The Superthreaded Processor Architecture *

Jenn-Yuan Tsai, Jian Huang†, Christoffer Amlo‡, David J. Lilja‡, and Pen-Chung Yew†

Performance Delivery Lab
Hewlett-Packard Company
11000 Wolfe Road
Cupertino, CA 95014

jtsai@cup.hp.com

†Dept. of Computer Sci. and Engr.
‡Dept. of Electrical and Computer Engr.
University of Minnesota
Minneapolis, MN 55455
huangj,amlo,yew@cs.umn.edu
lilja@ece.umn.edu

Abstract

The common single-threaded execution model limits processors to exploiting only the relatively small amount of instruction-level parallelism available in application programs. The *superthreaded processor*, on the other hand, is a concurrent multithreaded architecture (CMA) that can exploit the multiple granularities of parallelism available in general-purpose application programs. Unlike other CMAs that rely primarily on hardware for run-time dependence detection and speculation, the superthreaded processor combines compiler-directed thread-level speculation of control and data dependences with run-time data dependence verification hardware. This hybrid of a superscalar processor and a multiprocessor-on-a-chip can utilize many of the existing compiler techniques used in traditional parallelizing compilers developed for multiprocessors. Additional unique compiler techniques, such as the conversion of data speculation into control speculation, are also introduced to generate the superthreaded code and to enhance the parallelism between threads. A detailed execution-driven simulator is used to evaluate the performance potential of this new architecture. It is found that a superthreaded processor can achieve good performance on complex application programs through this close coupling of compile-time and run-time information.

Keywords: multithreading, run-time dependence checking, speculation, compilers, performance evaluation.

* This work is supported in part by the National Science Foundation under Grant No. MIP 9610379; by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order No. D 346; by the Minnesota Supercomputing Institute, and by a gift from Intel Corporation. For more information, please visit http://www.cs.umn.edu/Research/Agassiz

1 Introduction

As the rapid progress of VLSI technology allows processor designers to incorporate more functional units on a single chip, future microprocessors will soon be able to issue more than a dozen instructions per machine cycle. Existing superscalar and VLIW microprocessors 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, however, the compiler or the hardware will have to extract more instruction-level parallelism from programs by analyzing a larger instruction window. Unfortunately, it is very difficult to extract enough parallelism with a single thread of control even for a small number of functional units [2, 10, 20].

A single-threaded sequencing mechanism has several major limitations in exploiting program parallelism. For example, to exploit instruction-level parallelism, independent instructions from different basic blocks in a single instruction stream need to be examined and issued together. As the issue rate increases, a larger instruction window is needed to contain several basic blocks, which may be control dependent on different branch conditions, but must 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 deeper levels of branch prediction and speculation, which quickly impairs the prediction accuracy [9]. Also, in VLIW architectures, the code size could expand exponentially since a single-threaded code needs to include all possible combinations of branch outcomes. This problem is especially serious when a compiler attempts to pipeline a loop with many conditional branches [21].

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 fashion to enforce the dependences between instructions. Any asynchronous event caused by any one of the sub-operations, such as a cache miss, will stall all of the 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 these limitations in the single-threaded execution model, it is important to consider other possible models for future microprocessors which will necessarily have more functional units and higher issue rates. The *superthreaded architecture* presented in this paper integrates compilation techniques and run-time hardware support to exploit both thread-level and instruction-level parallelism in programs [17, 18]. It uses a thread pipelining execution model to enhance the overlap between threads. It also supports compiler-directed thread-level control speculation and run-time data dependence checking. These features allow the compiler to exploit more potential thread-level parallelism in general-purpose applications.

In the remainder of this paper, Section 2 describes the superthreaded architectural model, while Section 3 presents the related compilation techniques. Section 4 evaluates the performance of the superthreaded architecture using a detailed cycle-by-cycle execution driven simulator. Section 5 then explores several design alternatives. Section 6 presents some related work. Section 7 concludes and summarizes the paper.

2 The Superthreaded Architectural Model

The superthreaded architecture, like other concurrent multiple-threaded architectures, exploits thread-level parallelism with multiple threads of control. In its general form, a superthreaded processor comprises a number of thread processing elements connected to each other with a unidirectional ring, as shown in Figure 1. The multiple thread processing elements each have a private level-one instruction cache, but share the level-one data cache and the level-two cache. There is also a shared register file that maintains some global registers and a lock register. At run-time, the multiple thread processing elements, each with its own program counter and instruction execution path, can fetch and execute instructions from multiple program locations simultaneously. Each thread processing element also has a private *memory buffer* to cache speculative stores and to support run-time data dependence checking.

The compiler statically partitions the control flow graph of a program into threads that correspond to a portion of the control flow graph. A thread performs a round of computation on a set of data which has no, or only a few, dependences with other concurrent threads. The compiler determines the granularity of the threads, which is typically one or several iterations of a loop.

The execution of a program starts from its entry thread. This thread can then 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 then 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, the thread performing the fork operation must ensure that all of the control dependences of the newly generated successor thread have

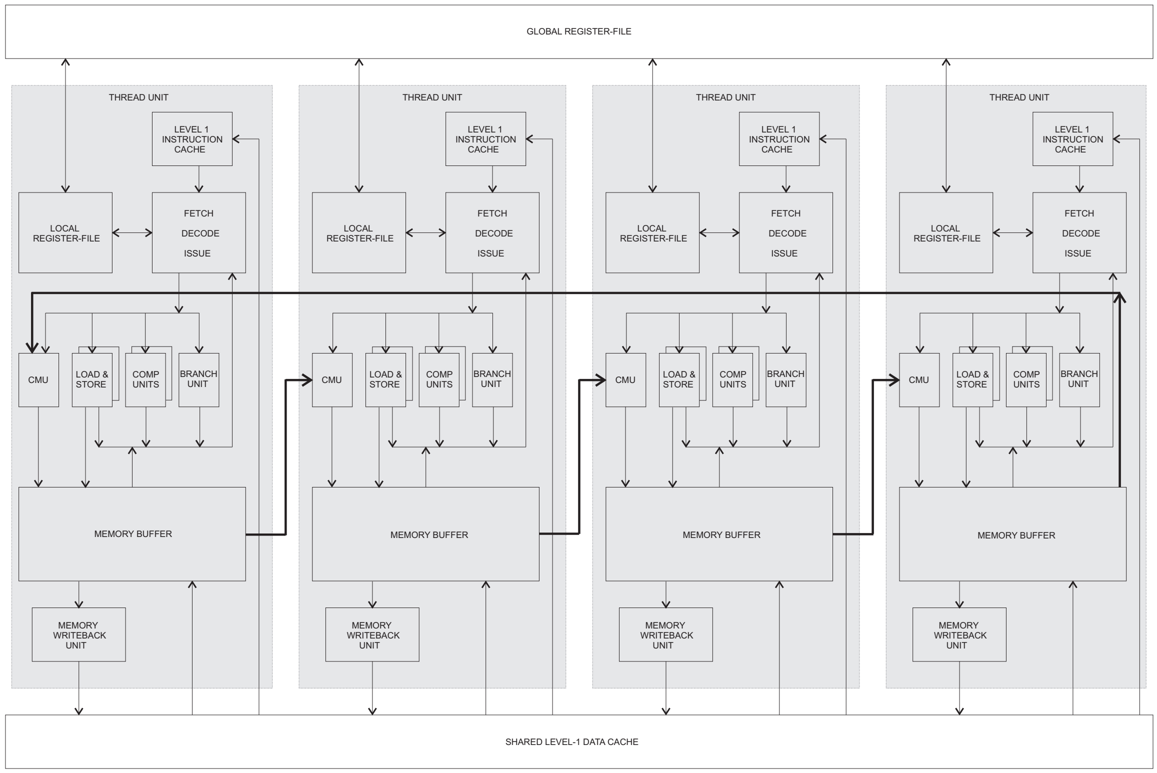

Figure 1: The organization of the superthreaded processor (CMU stands for Communication Unit).

been satisfied. If a thread forks a successor thread with control speculation, however, 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 subsequent threads.

