An Efficient Strategy for Developing a Simulator for a Novel Concurrent Multithreaded Processor Architecture

Jian Huang
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455 U.S.A.
huangj@cs.umn.edu

David J. Lilja
Department of Electrical and Computer Engineering
University of Minnesota
Minneapolis, MN 55455 U.S.A.
lilja@ece.umn.edu

December 11, 1997

Abstract

In developing a simulator for a new processor architecture, it often is not clear whether it is more efficient to write a new simulator from scratch or whether it is better to try and modify an existing simulator. Writing a new simulator has the advantages of needing to understand only the new processor architecture itself and being the "owner" of the final code. However, it also forces the processor architect to develop or adapt all of the related software tools, such as compilers, assemblers, and so forth. On the other hand, modifying an existing simulator and related tools can be time-consuming and error-prone since the simulator writer must learn the details of someone else's code which is, typically, not well documented. We describe the *SImulator for Multithreaded Computer Architectures* (SIMCA) that was developed with the primary goal of obtaining a functional simulator as quickly as possible to begin evaluating the superthreaded architecture being developed at the University of Minnesota [2, 3]. The performance of the simulator itself was important, but secondary. We achieved our goal using a technique we call *process-pipelining* that exploits the unique features of this new architecture to hide the details of the underlying simulator, which is based on the SimpleScalar tool set [4]. This approach allowed us to quickly produce a functional simulator whose performance was only a factor of 3.8 to 4.9 times slower than the base SimpleScalar simulator. Because of the modularity of this approach, some simple optimizations allowed us to improve its performance by a factor of two with little effort.

Keywords: execution-driven simulation, processor design, multithreading, superthreading, system modeling.

1 Introduction

Computer system performance has experienced dramatic increases over the past several decades. Much of this improvement is due to the introduction and implementation of such previously exotic architectural techniques as out-of-order execution, Very Long Instruction Word (VLIW) architectures, concurrent multithreading, data and control speculation, and the relaxation of memory consistency models. Since these features can substantially increase the complexity of a processor, the study of these novel architectures is based more and more on detailed simulations. Previous research [9] has shown that execution-driven simulation tools, such as the SimpleScalar tool set [4], FastMIPS [7], MINT [5], and CacheMire [6], provide better evaluations for these new designs than traditional trace-driven simulations. Unfortunately, these simulators can be quite complex to develop in themselves.

This complexity of both simulator development and processor architecture leads to an interesting trade-off between adapting an existing simulation tool set to current needs, and developing a completely new simulator from scratch. It would seem to be more efficient and economical to extend one of these existing tool sets to build a simulator for a newly proposed architecture. This would save the time to develop the auxiliary components, such as the assembler, loader, instruction decoder, basic execution unit, and so forth. However, adapting an existing tool set to fit a proposed architecture remains a nontrivial task due to the difficulties of understanding someone else's code, and trying to deal with the problems introduced by the extensive use of global variables and macros, in addition to debugging the new code needed to simulate the new architecture. In this paper, we describe our strategy for developing a simulator for the superthreaded processor architecture [2, 3] that takes advantage of the structure of this proposed architecture to hide many of the nasty details of the the selected tool set.

In particular, we developed the *SImulator for Multithreaded Computer Architectures* (SIMCA) on top of the SimpleScalar Tool set. The superthreaded architecture uses the multithreading concept to execute multiple independent program tasks concurrently. Thus, it requires the simultaneous simulation of multiple thread processing units. In this new processor, thread processing units are dynamically activated and deactivated. Data dependences are resolved at runtime, while control dependences can be executed speculatively. This overlapped execution of multiple thread processing units relies on the *thread-pipelining* execution model. Our simulator mirrors this novel execution model by using a unique process to simulate each thread unit. We call this technique *process-pipelining* and find that, in addition to providing good simulation performance, it also allowed us to shorten the time required to develop the simulator compared to the time it would have required to develop a simulator completely from scratch.

Instead of trying to extend the basic single-threaded SimpleScalar simulator to incorporate multithreading, SIMCA forks a new process for every newly activated thread-unit. This approach eliminates all of the code that would have been needed to explicitly copy all of the simulator state variables, and, more importantly for reducing the simulator development time, it eliminates the need for the programmer to identify the variables that need copying. It also eliminates the need for the programmer to identify and understand all of the global variables. Instead, the programmer's efforts can focus on adding the code needed to simulate the new processor architecture without fully understanding the base code or worrying about errors triggered by unintended modifications of global variables. One disadvantage of this approach is the large amount of process creation and termination and synchronization overhead. To compensate for this overhead, we also developed a scheme that allows the processes to synchronize every simulated processor cycle without using expensive semaphores.

We have used SIMCA to successfully simulate several of the SPEC benchmark programs. These preliminary experiments showed that it can execute approximately 3,000 instructions per cycle on an SGI Challenge Cluster with MIPS R10000 processors compared to the simulation rate of approximately 11,000 instructions per second for the base SimpleScalar simulator.

