Improving Instruction Throughput and Memory Latency Using Two-Dimensional Superthreading

Jenn-Yuan Tsai[†], Bixia Zheng[‡], and Pen-Chung Yew[‡]

[†]Department of Computer Science
University of Illinois
Urbana, IL 61801

[‡]Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

Corresponding author:
Jenn-Yuan Tsai
4-192 EE/CS Building
200 Union Street, SE
Minneapolis, MN 55455-0159
USA
E-mail: jtsai@cs.umn.edu
Phone: +1-612- 625-7876
Fax: +1-612-625-0572

Abstract

In this paper, 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 program parallelism both at the compile time and at the run time. It can then expend such program parallelism for memory latency hiding.

Such an architecture allows 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.

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[4], 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 paper, we describe a two-dimensional concurrent multi-threaded architecture, called superthreeaded 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 paper, we describe its program execution model in section 2. In section 3, we describe some possible hardware implementations of such an execution model. Some related work is described in section 4. We conclude our paper in section 5.

2 The Two-Dimensional Superthreaded Execution Model

2.1 Processor Overview

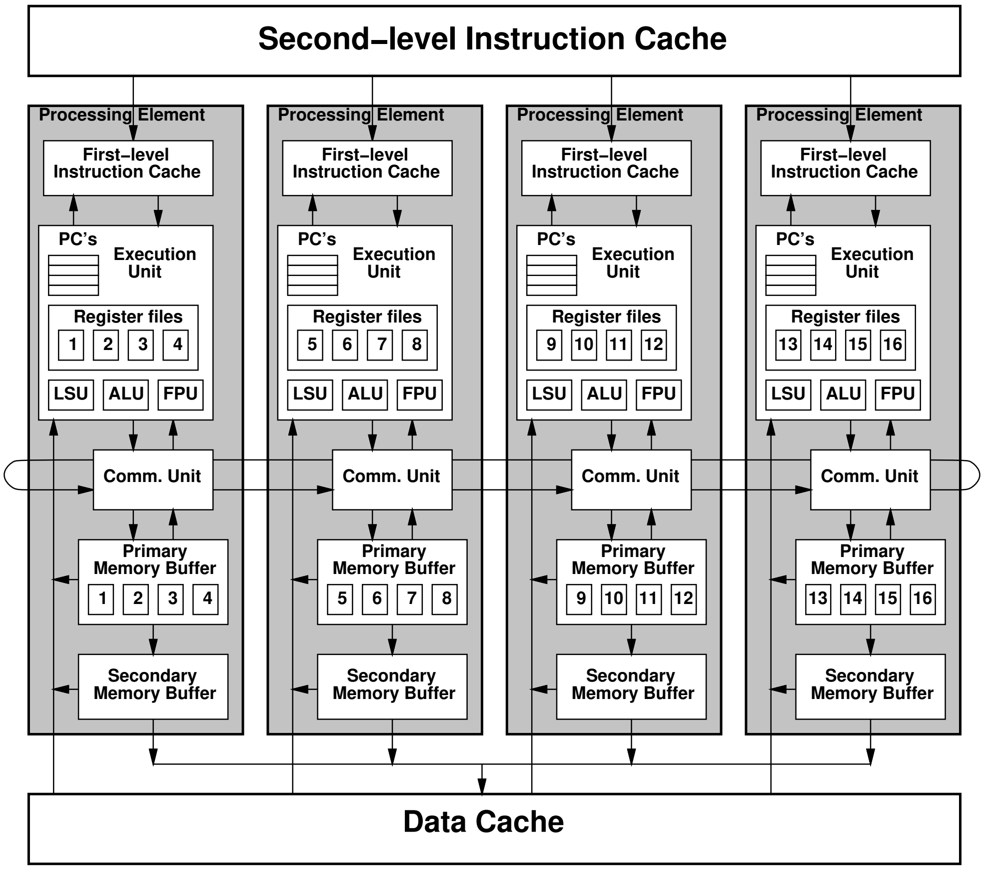

Figure 1: The organization of the two-dimensional superthreaded processor

As shown in Figure 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.

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 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.