2.1 Thread Pipelining Execution Model

The superthreaded architecture uses a *thread pipelining* execution model to enforce data dependences between concurrent threads. Unlike the instruction pipelining mechanism in a superscalar processor, where instruction sequencing, 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 in the multithreaded mode is partitioned into several stages, each of which performs a specific function, as described below. Figure 2 shows the execution stages of a thread and the relationship between concurrent threads.

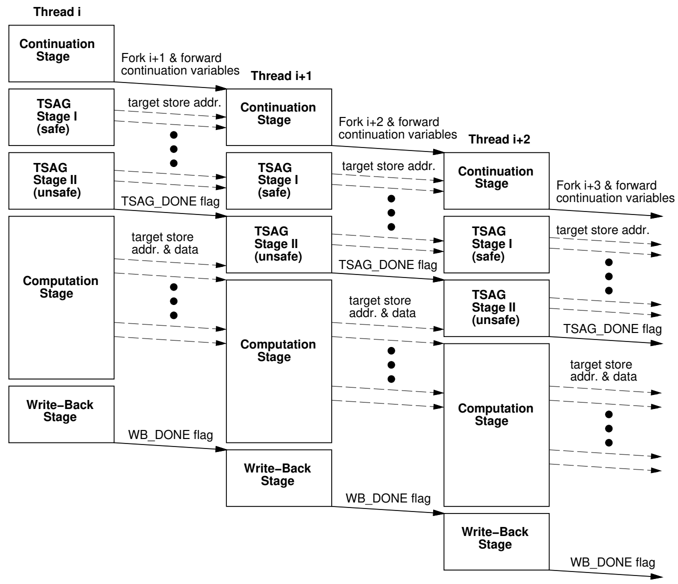

Figure 2: The thread pipelining execution model for contiguous threads.

2.1.1 Continuation Stage

After a thread is initiated by its predecessor thread, it begins executing its *continuation stage*. The major function of this 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 these 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 $p = p\text{->}next$, will be computed and forwarded to the next thread processing element. The continuation stage of a thread ends with a fork instruction, which causes the next thread to begin. As shown in Figure 2, the continuation stages of two adjacent threads will never overlap. This 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.

2.1.2 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 does not allow speculation on data dependences. To facilitate run-time data dependence checking, the addresses of these target stores need to be calculated as early as possible. The *target-store-address-generation* (TSAG) stage performs the address computation for these target stores. These addresses are stored in the memory buffer of each thread processing element and are forwarded to the memory buffers of all succeeding concurrent threads.

After a thread completes the TSAG stage and all of the target store addresses have been forwarded, it sends a *tsag_done* flag to the successor thread. This 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 of the target stores of its active predecessor threads, except those previously computed continuation variables.

To increase the overlap between threads, the target-store-address-generation stage can be further partitioned into two parts. The first part is for target store address generations that do not have any data dependences on earlier threads, called *safe* TSAGs. These target store addresses can be computed quickly and forwarded to the next thread. The second part computes *unsafe* target store addresses that may be data dependent on an earlier thread. These computations must wait for the *tsag_done* flag from the predecessor thread before beginning (see Figure 2).

2.1.3 Computation Stage

The *computation stage* performs the main computation of a thread. If the address of a load operation matches that of a target store entry in its memory buffer during this stage, the thread will either read the data from the entry if it is available, or it will 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 value needs to be forwarded with the corresponding target store address 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, ends with a *stop* instruction.

2.1.4 Write-Back Stage

If the control dependences are cleared after the computation stage, the thread completes its execution by writing all of the data from the store operations in its memory buffer to memory, including data from both target and regular stores. Data from store operations need to be kept in the memory buffer until this *write-back stage* to prevent the memory state from being altered by a speculative thread that is subsequently aborted by an earlier concurrent thread due to an incorrect control speculation. To maintain the correct memory state, concurrent threads must perform their write-back stages in their original order. To simplify the necessary control logic, 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 own 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 run-time. Note that a target store address need not be forwarded further after the data has been stored to memory. A target store needs to be forwarded to at most $N-1$ concurrent successor threads, where N is the total number of thread processing elements. 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 memory, the Nth successor thread will never need a target store address from the currently retired thread.

2.2 Superthreading Instructions

To facilitate thread pipelining and data forwarding, a set of thread management and communication instructions are provided as part of the processor's instruction set, which are described below.

2.2.1 Fork

The *st_fork* *<address>* instruction starts a new thread on the next thread processing element. When an *st_fork* instruction is executed, the thread processing element activates the next thread processing element, and forwards a portion of the memory buffer to that thread processing element. The forwarded portion includes the continuation variables computed in the continuation stage, the target store entries from the current thread, and the target stores from all its predecessor threads that have not yet retired. The next thread processing element then initializes its program counter with the given address, copies necessary register values from the global register file, and starts the execution of the new thread. The details of the initialization process are discussed in Section 5.1.

If the next thread processing element is still busy, the thread processing element will delay the *st_fork* instruction. The current thread processing element continues executing instructions following the *st_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 *st_fork* instruction is finally executed.

2.2.2 Target Store Instructions

The superthreaded model provides three special instructions to facilitate the generation and forwarding of target store addresses and data. All of these instructions have four versions for each data size, i.e., byte, half-word, word, and double-word [8].

- *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 that executes the corresponding *store_ts* instruction is not taken. If the data field of the entry is empty, null data will be forwarded.

2.2.3 Abort and Stop Instructions

When a thread determines that a control speculation is incorrect, it must kill all of the successor threads using the *abort_future* instruction. When a thread processing element receives an abort command, it propagates the command to the next thread and discards 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.

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 current thread retires and the thread processing element becomes idle if there are any valid succeeding threads. Otherwise, this instruction signals the end of a multithreaded region allowing the thread unit to continue executing the following instructions.

2.2.4 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 necessary 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 the 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 that executes the last iteration of a loop uses 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 provides two explicit synchronization instructions 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 that holds the lock, and then set the bit.

- *release_lock <lock_bit>:* This instruction resets the specified bit in the shared lock register to release a previously acquired lock.

2.3 Example Program

The code segment shown in Figure 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`. This loop is very difficult to parallelize by using conventional software pipelining techniques because of its control-flow intensive loop body and the conditional loop-carried data dependence. However, with the help of architectural support for multiple threads of control, control speculation, and run-time dependence checking, this loop can be easily parallelized and executed on a superthreaded architecture.

Figure 4 shows the superthreaded code for the loop. In this code, each thread corresponds to a loop iteration. We transform the execution of each loop iteration into three explicit thread pipelining stages and an implicit write-back stage as described in Section 2.1. We will discuss the compiler techniques to generate

```
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: An example code segment from 124.m88ksim.

superthreaded code in more detail in the next section.

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 original value of `i` is saved in `i_1` for later use locally. 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 its predecessor thread. This waiting is enforced with the *wait_tsag_done* instruction.

In the computation stage, a thread first begins by checking if the first exit condition is true. If it is, the thread will abort its successor threads with 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 stage, the update to the variable `minclk` is performed with 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 corresponding target store data. If both exit conditions are false, the thread will execute a *stop* instruction, and then automatically perform the write-back.

```
/* 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 4: The corresponding superthreaded code for the example shown in Figure 3.

3 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 code. 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 previously developed for traditional parallelizing compilers. We can then leverage these techniques for the superthreaded processor.

In addition to generating superthreaded code, the compiler can further enhance parallelism between threads and reduce run-time data dependence checking overhead by applying some program transformations unique to the superthreaded processor. In this section, we first present the program analysis and transformation techniques for program partitioning and thread pipelining, and then describe the advanced program transformation techniques used to enhance the performance of superthreaded processors.

3.1 Basic Compiler Techniques for Superthreading

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

- *Function inlining* to allow the compiler to perform data flow analysis and instruction movement across function boundaries for thread pipelining.
- *Inductive variable substitution* to eliminate loop-carried data dependences caused by inductive variables and to reduce the number of required target stores.
- *Alias analysis* for analyzing data dependences caused by pointers.
- *Data dependence analysis* to identify loop-carried data dependences for thread-level parallelism estimation and target store generation.
- *Variable privatization* to eliminate data dependences caused by variables whose updated values do not reach a concurrent successor thread.