In the remainder of this paper, Section 2 briefly introduces the thread-pipelining model of the superthreaded architecture. Sections 3 and 4 then describe the details of the simulator, while Section 5 evaluates its performance. Section 6 summarizes and concludes the paper.

2 Background: The Superthreaded Architecture

Similar to other concurrent multiple-threaded architectures, the superthreaded architecture [2, 3] exploits task-level parallelism using multiple threads of control. The superthreaded processor typically consists of a number of thread processing units with shared instruction and data caches, as shown in Figure 1. Each thread processing unit has its own program counter and data path logic so that the entire processor can fetch and execute instructions from several different program locations simultaneously.

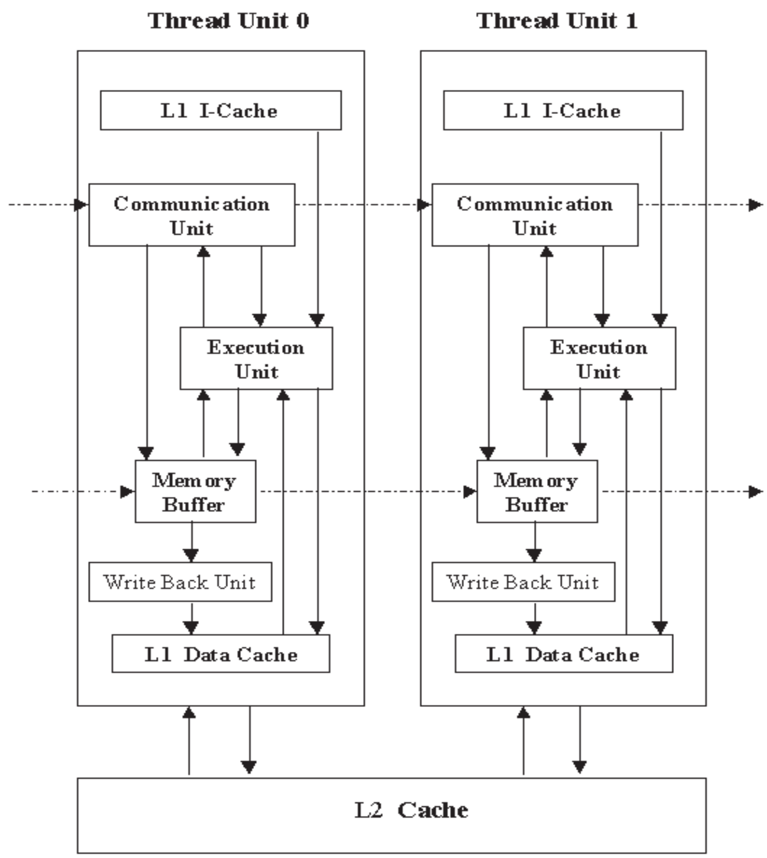

Figure 1: The organization of the superthreaded processor

2.1 Thread Partitioning

The control flow graph of a program is statically partitioned by the compiler into individual threads. Each thread performs a sequence of computations using a set of data that has no, or only a small number of, dependences with other threads. While the compiler can adjust the granularity of the threads, a thread is generally one to several iterations of a loop.

The code segment shown in Figure 2 performs a depth-first traversal of a nested list pointed to by the variable `this`. It marks every node it traverses reversing the direction of the links along the path. This loop is one of the most frequently executed loops in the SPECint95 benchmark *130.li*, making it an important representative of the types of program structures that can be executed on the superthreaded architecture. There is only one explicit data dependence between neighboring iterations in this loop that is caused by the recurrence variable `this`. The compiler would partition this loop so that each thread corresponds to a single iteration of the loop.

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

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

2.2 The Thread Pipelining Execution Model

The superthreaded architecture uses the *thread pipelining* execution model, shown in Figure 3, to enforce data dependences between concurrent threads. Unlike traditional instruction pipelining, in which instruction sequencing and data dependence checking and forwarding are performed automatically by hardware, the superthreaded architecture performs thread initiation and data forwarding through explicit thread management and communication instructions.

2.2.1 Thread Ordering

Execution of any program in this architecture starts from the entry thread. This thread can then fork a successor thread on another thread unit. We call this fork operation an `st_fork` to distinguish it from the standard Unix system call fork. The successor thread can further `st_fork` its own successor thread. This st_forking process continues until all thread units are busy. In the example program of Figure 2, for instance, a loop iteration would use `st_fork` to spawn a subsequent iteration after it updates the recurrence variable `this`.

The oldest thread from the original program order is referred to as the *head thread*. All of the other threads derived from it are called *successor threads*. When the head thread completes its computation, it releases its thread execution unit making its successor thread the new head thread. Thread completion must follow the original program order to ensure program correctness.

A thread can `st_fork` its successor thread with or without control speculation. Before initiating a successor thread without control speculation, it must ensure that all of the control dependences of its successor thread have been satisfied. If a thread initiates a successor thread speculatively, however, it must later verify all of the speculated control dependences. If any of the speculated dependences turn out to be false, the thread must issue a command to kill its immediate successor thread and all of its subsequent successor threads.