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 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 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

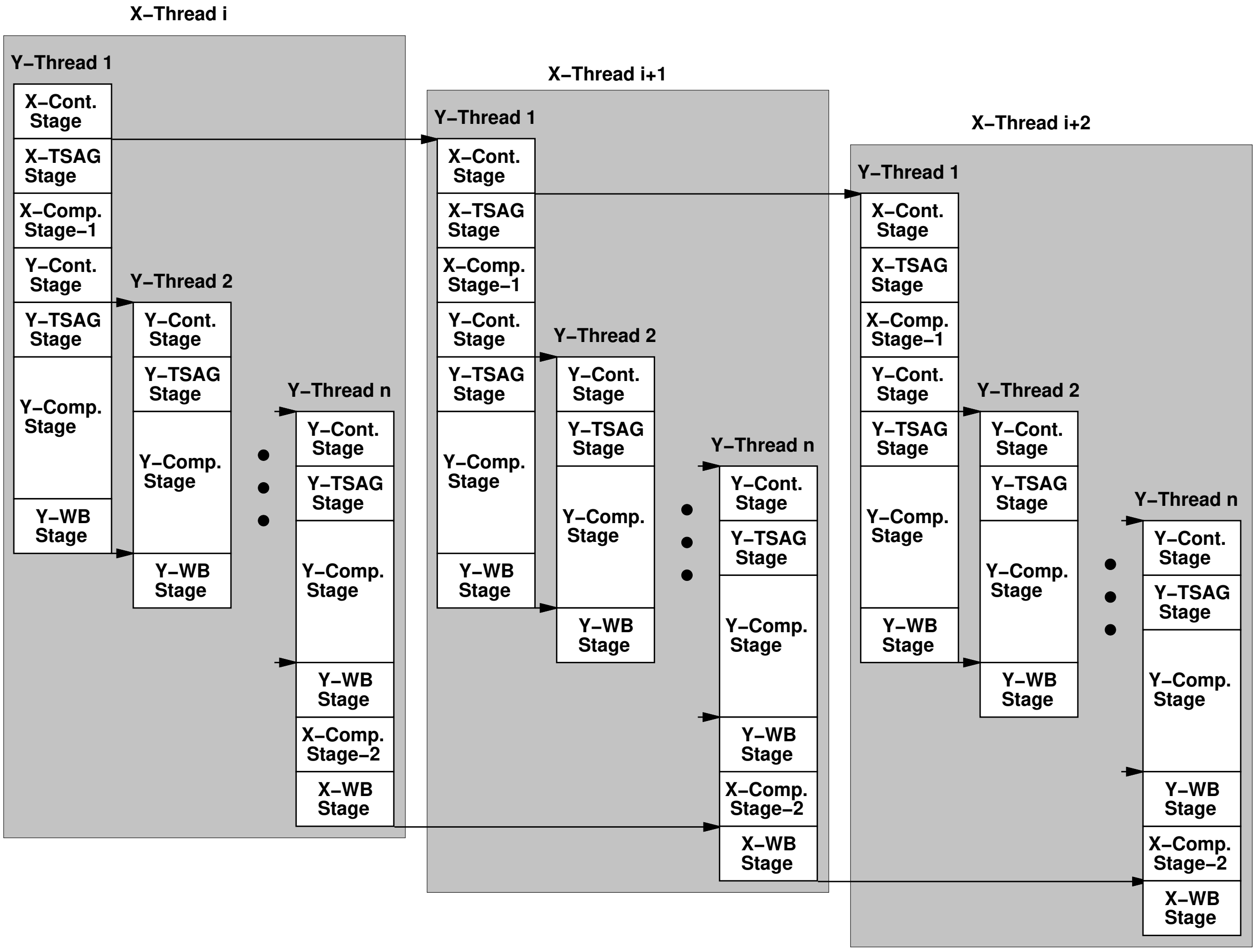

Figure 2: The pipelined execution of x-threads and y-threads

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.

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.

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*.

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.

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.

2.3.4 An Example Program

The example in Figure 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.