3.2 Program Partitioning

To achieve the best performance for superthreading, the compiler should partition a program at a level where sufficient thread-level parallelism is available to compensate for the overhead of the parallelization. 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 a level where the maximum combined performance gains can be achieved from both superthreading and superscalar processing. In addition, to prevent overflowing the memory buffer, the maximum size of a thread must be limited [18].

To generate threads at the right level, the compiler examines data dependences between contiguous portions of a program and estimates the amount of thread-level parallelism at each level taking into account all of the data dependences. If the program has more than one level of loops with adequate parallelism, the compiler schedules the outer-most loop that would not overflow the memory buffer for thread-level execution, leaving the inner loop for each thread processing element to exploit the instruction-level parallelism (ILP). If only the 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 thereby provide each thread processing element with sufficient exploitable ILP. Another technique to optimize program partitioning is *loop interchanging* [22]. This technique can be used either to interchange an outer parallel loop that would 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.

3.3 Thread Pipelining

After partitioning a program into threads, the compiler further partitions each thread into the thread pipelining stages described in Section 2.1. The compiler first analyzes data dependences between contiguous threads and marks as *target stores* those store operations in each thread that could cause data dependences in its concurrent successor threads. The compiler then moves all of the instructions that are needed to compute the target store addresses to the TSAG (target-store-address-generation) stage. For each target store address computed in this stage, the compiler places an *allocate_ts* instruction 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 thread's computation stage.

After the TSAG stage is formed, the remaining computation in a thread becomes the computation stage. The last store operation to a target store address is replaced with a *store_ts* instruction to ensure that the last value written is the one that gets forwarded. 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 this 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. However, only a few of them may 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 of target store address generation, the compiler can move the condition tests to the TSAG stage and generate only these 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 in partitioning the thread pipelining stages 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 its TSAG stage. However, because the continuation stages of 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 must depend on only the continuation stages of the predecessor threads. Second, they must be needed in the TSAG stages of the current and successor threads. In general, the continuation stage contains instructions to update the value of recurrence variables, such as the loop index, which are typically 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 that are data dependent on its concurrent predecessor threads. In this case, the compiler needs to place a *wait_tsag_done* instruction before those instructions to delay their execution until the predecessor threads complete their TSAG stages and forward all of their target store addresses.

The compiler will try to increase the execution overlap of concurrent threads by minimizing the stall time caused by data dependences between threads. To do this, the compiler can perform statement reordering [3] and schedule target stores as early as possible. The compiler can also schedule load instructions that may be data dependent on the target stores of some predecessor threads as late as possible.

3.4 Advanced Compiler Techniques for Superthreading

3.4.1 Conversion of Data Speculation to Control Speculation

Some concurrent multiple-threaded architectures, such as the *Multiscalar* [6, 14] the *SPSM* [4], and the *trace processor* [12], 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, while relying on the hardware to detect any actual violations that occur 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 situation occurs, the hardware will squash the later thread and all of its successor threads.

Hardware support for data speculation can be very expensive, since a buffer such as the *Address Resolution Buffer* in the multiscalar [6, 14] is needed to track all *load* and *store* addresses from all active threads. To avoid using this type of expensive hardware for data speculation, 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 address resolution buffer in the multiscalar, for instance.

However, data speculation still 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 that exists when the control paths that contain the data dependences are not taken at runtime.

The compiler can perform this type of the data speculation by converting it to an equivalent control speculation, which is supported by the superthreaded 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 operation, for example, the compiler does not generate any target store addresses for the write. Instead, the compiler inserts an *abort_future* instruction in all of 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. Thus, control speculation is used to effectively implement data speculation.

The WHILE loop shown in Figure 5 is from the SPEC95 integer benchmark program *129.compress*. In this loop, each iteration computes the value of `fcode` and checks if it matches the value of the corresponding entry in the hash table (`htabof`). If it matches, the loop continues without interruption. If it does not match, the loop iteration updates the hash table, which introduces numerous data dependences to successor iterations. At run time, most of the loop iterations do not actually update the hash table. Thus, we can

```
while ((c = getbyte()) != EOF ) {
    in_count++;
    fcode = (long)(((long) c << maxbits)+ent);
    i = ((c << hshift) ˆ 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: An example code extracted from the SPEC benchmark program 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 6: The corresponding superthreaded code from the program in Figure 5 showing how data speculation is converted into control speculation.

profitably speculate on the data dependences caused by the update of the hash table using control speculation. The corresponding superthreaded code is shown in Figure 6. In this superthreaded code, we generate target store instructions only 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 transfers to `nomatch` and thereby updates the hash table, the thread will execute an *abort_future* instruction to squash the successor threads that may be data dependent on the updated hash table.

3.4.2 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 the *malloc* and *free* functions provided by the standard C library. The heap memory manager typically uses a free list to keep track of the free memory blocks available for allocation. Since allocation or deallocation of a memory block will modify both the free list and the memory blocks, heap memory management can cause potential data dependences between threads and so can create a bottleneck for thread parallelization.

To eliminate these 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 in this way, the allocation and deallocation of memory blocks on concurrent threads are 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 themselves. 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 so that 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 free memory blocks, the heap memory manager will invoke a redistribution routine to balance the number of free memory blocks among the thread processing elements.

The FOR loop shown in Figure 7 is from the SPEC95 integer benchmark program *130.li*. This loop traverses the linked-list `p` and frees any memory space pointed to by `p`. The deallocation of heap memory space causes data dependences between loop iterations on the system free list. To generate superthreaded code for this loop, we use the modified heap memory management routines described above to deallocate heap memory space as shown in Figure 8. In this superthreaded code, heap memory space is deallocated by using a function call to the new version of the library `st_free`, which maintains a free list for each thread processing element. The `st_free` function has an additional parameter which indicates the ID of the current thread processing element to free the memory space of the free list associated with the thread processing element.

3.4.3 Using Critical Sections 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 in a list in which the node 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 quickly obtain the updated data. Second, only non-speculative threads can enter the critical section to perform order-independent operations. This restriction prevents a speculative thread from performing an order-independent operation and writing uncommitted results to the data cache.

3.4.4 Memory Buffering in the Main Memory

The memory buffer saves target store addresses and data for run-time data dependence checking, and buffers uncommitted store data generated during speculative execution. The memory buffer can also be used to store private variables to thereby eliminate anti- and output dependences. Since the memory buffer has a finite size, the compiler must estimate the memory buffer usage and partition threads to avoid memory buffer overflows. When a memory buffer overflow occurs, the thread must be stalled until all of its predecessor threads are completed.

To exploit parallelism from a large outer loop while avoiding memory buffer overflow, the compiler can temporarily store the uncommitted store data, as well as the private store data, to a main memory segment dedicated to each thread processing element. A special store instruction is provided for this purpose that bypasses the memory buffer in the thread processing unit and directly writes 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-

```
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 7: An example code from the SPEC benchmark program 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 8: The corresponding superthreaded code in Figure 7 modified to use the distributed heap memory management.

time data dependence checking. When a thread becomes the head thread and has completed its computation stage, it must ensure that it writes back all of the store data buffered in the main memory.

4 Performance Evaluation

4.1 Simulation Methodology

The performance of the superthreaded architecture is evaluated using the SImulator for Multithreaded Computer Architectures (SIMCA) [8], which is an execution-driven simulator based on the SimpleScalar Tool Set [1]. Benchmark programs are transformed into their corresponding superthreaded programs at the source level. In the transformed superthreaded programs, special superthreading instructions, such as *st_fork* and *store_ts*, are represented using function calls. The transformed superthreaded program is then compiled by SimpleScalar's GCC compiler (Version 2.6.3) to generate the corresponding assembly code. A special parser is used to replace the function calls with appropriate superthreaded instructions. This modified assembly code is fed to the SimpleScalar *gas* assembler and *gld* loader to obtain the final binary code.

The simulator implements the thread processing element microarchitecture shown in Figure 9. The interconnection between thread processing elements is modeled as a unidirectional ring. This ring has a specific number of ports and a few cycles of line delay, both of which are parameterized. Each communication unit can forward a configurable number of commands or target store entries per cycle to the downstream communication unit and can interpret the messages from its upstream thread processing elements. Messages traveling from one thread processing element to another are delayed by an adjustable communication latency. The memory buffer is fully-associative with a configurable number of entries and ports. These ports are shared by the communication unit, the execution unit and the write-back unit. Each memory buffer entry contains one word and every read and write to the memory buffer takes one cycle.

