unit will delay the *fork* instruction until the next thread processing unit completes its execution and becomes idle. The current thread processing unit 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 unit when the *fork* instruction is actually executed.

3.2.2 Target Store Operation

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 unit.

- *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 unit. 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 unit. 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 aliase count, which will be described in Section 3.3.3, to determine when a target store data can be released.

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

3.2.3 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 unit 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 unit executes a *stop* instruction, it waits for the *wb_done* flag from the previous thread processing unit 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.

3.2.4 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 units to serialize the write-back stages. However, the compiler can place a *wait_wb_done* instruction in a thread to force the thread processing unit 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.

3.2.5 Example

In this subsection we use the example in Figure 2 to show how these thread management and communication instructions are used. The superthreaded code for this program is shown in Figure 4. In the code, each thread corresponds to a loop iteration. In the continuation stage each thread computes the continuation variable `this`, which is needed, directly or indirectly, in the next thread. Unfortunately, the value of `this` is control dependent on some IF statements in the loop body. To allow the continuation stage to compute its value, those IF statements are also moved to the continuation stage. The computation of `this` is also dependent on the memory location pointed to by `this`. Because of the pointer data structure, the compiler cannot determine if the memory location will never be modified by a previous thread. To be safe, a *wait_tsag_done* instruction is placed before the computation of `this` to check the data dependence at run-

```
/* Continuation stage  */
L1:
    this_tmp = this;
    wait_tsag_done;
    if (cond1 = livecar(this))
        store_ts(&this, this->n_car);
    else if (cond2 = livecdr(this))
        store_ts(&this, this->n_cdr);
    fork L1;
/* Target-store-address-generation stage */
    allocate_ts(&prev);
    addr1 = &(this_tmp->n_flags);
    allocate_ts(addr1);
    addr2 = &(this_tmp->n_car);
    allocate_ts(addr2);
    addr3 = &(this_tmp->n_cdr);
    allocate_ts(addr3);
    release_tsag_done;
/* Computation stage */
    if (*addr1 & MARK) {
        abort_future;
        goto L2;
    } else {
        *addr1 |= MARK;
        if (cond1) {
            release_ts(addr3);
            store_ts(addr1, *addr1 | LEFT);
            tmp = prev;
            store_ts(&prev,this_tmp);
            store_ts(addr2, tmp);
        }
        else if (cond2) {
            release_ts(addr2);
            store_ts(addr1, *addr1 & ~LEFT);
            tmp = prev;
            store_ts(&prev,this_tmp);
            store_ts(addr3, tmp);
        } else {
            abort_future;
            goto L2;
        }
    }
    stop;
/* Write-back stage */
/*   -> performed automatically after stop */
/* End of thread pipelining */
L2:
    wait_wb_done;
```

Figure 4: The superthreaded code for the code segment shown in Figure 2.

time. After the recurrence value is computed and forwarded by a *store_ts* instruction, the next thread is activated by a *fork* instruction with control speculation.

The TSAG stage includes the computation of the target store addresses. These addresses are forwarded to the next thread through *allocate_ts* instructions and saved in registers for later use. In this code, four target store addresses are generated: `&prev`, `addr1`, `addr2` and `addr3`. The TSAG stage ends with a *release_tsag_done* instruction.

The code in the computation stage performs the main computation. It uses a *store_ts* instruction to perform the last write to each target store address, such as the write to `prev`. The target store address `addr3` will not be written if `cond1` is true. To avoid later threads from being suspended indefinitely on `addr3`, a *release_ts* instruction is used to release `addr3` when `cond1` is true. After the computation stage, a thread will either end with a *stop* instruction or exit the loop if one of the break conditions is detected. When a thread ends with a *stop* instruction, the processor will perform the write-back stage automatically and then retire the thread. When a loop exit condition is detected, it will execute an *abort_future* instruction to kill the later speculative threads. After the thread exits the loop, it will execute a *wait_wb_done* instruction and perform its write-back stage before it starts another portion of the program.

3.3 Superthreaded Processor Microarchitecture

Notice that 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 units can be connected by a unidirectional wrap-around ring to simplify the hardware complexity, as shown in Figure 1. Each thread processing unit has a communication link to its next thread processing unit along the ring. The commands and data are always forwarded by a thread to its successor thread in the same direction.

Figure 5 shows the detailed block diagram of a thread processing unit. A thread processing unit consists of four major parts: an execution unit, a communication unit, a memory buffer, and a write back unit. The following subsections describe the functions and some design issues of these units.

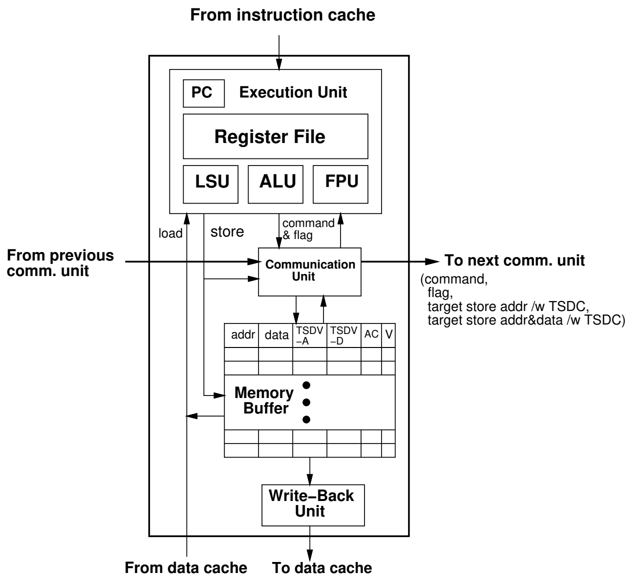

Figure 5: The block diagram of the thread processing unit

3.3.1 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 units may share some expensive functional units. However, the communication overhead for sharing should be less than the cost of the functional units.