3 Hardware Support for Two-dimensional Superthreading

As shown in Figure 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-

```
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 3: An example program from SPICE and its superthreaded code

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.

3.1 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.

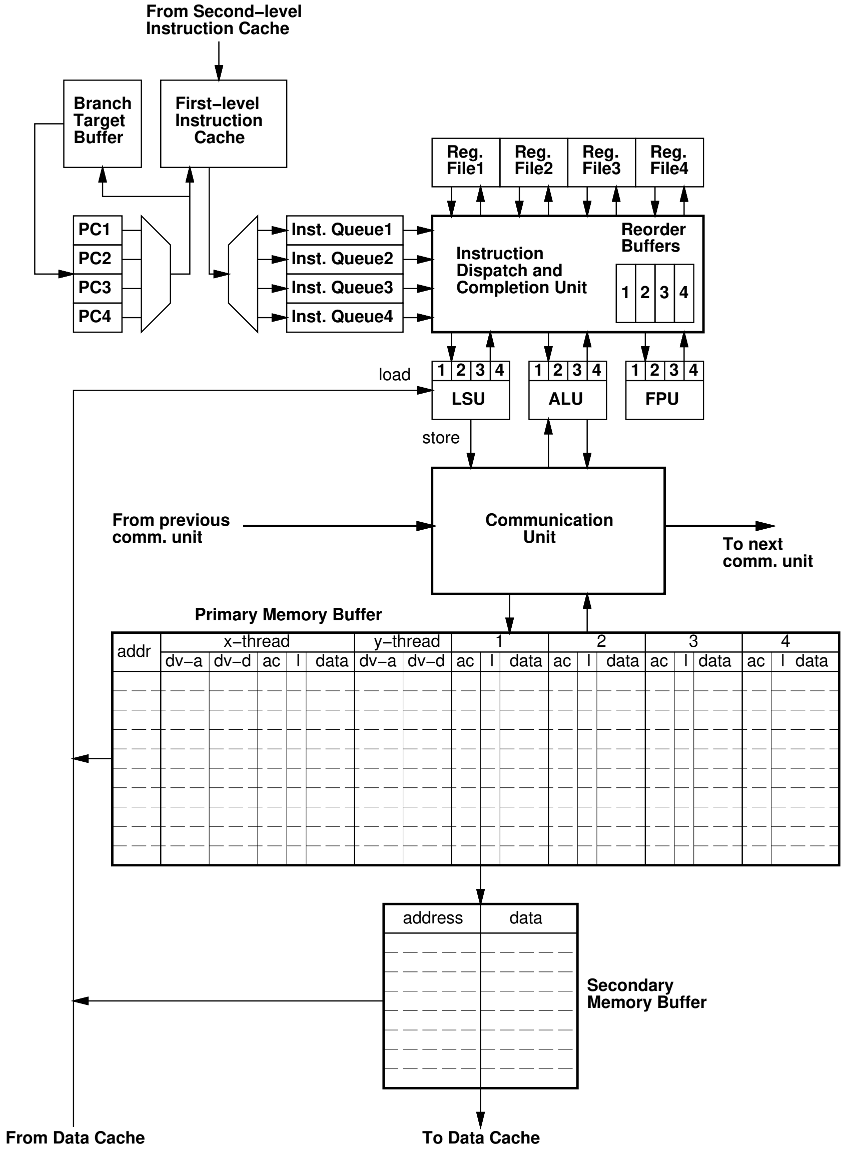

Figure 4: The block diagram of a thread processing element

Figure 4 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 [10] 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[11] as in a conventional superscalar processor.

3.2 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 4, 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.

4 Related Work

Most of the previous or ongoing research in multithreaded architectures use 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[8], Horizon[9], Tera[2], and Alewife[1].

To achieve high instruction issue rate, the XIMD[15], Elementary Multithreading[6], M-machine[4], Simultaneous Multithreading[14], Multiscalar[5, 12], and SPSM[3] 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). The XIMD, the Elementary Multithreading, the M-machine, and the Multiscalar do allow data synchronization and communication between threads, though.

To exploit more thread-level parallelism in programs, the SPSM and the Multiscalar support speculation on both control and data dependences between threads. The hardware for the full data speculation support can be very expensive, since all load and store address from all active threads must be saved for run-time checking.

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. 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. However, neither machine support thread-level speculation and run-time data dependence checking between threads.

5 Conclusion and Future Work

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 paper, 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.

To study the performance of such an architecture, we are modifying an existing single-dimension superthreaded architecture[13] simulator to perform cycle-by-cycle trace-driven simulation [7]. Compiler techniques are also being developed to take advantage of such an architecture. Preliminary results show that such an approach can be very useful to several time-consuming routines in some important benchmarks, such as SPEC95 and Perfect. However, more studies are needed to determine how much performance improvement can be achieved using such an approach.

References

[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 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] 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] 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.

[5] 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.

[6] 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.

[7] Zhenzhen Jiang. Performance study of superthreaded architectures. Master's thesis, University of Minnesota, Department of Computer Science, April 1997.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.