The simulator also models a two-level cache hierarchy. Each thread processing element has a private level-one instruction cache, a shared level-one data cache, and a shared level-two combined instruction/data cache. To provide sufficient cache access bandwidth, the cache is non-blocking and can be made multi-ported, multi-banked, or both. In the following performance evaluation results, we used multi-port and multi-bank cache. The number of interleaved banks is scaled-up with the total issue rate (number of thread elements $\times$ issue rate per thread unit) of the processor. The basic configuration for the cache is:

- Level-1 data cache: 32 KB, write-back, 4-way set associative, 128-byte block, two read ports and one write port, 1 cycle hit-time.
- Level-1 instruction cache: 32KB, direct-mapped, 128-byte block, two read ports, 1-cycle hit-time.

- Level-2 cache: 2MB, write-back, 128-byte block, 4-way set associative, two ports for both read and write, 6-cycle hit-time, 30-cycle miss-penalty. The sensitivity of the performance to these delay parameters is analyzed in Section 5.5

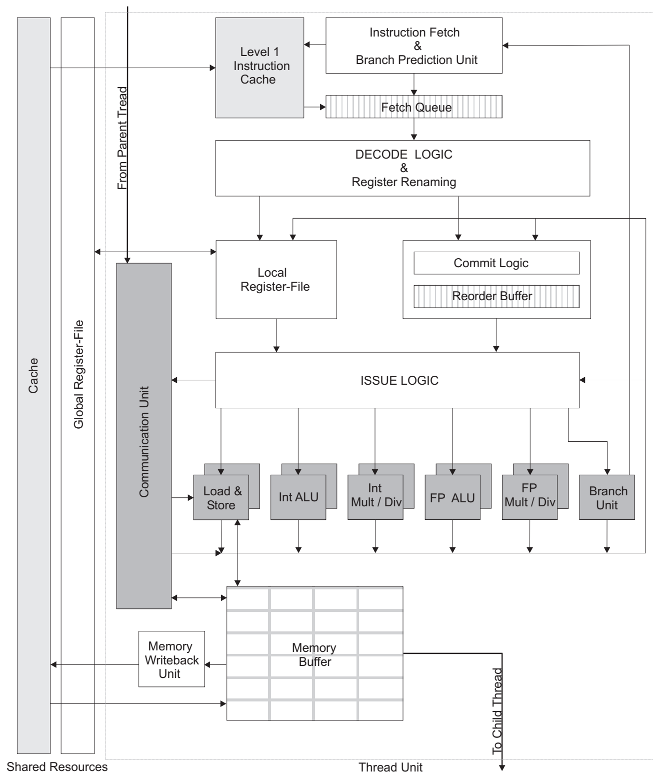

Figure 9: An example microarchitecture of a thread processing element.

4.2 Benchmark Programs

We use two GNU utilities (*wordcount* and *cmp*), three SPEC92 floating-point programs (*052.alvinn, 090.hydro2d* and *056.ear*) and three SPEC95 integer programs (*129.compress*, *132.ijpeg*, and *m88ksim*) for our performance study. All of these programs are written in or transformed to C, and were chosen to represent a range of parallelization difficulty. Since the compiler development for the superthreaded processor is still under way [11, 17], we manually transformed the most time-consuming routines of these test programs into the equivalent superthreaded code at the source level. This approach allows us to evaluate and study various architectural issues, design parameters, and the feasibility of the thread pipelining execution model before its compiler is fully functional.

Transformations	wc	cmp	hydro2d	alvinn	ear	compress	ijpeg	m88ksim
Function inlining						√		
Induction variable substitution	√	√	√	√			√	√
Loop unrolling		√		√	√			√
Loop interchanging				√				
Statement reordering to increase overlap	√					√		√
Converting data speculation to control speculation						√		

Table 1: The program transformations used in manually transforming the benchmarks into equivalent superthreaded programs.

To faithfully mimic the action of the superthreaded compiler, we limited ourselves to use only those compiler techniques described in Section 3 during the manual transformation. Table 1 summarizes the program transformation techniques that we actually employed for each of the benchmarks. *Hydro2d* contains many two-level loop nests. Parallelizing the outer loops in this program would overflow the memory buffer (currently set to 128 entries). Thus, we mostly parallelize the inner loop body for our experiments. However, we did allow some outer loops to run in parallel if the number of memory writes in the inner loop was estimated to be very close to the memory buffer capacity to test the performance of the superthreaded processor with memory buffer overflows.

Even though the manual transformation tries to mimic the compiler algorithms as closely as possible, the transformed programs should still be interpreted as the results of a good compiler with profiling information. On the other hand, the manual transformations are applied only at the source code level. The SimpleScalar *gcc* compiler then generates executable code without any specific consideration for the superthreaded execution model. Some performance degradation could result from this two-step compilation process. We thus restrict our performance comparison to varying architectural design parameters, while not comparing the results directly to other similar architectures [4, 5, 6, 7, 12, 14, 19].

Table 2 shows the input sets used for the simulation of the benchmark programs, the dynamic instruction counts of the original programs, the percentage of code that is transformed into superthreaded code, the dynamic instruction counts after the transformation, and the resulting increase in the instruction counts. Table 2 also shows the average number of committed Instructions Per Cycle (IPC) of the original programs when executed on a single-threaded, two-issue superscalar processor, which is used as our baseline for later performance comparisons.

Program	Input set	Insn. Count Total (original)	Fraction Transformed (before)	Insn. Count Transformed (after)	Percent Increase	Base Line IPC
wc	expr.c from gcc	3.11M	99.7%	3.59M	15.4%	0.88
cmp	expr.c from gcc	2.69M	99.8%	3.00M	11.5%	0.87
052.alvinn	ref(20 iterations)	613.6M	88.6%	559.4M	2.9%	1.07
056.ear	short	812.7M	98.6%	846.9M	5.6%	1.13
129.compress	train	50.6M	90.2%	52.1M	12.3%	0.92
132.ijpeg	test	1046.1M	69.9%	1150.4M	10.0%	0.99
090.hydro2d	test	33.93M	91.1%	35.28M	4.0%	0.96
124.m88ksim	test	972.1M	75.8%	1095.6M	12.69%	0.97

Table 2: The dynamic instruction counts of the benchmark programs.

# of TPEs $\times$ Issue rate	1 2	1 32	2 16	4 8	8 4	16 2
Reorder buffer size	16	256	128	64	32	16
INT ALU	1	16	8	4	2	1
INT MULT	1	4	2	1	1	1
FP ALU	1	16	8	4	2	1
FP MULT	1	4	2	1	1	1
Communication bandwidth	-	-	2	2	2	2
Write back bandwidth	1	4	4	4	4	4
Data cache banks	1	4	4	4	4	4

Table 3: The simulation configurations used for the thread processing elements.

4.3 Performance Comparison

We ran the benchmark programs on the simulator with the number of thread processing elements ranging from 1 to 16, and the issue rate within each thread processing element ranging from 2 to 32. These issue rates are chosen so that our data will reflect the configurations of a potential future superscalar processor. The two configurations with a single processing element are equivalent to a comparable superscalar processor.

Note that we run the original benchmark programs on these single-threaded processor models, and run the transformed superthreaded programs, which have more run-time overhead than the original programs, on the multiple-threaded models. Also note that the total number of instructions that can be issued each cycle (i.e., the number of TPEs * issue rate) is held constant at 32.

In all of the simulation configurations, each thread processing element uses a 512-entry, 4-way associative branch target buffer and a 128-entry fully-associative memory buffer. The detailed simulation configuration for each model is given in Table 3. Note that the different configurations consist of approximately the same amount of superscalar hardware. In particular, the reorder buffer (instruction window) size, and the number of functional units are scaled up with the issue rate of each thread processing element, while the number of thread processing elements scales down. The communication bandwidth, write-back bandwidth (for store data), and the number of interleaved cache banks are kept constant, since the product of the issue rate and the number of thread processing elements is the same across different configurations.