The execution of each individual thread is partitioned into several stages. Figure 3 also shows the thread pipelining stages of an individual thread in addition to the relationship between concurrent threads. The function of each of these stages is described in the following subsections.

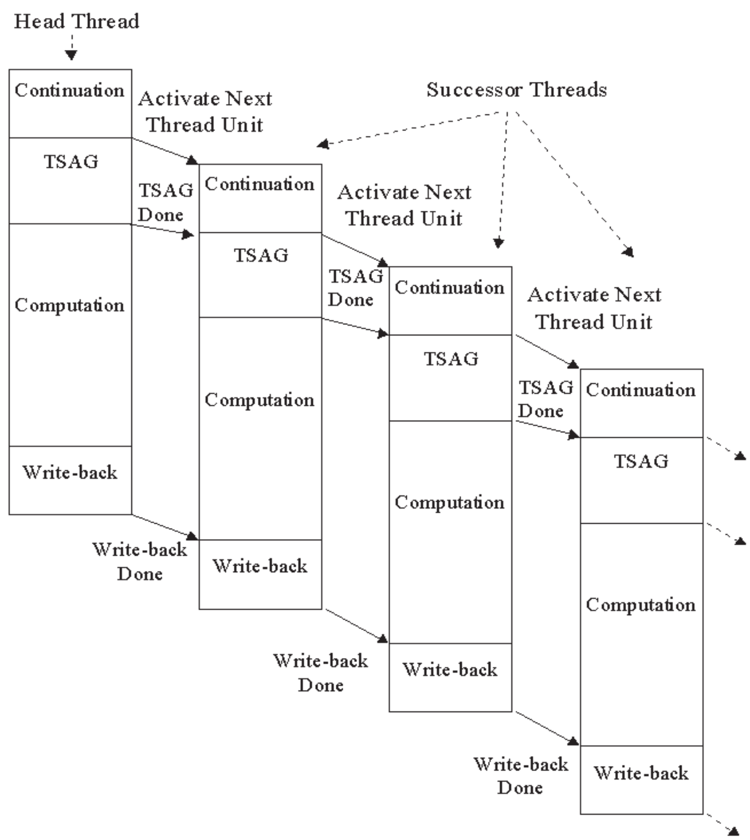

Figure 3: The pipelined execution of contiguous threads

2.2.2 Continuation Stage

Each thread begins with the *continuation* stage. The major function of this stage is to compute the values of loop recurrence variables, such as loop index variables, that are needed to fork the next thread. These variables are called *continuation variables*. The values of these continuation variables are forwarded to the next thread processing unit before the subsequent thread is activated. For example, the continuation variables in a `for` loop, such as `i = i+1` or `p = p->next`, must be computed and forwarded to the next thread processing unit before the next iteration can begin executing. The continuation stage of each thread terminates with an `st_fork` instruction which initiates the next thread. As shown in Figure 3, the continuation stages of two adjacent threads will never overlap. This serialization is necessary since the computation of the continuation variables for a subsequent thread depends on the results produced in the continuation stage of the predecessor thread.

2.2.3 Target-Store-Address Generation Stage

In addition to the continuation variable values, a thread may need other values produced by predecessor threads. To control hardware complexity, the superthreaded processor does not speculate on these inter-thread data dependences. Instead, run-time data dependence checking is implemented using the *target store (TS)* operation to notify successor threads of potentially dependent memory locations. The addresses of these target stores are calculated in the target-store-address generation (TSAG) stage. These target store addresses are then forwarded to the memory buffers of all of a thread's successor threads, as shown in Figure 1.

After all target store addresses have been forwarded, a thread will send a `tsag_done` flag to its successor thread. Prior to receiving this flag, a successor thread can perform only computation that is not dependent on any target store variables of its active predecessor threads, except for the previously computed continuation variables. Receiving this flag allows the successor thread to begin executing any computations that may be dependent on computations from its predecessor threads.

2.2.4 Computation Stage

The computation stage performs the main work of a thread. If the address of a load operation performed during this stage matches that of a target store entry in a thread's memory buffer, the thread will read the data f it is valid. If it is not valid, the thread must wait until the data is forwarded from an earlier thread. If the value of a target store variable is computed during this stage, though, the thread must forward both the address and the data to the memory buffer of all its successor threads. If a target store is executed in only one of the paths of a conditional branch, a null value needs to be forwarded to successor threads if that branch turns out to be not taken during the execution of the thread. If this null value is not forwarded, all successor threads will be suspended indefinitely waiting for a value that will never be computed. The computation stage of a thread is terminated by the execution of a `stop` operation, unless it is killed by a predecessor thread using the `abort_future` instruction.