Most of the instructions from a thread are executed by the execution unit alone. However, the execution of memory, thread management, and communication instructions will go through the communication unit and the memory buffer.

3.3.2 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.

3.3.3 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 units.
- *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 unit can resume the execution of the thread.

3.3.4 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 unit 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.

4 Compiler Support

With the global information available to the compiler, it can partition the computation of a program into threads and can generate code for the superthreaded processor. The performance depends strongly on how well the compiler can partition the computation to reduce the run-time overhead through more accurate program analyses. On the other hand, the run-time support for data dependence checking and control speculation provided by the superthreaded model allows more potential parallelism to be exploited. In this section we briefly describe the required compiler support.

4.1 Thread Generation

Before partitioning a program into threads, the compiler regards the entire control flow graph of the program as a single thread. In the process of thread generation, the compiler examines each portion of the control flow graph and identifies threads for pipelined execution. An ideal candidate for a thread is a group of instructions which perform a round of computation on a set of data. If the compiler can further extract the continuation portion and the target store address generation portion from the instructions, with a considerable amount of computation remaining for the computation stage, the compiler will generate a thread corresponding to this group of instructions, and will place a *fork* instruction after the continuation portion.

To generate threads from a program, the compiler requires the following program analyses.

- *Function inlining:* If there are function calls inside a thread, the compiler needs to perform function inlining to allow interprocedural data flow analysis and instruction movement across function boundaries.

- *Alias analysis:* For applications written in C or C++, which often use many pointers and structures, the compiler will encounter the alias problem. Although the superthreaded architecture can detect and handle aliases at run-time, more precise alias information will allow more accurate data dependence and flow analyses.

- *Data dependence analysis for WHILE loops:* Existing data dependence analysis algorithms can only be applied to scalar and array accesses inside a regular DO loop. However, as we try to exploit parallelism in WHILE loops using speculative execution, the compiler needs a more general data dependence analysis technique for WHILE loops to identify target stores that may cause cross-iteration dependences.

4.2 Thread Continuation

The continuation stage contains the computation that is essential to start the next thread. The data generated in the continuation stage will be forwarded to the next thread directly without forwarding their addresses first. A thread can immediately load and use the data generated in the continuation stage of the previous thread before receiving the *tsag_done* flag from the previous thread. With these features, the compiler can move the computation of the continuation variables to the continuation stage to allow new threads to be forked quickly. On the other hand, because the continuation stages of the contiguous threads are executed sequentially, the compiler must minimize the amount of computation in the continuation stage to prevent the sequential execution of the continuation stage from becoming the critical path.

The superthreaded model does not support direct data transfer between register files in two threads as in [5, 15]. However, the compiler can map the global registers, such as the stack pointer and the frame pointer, to some memory addresses. At run time, a thread can obtain the contents of the global registers from the corresponding addresses. When a thread updates the content of a global register, the data can be forwarded to the successor threads by means of target store data forwarding.

4.3 Target Store Address Generation

To extract the computation of the target store addresses for the TSAG stage, the compiler first performs data dependence analysis between current thread and its successor threads and marks the store operations that may have data dependences between threads as target stores. After marking the target stores, the compiler performs code motion to promote the instructions that compute the target store addresses to the TSAG stage. The addresses computed in the TSAG stage can be saved in registers and used in the following computation stage.

After the compiler extracts the address computation for the TSAG stage, it replaces every last store operation to a target store address with a *store_ts* instruction. In addition, if a control path taken at runtime will prevent some target store entries from being updated, the compiler needs to generate *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 for 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.

5 Conclusions and Future Work

Exploiting more parallelism from programs is the key to improving the performance of future microprocessors with more hardware resources. While the fine-grained instruction-level parallelism available in a basic block or a small set of basic blocks is very limited, there is plenty of loop-level parallelism available in most programs. The concurrent multiple-threaded architectural models, which can exploit loop-level parallelism flexibly and efficiently, have a great potential to become one of the major trends in future microprocessor design. However, the loops of general applications often have control dependences and data dependences between their loop iterations, which can prevent the iterations of a loop 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 efficiently.

In this paper we introduced 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 on its own thread processing unit. The memory buffering and the in-order thread completion schemes allow control dependent threads to be executed concurrently with control speculation.

This paper also described the superthreaded microarchitecture and the required compiler techniques for generating efficient superthreaded code. As both the processor hardware and the compiler techniques for the superthreaded model are quite feasible with respect to today's VLSI and compiler technologies, we believe that the superthreaded model will be able to be realized on a real microprocessor with full support from the compiler in the near future.

To investigate the performance issues of the superthreaded architectural model, we are currently implementing a cycle-by-cycle, trace-driven simulator. We are also developing a compiler system, which includes a front end for program analysis and a back end for code generation, for the superthreaded model. Combining these efforts, we can further study and improve the performance of the superthreaded architectural model and its compiler for real applications.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] Marco Fillo, Stephen W. Keckler, Dally William J, 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.