To compare the relative performance among different superthreaded configurations, and between superthreaded processors and superscalar processors, we use a single-threaded, two-issue superscalar processor as our baseline. This approach allows us to compare the improvement due to the very wide superscalar approach, and the additional improvement obtained through superthreading. We use the total number of committed instructions in the sequential program as the normalized workload and divide this instruction count by the number of cycles it takes to complete execution for each configuration to produce the *Useful Instructions Per Cycle* (UIPC) performance metric. The superthreaded instructions that we introduce are used only for facilitating the execution model, but not for doing useful computation. Therefore, they are excluded from the UIPC calculation. Figure 10 shows the UIPC of the different configurations in Table 3.

We can see in this figure that the superthreaded model can further improve the performance of a single-threaded (1 TPE) superscalar architectural model for four out of the eight benchmark programs. For programs with high thread-level parallelism, such as *wc, alvinn, cmp*, and *hydro2d*, the UIPC achieved by combining superscalar with superthreading can reach 5 to 11, while a wide-issue superscalar processor with the same amount of hardware can only reach 2 to 4. On the other hand, programs with intensive loop-carried data dependences, such as *compress*, do not benefit as much from the superthreaded execution model. Both the wide-issue superscalar processor and a superthreaded processor have roughly the same potential for improving the performance of *ijpeg*. The wide-issue superscalar approach outperforms the superthreading approach for *Ear* and *M88Ksim* since they have good instruction-level parallelism and relatively small granularity threads. Taking into account the complexity of building a wide superscalar processor capable of issuing 32 instructions per cycle with a 256-entry reorder buffer, however, the superthreaded processor may still be a much better choice. The 8-thread-unit and 4-thread-unit configurations typically outperform other superthreaded configurations by about 30 to 60 percent. This result suggests that for the same amount of superscalar hardware investment, superthreaded processors capable of issuing 4 to 8 instructions per cycle within each thread processing element have the best price-performance ratio.

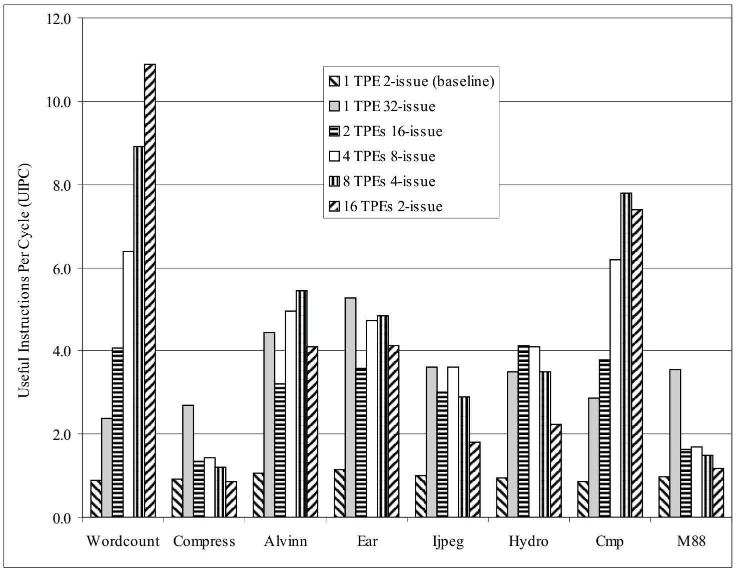

Figure 10: The performance of the superthreaded processors with the hardware configurations shown in Table 3.

Figure 11 shows the distribution of issued instructions for the different configurations. The "parallel overhead" refers to the committed superthreading instructions that are used to support this novel execution model, but not to actually compute program results. We see that the parallel overhead is the most severe for *alvinn* (over 60 percent). It is very small for *hydro2d* (below 10 percent), though, since this program has few inter-iteration data dependences. The number of instructions squashed due to thread misspeculation generally increases with the number of thread processing elements. Hence, the configuration with 16 thread processing elements typically does not show very good performance. The very wide-issue superscalar processor encounters severe branch-misprediction penalty when the issue width is 32 instructions per cycle. Since the superthreaded processor exploits loop-level parallelism with a relatively low instruction issue rate within each thread processing element, it more effectively avoids this branch misprediction overhead.

4.4 Memory Buffer Usage

The memory buffer of each thread processing element could be very expensive to implement since it needs to use associative hardware for run-time data dependence checking. Thus, the memory buffer should not be too

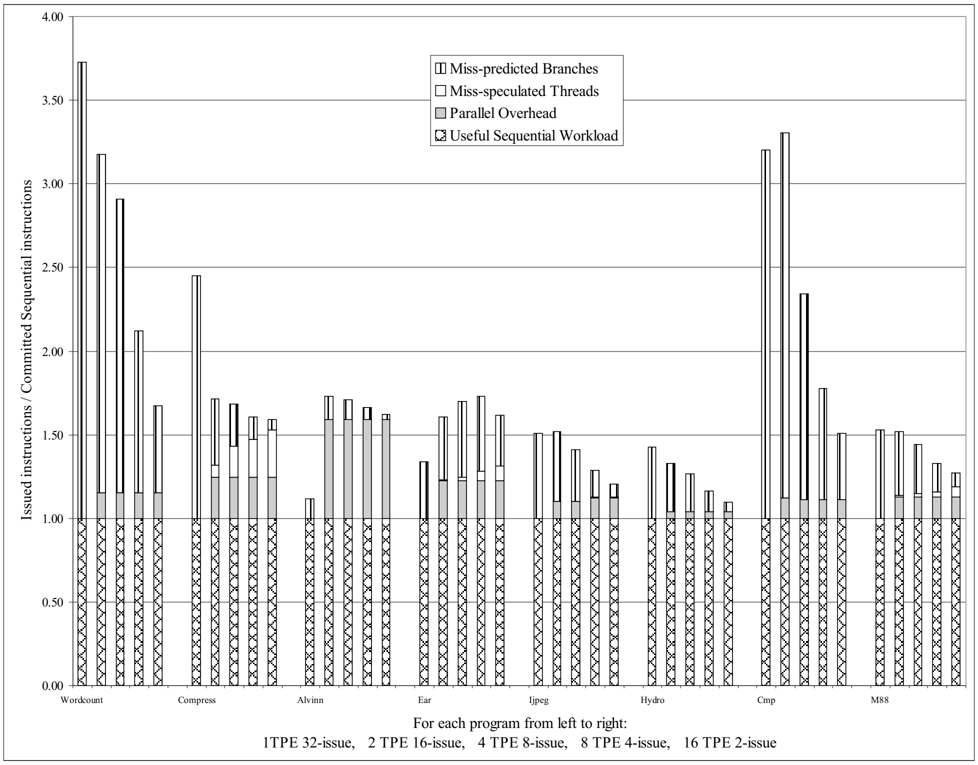

Figure 11: Distribution of issued instructions for the processor configurations shown in Table 3.

large. Table 4 shows the average and maximum number of memory buffer entries used in each benchmark program when executed in a superthreaded processor with different numbers of thread processing elements. Note that a memory buffer may collect more target store entries from its predecessor threads as the number of thread processing elements increases. The results show that using a small number of memory buffer entries (no more than 100) to buffer target stores and local stores is sufficient for most of the test programs.

For *hydro2d*, the maximum number of entries used is equal to the memory buffer capacity, but the average memory buffer usage is actually quite low. Thus, memory buffer overflows will be relatively infrequent. Even with this memory buffer overflowing, however, this program still generates very good UIPC performance compared to the wide-issue superscalar processor, as previously shown in Figure 10. This result suggests that if the memory buffer overflows very late in the thread execution stage, there is still considerable overlapped execution between thread processing elements.

The memory buffer could be split into two sections. One section could store only the TSAG data (data dependence buffer), while the other could store the memory writes before the thread becomes non-speculative (speculative-write buffer). Since the number of entries needed by the data dependence buffer is quite small (usually fewer than 3 entries per thread), making it fully-associative is easy. On the other hand, the speculative-write buffer can be made larger with lower associativity since it can be queried like a regular cache. The evaluation of this optimization, however, is beyond the scope of this paper.