2.2.5 Write-Back Stage

Data from all store operations need to be temporarily stored in a thread processing unit's memory buffer to avoid changing the state of the main memory by a speculative thread that is later aborted by one of its predecessor threads. Thus, at the completion of the computation stage, all control dependences of speculative successor threads are checked. If these dependences are found to have been speculated correctly, the thread will terminate by writing all of the data in its memory buffer back to the main memory. This data includes the results of both target stores and regular non-target stores. To correctly follow the semantics of the application program, threads must perform their write-back operations in their original program order. The `wb_done` flag is used to signal the end of a thread's write-back stage to thereby inform its successor thread that it may begin its write-back operation.

2.3 A Thread-Pipelining Example

The example program shown in Figure 2 would be compiled into the corresponding superthreaded code shown in Figure 4. In this example, the compiler partitions the application program so that a thread

```
/* Continuation stage */
L100:
   register *this_tmp, cond1, cond2, *addr1, *addr2, *addr3;
   this_tmp = this;
   st_wait_tsag_done();
   if (cond1 = livecar(this))
      st_store_ts_w(&this, this->n_car);
   else if (cond2 = livecdr(this))
      st_store_ts_w(&this, this->n_cdr);
   st_fork(L100);
/* Target-store-address-generation stage */
   st_allocate_ts_w(&prev);
   addr1 = &(this_tmp->n_flags);
   allocate_ts(addr1);
   addr2 = &(this_tmp->n_car);
   st_allocate_ts_w(addr2);
   addr3 = &(this_tmp->n_cdr);
   st_allocate_ts_w(addr3);
   st_release_tsag_done();
/* Computation stage */
   if (*addr1 & MARK) {
      st_abort_future();
      goto L200;
   } else {
      *addr1 |= MARK;
      if (cond1) {
         st_release_ts_w(addr3);
         st_store_ts_w(addr1, *addr1 | LEFT);
         tmp = prev; /* from the predecessor thread */
         st_store_ts_w(&prev,this_tmp); /* to the successor thread */
         st_store_ts_w(addr2, tmp);
      }
      else if (cond2) {
         st_release_ts_w(addr2);
         st_store_ts_w(addr1, *addr1 &  LEFT);
         tmp = prev; /* from the predecessor thread */
         st_store_ts_w(&prev,this_tmp); /* to the successor thread */
         st_store_ts_w(addr3, tmp);
      }
      else {
         st_abort_future();
         goto L200;
      }
   }
   st_stop();
/* Write-back stage: performed by hardware automatically */
L200:
   st_wait_wb_done();
```

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

corresponds to a loop iteration. The continuation variable `this`, which is computed in each thread's continuation stage, is needed, either directly or indirectly, in each thread's successor thread. To complicate the situation, however, the value of `this` depends on the outcome of a conditional statement in the loop body. So that the continuation stage can compute the necessary values to forward as early as possible, the conditional statement and its related operations are also moved into the continuation stage. The value of `this` also depends on the memory location pointed to by `this`. Due to the pointer data structure, it is impossible to determine at compile-time whether the memory location will ever be modified by a predecessor thread. Consequently, it is necessary for the compiler to insert a `wait_tsag_done` instruction before the computation of `this` to force threads to check for the data dependence at run-time. After this recurrence value is forwarded to the next thread processing unit's memory buffer using a `store_ts` instruction, the next thread is speculatively initiated with an `st_fork` instruction.

The TSAG stage computes the four required target store addresses: `&prev`, `addr1`, `addr2` and `addr3`. These addresses are forwarded to the successor threads with the `allocate_ts` instruction. These addresses are then saved in registers for later use. The `tsag_done` instruction terminates the TSAG stage. Here the prefix `st_` marks a function call as our architecture-specific instruction, and the suffix `_w` indicates that the variable at that address takes one word of space.

The computation stage performs the last write to each target store address, such as the write to `prev`, using the `store_ts` instruction. This instruction forwards the final value computed for that address to successor threads. If `cond1` is true, target store address `addr3` will not be written. To prevent successor threads that are waiting for `addr3` to be written from suspending indefinitely, a `release_ts` instruction is executed. Receiving the signal from the `release_ts` instruction allows these successor threads to proceed when `cond1` is true.

At the completion of the computation stage, a thread will either terminate by executing a `stop` instruction, or one of the break conditions will cause it to exit the loop. When a thread encounters the `stop` instruction, it will enter the write-back stage automatically, after which it releases its thread processing unit. However, threads that detect a loop termination condition will execute an `abort_future` instruction that kills all of its speculative successor threads. After the head thread exits the loop, it performs its write-back operation and executes a `wait_wb_done` instruction before initiating the next portion of the program.

3 Organization of SIMCA

The SimpleScalar tool set [4] comes with its own versions of *gcc*, *gas*, and *gld*, the GNU implementations of a C compiler, an assembler, and a loader. The tool kit also comes with its own versions of the standard UNIX libraries. We call these SimpleScalar modified tools *ss-gcc*, *ss-gas*, and *ss-gld.* To simulate an application program with the SimpleScalar simulator, the application source code is fed to *ss-gcc*, *ss-gas*, and finally to *ss-gld* to produce the target binary code that can then be simulated. Since SIMCA is implemented on top of the SimpleScalar tool set, it uses the same scheme to prepare the binary object code for simulation, with the addition of a few extra steps to customize the code with the superthreaded architecture instructions.

While we are developing a compiler to automatically transform generic C or FORTRAN source code into the superthreaded instruction set, we currently must insert superthreaded processor instructions by hand. These special instructions are added in the application source code by inserting special function calls for each instruction, as shown in 4, where the parameters match the register fields of each of the superthreaded instructions. This hand-compiled source code is passed to the *ss-gcc* compiler to produce MIPS assembly code, in which each superthreaded instruction will appear as `jal st_function_name`.

Next, a special parser we constructed scans the assembly code output of *ss-gcc* and performs a textual replacement of each `jal st_function_name` that we inserted for the new superthreaded processor instructions. We also modified *ss-gas* to recognize the new instructions and generate the appropriate object code. Finally, *ss-gld* is used to produce the superthreaded binary object module to be executed on the SIMCA simulator. This compilation and assembly process is summarized in Figure 5.

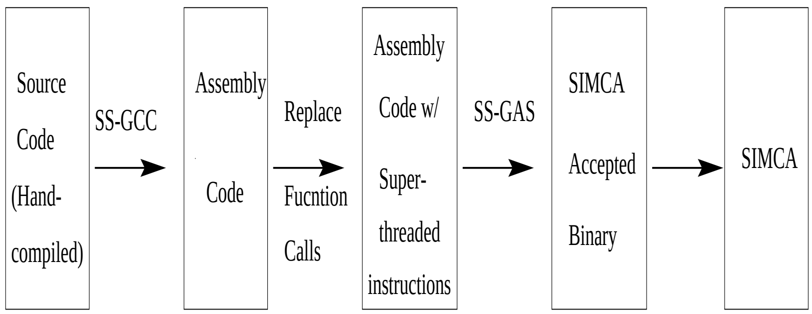

Figure 5: The Organization of SIMCA

4 Implementation of SIMCA

Our primary goal in building this simulator was to obtain a functional version as quickly as possible while not worrying about its actual performance. After it was proven to be functionality correct, we could improve its performance. Since the SimpleScalar tool set is written in the C programming language, it inevitably contains a large number of global variables. Consequently, it is very difficult not to introduce errors into the simulator when modifying it without completely understanding its internal details. Developing a sufficient level of understanding, of course, typically requires a great deal of time. To address this problem, we developed what we call the *process-pipelining* scheme.

4.1 Process-Pipelining

Mimicking the thread pipelining model described in Section 2.2, we implemented the *process-pipelining* scheme to simulate the activation, deactivation and interlocking of the thread units. A head process is created to correspond to the head thread. When any thread `st_forks` a successor thread, the simulator simply forks a new process using the standard UNIX `fork` command, as shown in Figure 6. This `fork` creates an entirely new copy of the simulator, except that it also automatically copies all of the values of all of the global variables. This new simulator invocation then simulates the execution of the successor thread. Using this UNIX `fork` to simulate the `st_fork` creation of successor threads eliminates the need for the programmer building the simulator to understand all of the details of the SimpleScalar tool set's global variables and the internal operation of the simulator itself.

Each process simulates the continuation, TSAG, computation, and write-back stages of the thread pipelining model. When the successor thread is waiting for data from its predecessor, the successor process simply spins on the address of the data. When a thread finishes its write-back stage and is ready to release the thread processing unit, the process simulating that thread needs only to exit. Its immediate successor process then becomes the new head process. Note that processes are initiated and retired in exactly the same order in which thread units would be initiated and retired. Data also flows in the same direction as the thread order. When a misspeculation causes a thread to be aborted, the process simulating that thread is killed by a signal from its predecessor process. Its successor processes also will be killed in the same manner as the threads being simulated would be aborted.