Program (Average/Max)	Number of TPEs 2	4	8	16
wc	74/74	74/74	74/74	74/74
cmp	3/3	3/3	3/3	3/3
052.alvinn	18/73	28/73	33/73	34/73
056.ear	8/25	13/25	13/25	13/25
090.hydro2d	9/128	11/128	11/128	11/128
129.compress	9/13	9/14	9/15	9/15
132.ijpeg	7/27	9/27	10/27	9/27
124.m88ksim	2/21	2/21	2/21	2/21

Table 4: Memory buffer usage in each benchmark program for different numbers of thread processing elements (TPE).

5 Sensitivity Analysis

This section evaluates the sensitivity of the performance of the superthreaded processor to some of the important design parameters. In all of the following studies, we use a superthreaded processor configuration consisting of four 8-issue thread processing elements as our base model.

5.1 Thread Initialization Overhead

A program begins execution with an entry thread. This thread by default is in single-threaded mode. When this entry thread executes an *st_fork* instruction, the first multithreaded execution region begins. This multithreaded execution region ends when the last nonspeculative thread does not execute an *st_fork* instruction and the processor moves back to single-threaded superscalar processing. This kind of fork-join cycle continues until the program finishes execution. Each multithreaded region needs a global state with which to begin. This global state includes a global register file and other necessary processor state variables. The beginning global state for each multithreaded region is the same as the local state of the thread processing element that executes the first *st_fork* instruction in the region.

The global register file has a *full* bit attached to each register to indicate whether the register has been written. When the processor is in single-threaded mode, every write of a register by the sequentially executing thread will be propagated to the global register file asynchronously. These writes set the corresponding *full* bits.

This propagation process ends when the processor enters multithreaded mode. When a new thread is spawned, it checks the global register file and copies all the global registers with the *full* bit set to its local register file. This guarantees that all thread units start with the most up-to-date register file. The compiler can identify only those registers that need copying for the child thread to save the overhead of this global-to-local copying process. This copying process occurs only once for the spawning of each new thread, and only one new thread can be spawned at a time. Thus, the global register file needs to have enough bandwidth to support the execution of only a single thread and the bandwidth of the global register file will not be a bottleneck. The effect of this delay can be reduced if we start this global-to-local copying process early when a new thread is to be spawned. However, this optimization is beyond the scope of this paper.

In addition to this copying process, the current thread needs to forward to the next thread processing element the program counter value at which it should start executing. It also must forward the portion of its memory buffer entries needed for subsequent run-time data dependence checking. The child's private instruction cache also needs to be filled before it can actually start execution.

We define the *fork-delay* to include all of the above delays except the delay to forward memory buffer entries, since this delay varies greatly with the number of entries that need be forwarded. The *fork-delay* is on the critical path of program execution because it limits the overlapped execution among threads. We investigate the performance impact of 1, 2, 3, 4, 8, and 16-cycle *fork-delays*. From the results in Figure 12, we can see that the performance of the superthreaded processor is not particularly sensitive to the *fork-delay* when it is below 5 cycles. The performance degrades drastically, however, when this delay increases above 5 cycles. A 16-cycle *fork-delay* slows down the execution by about 8 to 25 percent for most of the programs compared to the 1-cycle case.

5.2 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 and synchronization, and for 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 than one request per cycle as the issue rate and the number of thread processing elements increase. However, increasing the bandwidth of the com-

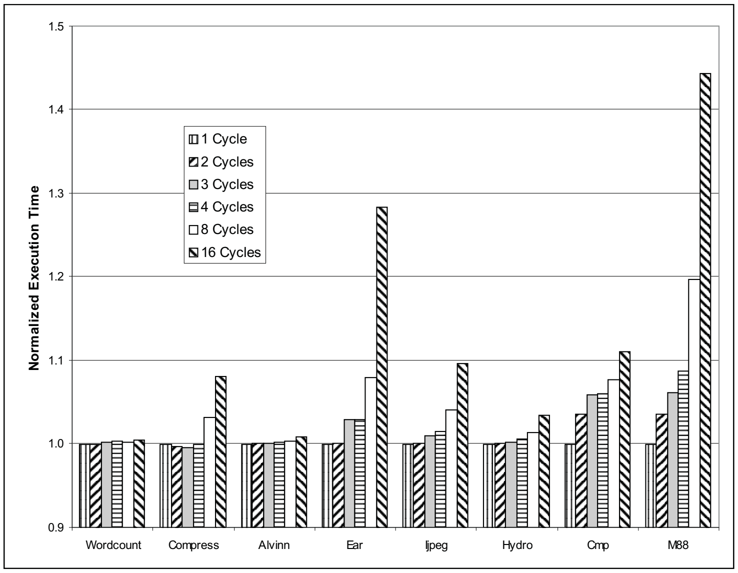

Figure 12: The sensitivity of 4-TPE, 8-issue superthreaded processor to changes in the thread initialization delay, i.e., *fork-delay.*

munication units could be expensive since it requires large send and receive buffers and multiple inter-thread connections. Therefore, it is very important to know the bandwidth requirements of the communication unit.

We ran the benchmark programs on the base superthreaded processor model with different communication bandwidth capabilities ranging from 1 request per cycle to 4 requests per cycle with the inter-thread communication latency set to 1 cycle and the number of memory buffer ports set to 4. These parameters are chosen so that the communication bandwidth is fully exercised. We see in Figure 13 that, for most of the programs, the performance improves a noticeable amount as the communication bandwidth increases from 1 request per cycle to 2 requests per cycle. However, except for *cmp* and *ijpeg*, the performance improvement tends to saturate beyond 3 and 4 requests per cycle. We conclude that a communication bandwidth of 2 requests per cycle is generally sufficient for a 4-TPE, 8-issue superthreaded processor when executing these benchmarks.

5.3 Memory Buffer Bandwidth

The memory buffer has inputs coming from the local communication unit and the load-store unit, and outputs going to the computation unit, the write-back unit. and the down-stream communication unit. In the simulator, all of the local physical functional units share the same memory buffer ports for either input or output. Among the tested programs, *Compress, Hydro2d*, and *M88Ksim* have very frequent memory

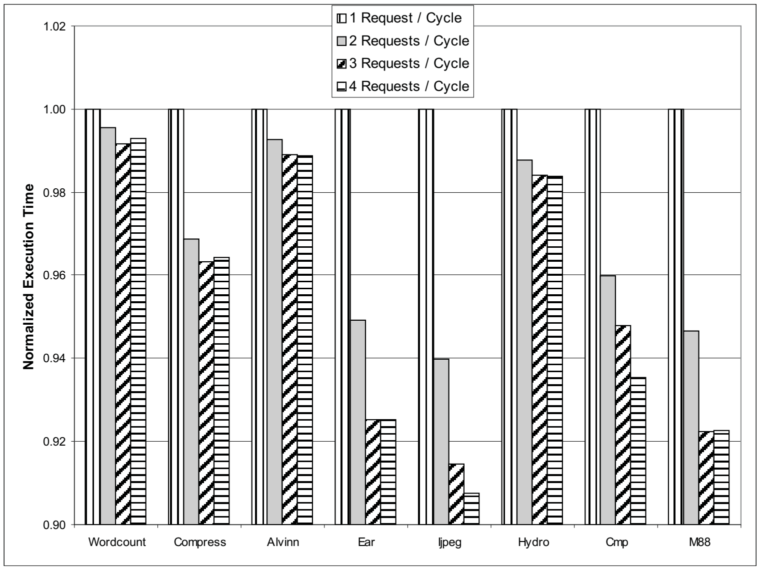

Figure 13: The sensitivity of a 4-TPE, 8-issue superthreaded processor to changes in the communication bandwidth (requests per cycle).

accesses. For *Compress*, in particular, over 60% of the instructions are load or store instructions. Since the computation unit needs to check the memory buffer for every load or store instruction it executes, the number of memory buffer ports directly affects the critical path of a program's execution. We experiment with varying the number of memory buffer ports ranging from 1 to 4 per thread processing element. The communication bandwidth is set to 4 requests per cycle, and the communication latency is 1 cycle.

We can see in Figure 14 that the performance of the superthreaded processor improves considerably when we increase the number of ports from 1 to 2. It remains relatively stable when we further increase the number of ports to 3 and 4, however. Hence, it appears that 2 memory buffer ports per thread processing element provide the best price/performance ratio.