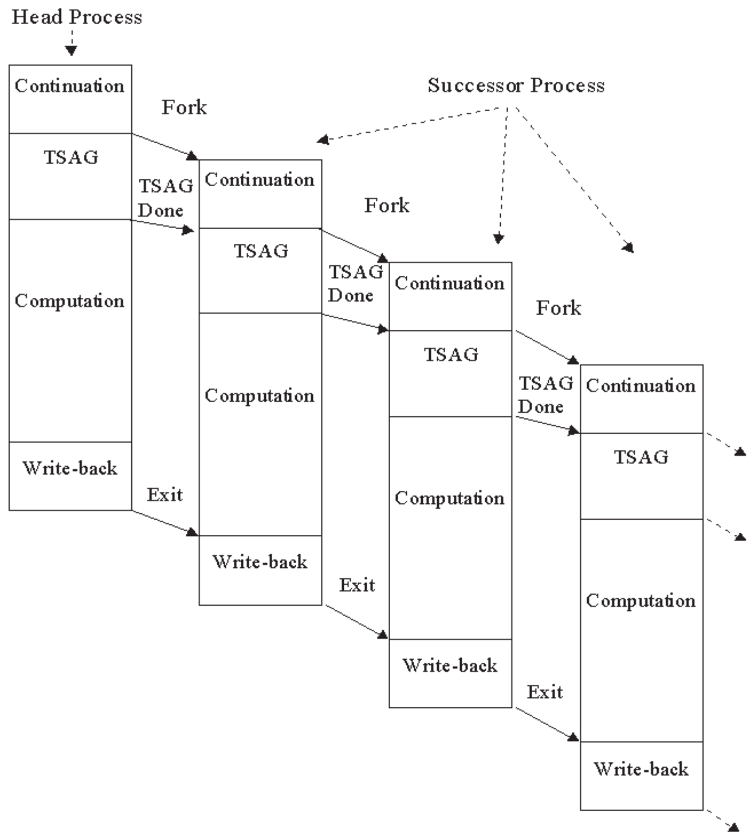

Figure 6: Process Pipelining

In a typical implementation of a superthreaded processor, as was shown in Figure 1, the thread units are connected using a unidirectional wrap-around ring. This ring structure simplifies the communication hardware compared to more complex interconnection networks. Each thread unit has a direct communication link to the next thread unit along the ring. Because the ring is unidirectional, commands and data are always forwarded by a thread to its successor threads in the same direction.

To allow easy simulation of the communication units and the memory buffers, these simulated structures are allocated in shared memory. When the simulator begins execution, it first allocates a fixed number of shared memory segments for the communication units and memory buffers. When a process is forked, it searches for available shared memory segments to attach and thereby claim for its communication unit and memory buffer. The process identifies the specific segments it has claimed by tagging them with its process ID number.

The parent-child relationship among the processes mimics the predecessor-successor order among the threads. As a result, the software links between simulated communication units in the shared memory segments are similar to the communication ring that interconnects the thread units. One key difference, however, is that this simulated ring expands and contracts dynamically as processes are created and destroyed.

The SimpleScalar tool set simulates a cache, a translation look-aside buffer (TLB), a branch-prediction table, and the main memory. Since these components are shared among all of the thread units in the superthreaded processor, they must be allocated to shared memory in the simulator. However, all accesses to the cache, the TLB, and the branch-prediction table must be controlled to prevent race conditions as the different processes read and write common entries. Semaphores are used to provide mutually exclusive access by the processes to these shared data structures. Mutual exclusion is not needed for the simulated main memory, though, since only the head process can write to the main memory.

4.2 Process Synchronization

Since the different thread processing units are tightly coupled in that they share a common data cache, can quickly forward addresses and data values to successor threads, and all operate from a common clock, accurately simulating the timing between thread processing units is of critical importance. To simulate the common clock, a global clock is used to control the individual simulation events. The use of this global clock then requires each of the processes simulating the thread execution units to synchronize on every simulated clock cycle. The simplest implementation for this synchronization would be to use semaphores. However, since a semaphore operation involves the operating system kernel, it takes a relatively long time, which, since synchronization is required on every simulated clock cycle, would severely slow down the simulator. Instead, we developed a synchronization protocol that allows us to synchronize without OS intervention.

To implement this synchronization operation we allocate a `cycle_completion` variable in a shared memory segment. Each process is assigned a specific bit of this variable that it sets to 1 to indicate that it has completed a simulated clock cycle. It then reads the variable and counts the number of bits that are set to 1. If this value equals the number of active thread units, all processes have completed this simulated clock cycle. One or more of the processes will then clear all of the bits. Every process continuously checks the variable waiting for it to be cleared. Once the bits are cleared, all of the processes can continue with next simulated cycle.

5 Hardware Component Simulation

Figure 1 shows the four major components of a thread processing unit – the execution unit, the communication unit, the memory buffer, and the write-back unit. In this section, we describe how each of these subunits are simulated.

5.1 Execution Unit

The execution unit is similar to a conventional single-threaded superscalar microprocessor. It has a single program counter, instruction fetch and issue logic, a complete set of private registers, and some number of functional units. To achieve information hiding and code reuse in the simulation of the execution unit, SIMCA uses the SimpleScalar processor simulator for the execution units inside each thread processing unit. The processor simulator provided with the SimpleScalar tool set provides out-of-order instruction execution on up to 4 integer ALUs, 1 integer multiply-divide unit, 4 floating-point adders, and 1 floating-point multiply-divide unit.

5.2 Communication Unit

The communication unit in each thread processing unit has an input link from, and an output link to, the communication units of its two neighboring thread processing units. This subunit receives and processes commands and data from the execution unit and the upstream communication unit. It also forwards commands and data to the downstream communication unit. The data structures for this subunit are allocated to a shared memory segment to simplify the communication among the other processes simulating the active thread processing units. The number of ports between neighboring communication units, and the number of bits per port, is parameterized to allow us to evaluate the impact of varying the inter-thread communication bandwidth.

When a successor process is forked (i.e. when a new thread processing unit is activated), the new process searches the list of shared memory segments for a free communication unit. It then inserts its process ID number in the communication unit, and starts executing from the program counter passed to it by its parent process. This parent process is automatically recorded as the predecessor thread of this process while the ID number of this new thread is recorded as the successor thread of the parent process.

The functions performed by the communication unit include:

- Recording the thread unit's corresponding process ID number, the process ID number of its successor thread process, and the index values of the resources the process is using, such as the ID numbers of the shared memory segments it has attached to itself for the communication unit and the thread's memory buffer.
- Recording the target store addresses and values from the upstream communication unit in the local memory buffer, and forwarding these values to the downstream communication unit as needed.
- Executing specific instructions that are used for inter-thread communication and thread management. These instructions are forwarded to the communication unit from the execution unit.
- Executing the synchronization instructions and the `stop` instruction forwarded from the execution unit.

5.3 Memory Buffer

The memory buffer plays a critical role in the run-time data dependence checking of the superthreaded processor. All of the target store entries generated by the current and the predecessor threads must be stored in the memory buffer. When a thread unit is activated, it must copy the necessary target-store addresses and valid data from its predecessor thread. Hence, in the simulator, the memory buffer data structures are all allocated in a shared-memory segment. The memory buffer also temporarily stores the values produced by non-target store writes when a thread is executing speculatively. When a thread is determined to be no longer speculative and executes a `stop` instruction, the values in the memory buffer are written back to the main memory.

To allow a thread processing unit to quickly search for target store addresses, the memory buffer must be organized as an associative memory structure. The degree of associativity is parameterized in the simulation to allow performance trade-off studies. As in a cache, a memory buffer is indexed by the address of the memory word being accessed. Thus, the memory buffer has both an address tag field and a data field. However, unlike a cache, each entry in the memory buffer needs the following additional fields:

- *Valid bit (V):* This bit indicates whether or not this entry is valid.
- *Alias Count (AC):* This field keeps track of the number of aliases that have the same target store address in this thread. The simulator currently implements an 8 bit value for this field, which supports up to 255 aliases for the same address.
- *Target Store Distance Vector for Address (TSDV-A):* This field is a vector of N bits that record which threads have generated a target store address corresponding to this entry. The number of bits, N, is the maximum number of thread processing units.

- *Target Store Distance Vector for Data (TSDV-D):* This field is a vector of N bits for recording which threads have released a target store data value corresponding to this entry. The number of bits, N, is again the maximum number of thread processing units.

The *TSDV-A* and *TSDV-D* fields are used to ensure that output dependences (write-after-write hazards) between threads are handled correctly. The memory buffer uses these fields to keep track of which threads have generated each target store address, and which threads have already released the data. The initial values of *TSDV-A* and *TSDV-D* are all 0. When a memory buffer entry is allocated to a target store address with a distance count of i, bit i of the corresponding *TSDV-A* field is set to 1.

5.4 Write-Back Unit

The write-back unit executes the write-back stage of the thread pipelining execution model by scanning the memory buffer and writing all the data generated by the thread back to the main memory via the shared data cache. A memory buffer entry will be written back to the data cache if it has been written by this thread processing unit, that is, if the corresponding bit of its *TSDV-D* field is 1. The write-back stage cannot begin until the thread unit becomes the head thread. In the simulator, the write-back stage of a process begins when its parent process, which corresponds to its predecessor thread, exits.

6 Evaluation

The process-pipelining mechanism used to implement this simulator allows us to use the ideas of information hiding and encapsulation even though it is written in C. The whole simulator behaves like four separate components – the communication unit, the memory buffer, the encapsulated execution unit, and the shared hardware components. The non-shared data values and simulated hardware components, together with the simulator's global data variables, are protected against corruption by the operating system when a thread is forked. This approach also eliminated the trouble of initializing the global variables since they are automatically initialized to zero. In addition, there is no need to restore any values that are changed in the global variables since they are guaranteed not to be used by any other processes.

Using these heavyweight processes to simulate such a tightly coupled architecture requires frequent synchronization and tight interlocking. The simplicity of this process-pipelining mechanism may be outweighed by poor simulation times. In addition, due to lack of good tools for debugging multithreaded application programs, debugging this type of simulator can be rather difficult. Debugging difficulties forced us to adopt rigorous testing strategies in which individual components were thoroughly tested before being integrated into the base simulator.

To evaluate the execution time performance of the simulator itself, we hand-compiled four programs from the SPEC benchmark suite. We inserted the necessary superthreading instructions at the appropriate points into the source code of *002.wc* and *056.ear* from SPEC92, *129.compress* and *132.ijpeg* from SPEC95. Since SIMCA actually executes the application programs, we we able to compare the output of the programs to the expected results to verify that the programs were simulated correctly. We then measured the total execution time of the simulator running on an 8-processor SGI Challenge system with version 6.2 of the IRIX operating system. The MIPS R10000 processors in this system run at 196 MHz and communicate with each other over a shared-bus. For *002.wc* we used a file of 9891 lines as input, for *132.ijpeg*, we tested with trivial input, and for both *056.ear* and *129.compress* we used short input.