5.4 Communication Latency

The communication latency refers to the time required for the messages to travel from the output buffer of the parent thread's memory buffer to the input buffer of the child thread's communication unit. Although the superthreaded processor is a tightly-coupled architecture, the line-delay between thread processing elements for message passing could still be significant when the messages are large. On the other hand, the hardware cost could be reduced if this latency is not a performance bottleneck. This communication latency also directly affects the total time needed for the parent thread to forward the necessary memory buffer entries to

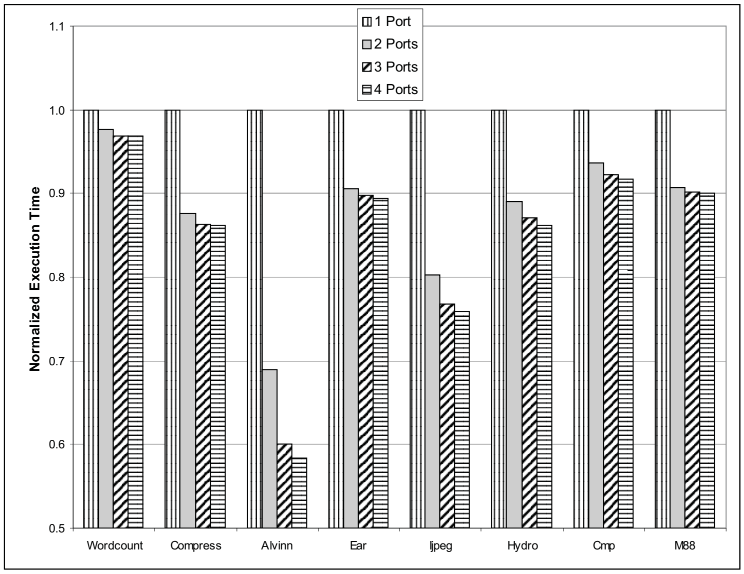

Figure 14: The sensitivity of a 4-TPE, 8-issue superthreaded processor to changes in the number of memory buffer ports.

its successor thread.

Figure 15 shows the performance impact of varying this communication latency. We see that, for some of the programs, the superthreaded processor is rather sensitive to the increase in this delay. Performance degrades quickly for *cmp, m88ksim, ear, and ijpeg*. For *wordcount, alvinn* and *hydro2d*, however, this latency increase slows down the programs by at most 2 percent since these programs do not communicate very often. Although *compress* communicates quite frequently between threads, most of the communication is between squashed threads, which removes the communication delay from the critical path. Hence, the penalty of the increased communication latency is not severe for this program. Comparing these results to Figure 12, we see that the programs that are sensitive to communication latency are also sensitive to the *fork-delay*. Programs that are sensitive to *fork-delay* spawn more useful threads, and spawning more threads usually implies more communication.

5.5 Data Cache Delays

All thread units share a single level-1 data cache, and a combined level-2 cache. Although the memory buffer supplies some of the data required by each thread unit, many memory references are still directed to the shared cache hierarchy. Hence, the latency of the shared cache could be the potential bottleneck for the performance of the superthreaded processor. On the other hand, the superthreaded processor saves execution

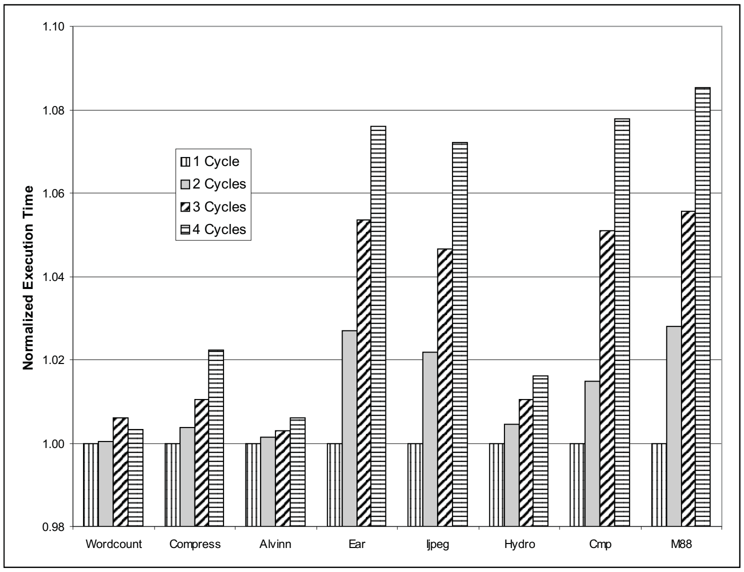

Figure 15: The sensitivity of a 4-TPE, 8-issue superthreaded processor to changes in the communication latency (in cycles) between threads.

time by overlapping the memory accesses from different thread units. As a result, this architecture may not be overly sensitive to this latency compared to a single-threaded processor. We study this issue by varying the level-1 cache hit latency from 1 to 5 cycles, and the level-2 cache hit latency from 6 to 24 cycles. Figures 16 and 17 show the sensitivity of the performance to these parameters.

All programs except *cmp* are very sensitive to the level-1 cache hit latency with *Ijpeg*'s performance degrading 28% when this latency is 5. Other programs' performance degrades between 13% to 26%. For *Cmp*, the performance actually improves when the level-1 cache hit-latency is increased from 1 cylce to 2 cycles, and then to 3 cycles. This happens since *cmp* is a very short program, and the longer level-1 cache hit latency is compensated by fewer level-2 cache misses. For the other programs, the relationship of the performance degradation and the level-1 cache hit latency is close to linear since the number of memory accesses for a program is constant when the hardware configuration and input data are the same.

When the level-2 cache's latency changes from 6 to 24 cycles, only *compress*, *hydro2d* and *m88ksim* show high sensitivity. *Compress* is a program with relatively high frequency of memory operations (over 60% of its executed instructions are loads or stores), and *hydro2d* is an array-access-based floating-point program. *M88ksim* also has a relatively high number of memory accesses since it simulates the operation of a microprocessor with its own memory hierarchy. Other programs have very low level-2 cache miss rates with this 2MB level-2 cache, and so are fairly insensitive to the level-2 cache hit latency.

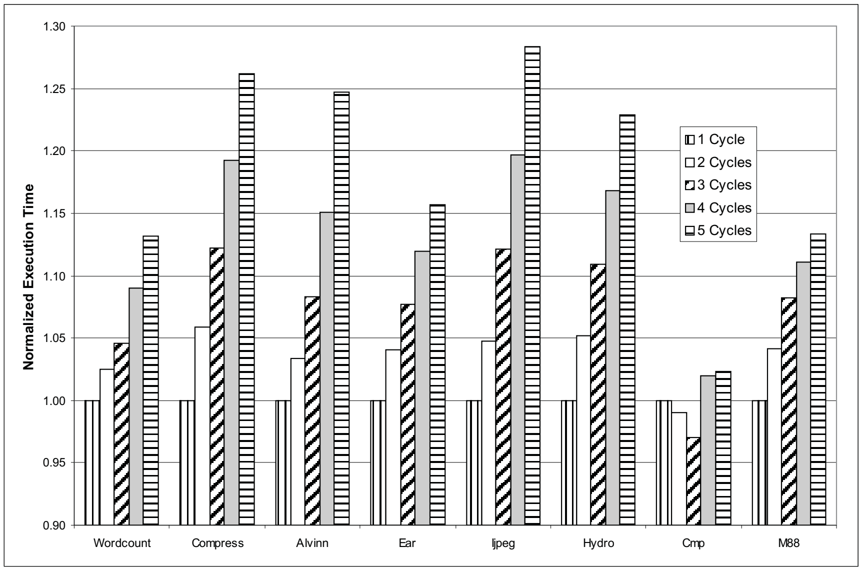

Figure 16: The sensitivity of a 4-TPE, 8-issue superthreaded processor to changes in level-1 cache hit latency (cycles).

5.6 Data Cache Bandwidth Requirement

As the total issue rate increases in the superthreaded architectural models, a higher data cache bandwidth is required to support multiple loads and stores per cycle. Techniques to improve cache bandwidth include non-blocking, multiporting, and interleaved multibanking [15]. Non-blocking improves the cache performance by reducing the bandwidth degradation due to misses. Multiporting (duplicating ports) and multibanking caches can serve multiple requests per cycle. Generally, a multiport cache is more effective than a multibank cache because it has less port contention. However, it is much more expensive to implement than a multibank cache. Previous results have shown that a 4-way interleaved cache with two read ports and one write port would provide sufficient bandwidth in the superthreaded processor for the benchmark programs studied here [16].

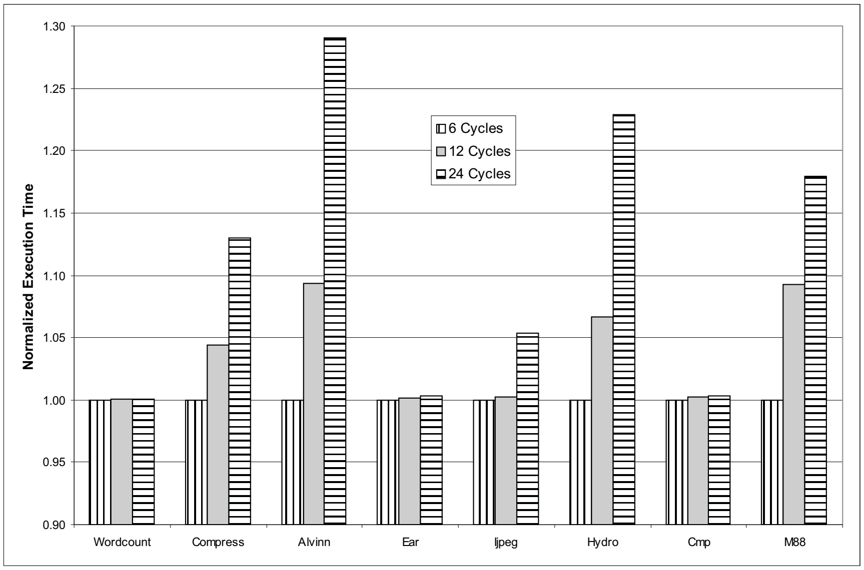

Figure 17: The sensitivity of a 4-TPE, 8-issue superthreaded processor to changes in level-2 cache hit latency (cycles).

6 Related Work

The similarity between the superthreaded architecture and existing small-scale shared-memory multiprocessors composed of superscalar processors allows the superthreaded architecture to leverage existing research in parallel hardware and compiler technologies. The superthreaded architectue provides extra features to avoid many of the shortcomings that prevent existing multiprocessors from becoming tomorrow's general-purpose computing engines. One key difference of the superthreaded architecture compared to a multiprocessor is that the superthreaded architecture supports thread-level control speculation with runtime data dependence checking. Another important difference is that communication between threads in the superthreaded architecture does not need to go through the memory, which is necessary in a traditional multiprocessor. This difference reduces the communication overhead of the superthreaded architecture allowing it to exploit lighter-weight threads (i.e. finer-grained parallelism) than is possible in a multiprocessor. Also, since the threads are more tightly coupled than in a multiprocessor, the superthreaded architecture requires a less complicated interconnection network. Furthermore, since the threads share the same data cache, the cache coherence problem is eliminated. This tighter coupling allows for more efficient execution of general-purpose application programs written in languages such as C and C++, in addition to numerical scientific applications written in Fortran, compared to traditional shared-memory multiprocessor systems.

To increase the instruction issue rate of a "single-processor" architecture, several other multithreaded architectures have also been proposed, including the *XIMD* [23], the *Elementary Multithreading* architecture [7], the *M-machine* [5], the *Simultaneous Multithreading (SMT)* architecture [19], the *Multiscalar* [14], and the *SPSM* [4]. Among them, models such as *Simultaneous Multithreading* and *SPSM* allow only 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 *XIMD*, *Elementary Multithreading*, and *M-machine* 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) are guaranteed 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.

To enhance the thread-level parallelism that can be exploited, both the *SPSM* [4] and the *Multiscalar* [14] support speculation on control and data dependences between threads. With support for control speculation, a thread that is control dependent on conditions determined by earlier threads can be initiated and speculatively executed before the results of the condition are available. 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* between operations and to execute load instructions before potentially dependent store instructions from earlier threads are executed. If a data dependence violation is detected at run-time, the thread that violates the data dependence must be squashed. Hardware support for data speculation can be very expensive, however, since all load and store addresses from all active threads must be saved for run-time detection of data dependence violations [6, 14].

The superthreaded architecture also provides hardware support for thread-level control speculation. However, it does not *speculate* on data dependences, but instead performs runtime data dependence *checking*. As a result, it needs to keep only the *store* addresses in each thread processing unit's memory buffer, but no *load* addresses. Since there are usually far more *loads* than *stores* in most application programs, not needing to save *load* addresses allows the size of the memory buffers in the superthreaded architecture to be much smaller than in the *Multiscalar*, for instance.

7 Conclusions

Exploiting more parallelism from programs is the key to improve 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 multithreaded architectural models, which can exploit loop-level parallelism efficiently, have a great potential to be used in future microprocessor designs. However, many loops in general application programs often have control dependences and data dependences between their loop iterations that can prevent them from being executed concurrently. Therefore, it is very important for a concurrent multithreaded architecture to provide mechanisms to efficiently tolerate both control dependences and data dependences between threads.

In this paper, we present the *superthreaded* architectural model for exploiting thread-level parallelism with run-time support for inter-thread data dependence checking and thread-level control speculation. The superthreaded model uses a thread pipelining mechanism to enforce data dependences between threads. The combination of *thread pipelining* with the target store address and data forwarding schemes allows each thread to perform run-time inter-thread data dependence checking and synchronization within 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, the superthreaded architecture relies on the compiler to extract thread-level parallelism and to generate efficient superthreaded code. In this paper, we also studied some compiler techniques for the superthreaded architecture. We found that many existing program analysis and transformation techniques developed for multiprocessor architectures 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.

We evaluated the performance of the superthreaded architecture through a detailed execution-driven superthreaded processor simulator using eight benchmark programs written in or transformed to C. The simulation results show that the superthreaded architectural model can further improve the performance of a single-threaded superscalar processor for the tested programs.

References

[1] Doug Burger, Todd Austin, and Steve Bennett. Evaluating Future Microprocessors: The Simple Scalar Tool Set. Technical Report 1342, Computer Science Department, University of Wisconsin-Madison.

[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] 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.

[4] 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.

[5] 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 Int'l Symposium on MicroArchitecture* (MICRO), Ann Arbor, Nov., 1995, pages 146-156.

[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] 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 19 Int'l Symposium on Computer Architecture* (ISCA), Gold Coast, Australia, May 1992, pages 136-145.

[8] Jian Huang and David J. Lilja. An Efficient Strategy for Developing a Simulator for a Novel Concurrent Multithreaded Processor. In *Proceedings of the 6th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* , pages 185–191, July 19–24, 1998.

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

[10] 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.

[11] 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.

[12] J. Smith and S. Vajapeyam. Trace Processors: Moving to Fourth Generation Microarchitectures. In *IEEE Computer*, Sep. 1997, volume 30, number 9, Pages 68 - 74.

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

[14] 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.

[15] 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.

[16] Jenn-Yuan Tsai, Zhenzhen Jiang, Eric Ness, and Pen-Chung Yew. Performance Study of a Concurrent Multithreaded Processor. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture* (HPCA-4), pages 24 - 33, Jan. 31 - Feb. 4, 1998.

[17] 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.

[18] 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.

[19] Dean M. Tullsen and Susan J. Eggers and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. IN *Proceedings of the 22nd Int'l Symposium on Computer Architecture* (ISCA), 1995, pages 392-403.

[20] 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.

[21] 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.

[22] Michael Wolfe. *High-Performance Compilers for Parallel Computing*. Addison-Wesley Publishers, Jan., 1996.

[23] Andrew Wolfe and John P. Shen. A Variable Instruction Stream Extension to the VLIW Architecture. In *Proceedings of the 4th Int'l Symposium on Architecture Support for Programming Languages and Operating Systems*, Santa Clara, April 1991, pages 2-14.