As shown in Tables 1 and 2, the total time required to simulate these application programs ranges from 1,440 to 279,040 seconds. This corresponds to an execution rate of of 2,977 to 3,395 superthreaded processor instructions simulated per second when simulating 4 thread processing units. Compared with the execution rate of the base SimpleScalar processor, the simulation of the superthreaded processor slows

Programs	Simulation Time (sec)	Simulated Cycles	Instr per Second
wc	1,440	3,021,234	2,977
compress	2,546	4,229,776	2,999
ear	279,044	354,289,112	3,945
ijpeg	74,426	67,234,841	3,093

Table 1: Performance of SIMCA with four thread processing units.

Programs	Simulation Time	Simulated Cycles	Instr per Second
wc	367	3,387,282	11,682
compress	673	3,929,912	8,285
ear	69,011	627,400,751	15,951
ijpeg	15,189	121,022,714	15,156

Table 2: Performance of the base SimpleScalar simulator.

down a factor of approximately 3.8 to 4.9.

This slowdown occurs primarily due to two reasons. First, the time required to fork and terminate the heavyweight processes is quite high compared to the amount of work being performed in each process to simulate one invocation of a thread. The small granularity of the jobs assigned to each thread processing unit when there is more than one active is insufficient to mask the multiprocessing overhead. Thus, the simulator sees no performance benefit from the use of multiple processors.

The second important factor contributing to the slowdown of the simulator compared to the base SimpleScalar simulator is that all accesses the the cache, the TLB, and the branch prediction table must be serialized. For many application programs, it is not unusual for one out of every five or six instructions to be a load or store [8]. Furthermore, each thread processing unit is capable of issuing several instructions per cycle. If the average effective instruction issue rate is about two, there will be a load or store instruction every two to three cycles. Thus, using a straight-forward application of Amdahl's Law [1] we find that the serialization of accesses to the memory system limits the maximum possible speedup for this simulator to no more than a factor of two to five times.

7 Conclusion

The primary goal in developing this simulator was to provide as quickly as possible a platform on which to begin experimenting with the new superthreaded processor architecture. While the performance of the simulator itself was secondary, we did make efforts to design the simulator to be easily modified to allow us to improve its performance after it was fully functional. We achieved our primary goal by building our simulator on top of an existing uniprocessor simulator, specifically, the simulator and related compilation tools provided in the SimpleScalar tool kit. We also used the *process pipelining* idea to mimic the thread pipelining execution model of the superthreaded processor.

Relying on the operating system to create and destroy processes to simulate the superthreaded processor's initiation and termination of very lightweight parallel threads allowed us to hide the details of the underlying SimpleScalar simulator. Instead of worrying about the specific functions of global variables and so forth in the SimpleScalar simulator, we could focus our efforts on adding only those features necessary to simulate the superthreaded processor.

The penalty of this information hiding is the factor of 3.3 to 4.0 times slowdown in the simulated instruction execution rate of our simulator compared to the base SimpleScalar simulator. Also, using the operating system's heavyweight threads to simulate the lightweight thread processing units introduces enough overhead to eliminate much of the performance benefit that could be obtained through the use of multiple processors.

As we have worked with the SimpleScalar simulator tool set over the past few months, we have gradually begun to understand its internal operation. This understanding has allowed us to make more substantial changes and expansions to the base simulator in an attempt to improve the performance of our simulator. For example, instead of creating and destroying an entirely new heavyweight process for each `st_fork` and `stop` instruction executed by the superthreaded processor, we have modified our simulator to create a fixed number of processes at the beginning of the simulation. Each one of these processes then idles waiting to be assigned to execute one simulated lightweight thread when the simulated superthreaded processor executes an `st_fork` instruction. The process then returns to an idle state when a simulated `abort_future` or `stop` instruction is executed.

Eliminating the overhead of the heavyweight process creation and termination more than doubles the performance of the simulator compared to the original approach. Due to the information hiding provided in the original approach, and our exhaustive testing, modifying the simulator for this new execution strategy took very little time. We are continuing our performance optimizations by examining ways of further reducing the multiprocessor overhead in the hopes of exploiting the parallelism inherent in the superthreaded processor architecture itself.

Acknowledgements

We would like to thank the developers of the SimpleScalar tool set for making this simulation package available to the computer architecture research community. We also would like to thank Jenn-Yuan Tsai for helping us understand the superthreaded processor architecture in enough detail to be able to simulate it. This work was supported in part by the National Science Foundation under grant nos. MIP-9610379, CDA-9502979, and CDA-9414015.

References

[1] G. M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities,". Proc. AFIPS 1967 *Spring Joint Computer Conf.*, Atlantic City, N.J., 483-485.

[2] J. Tsai, P.-C. Yew. "The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation". In the *Proceedings of Int'l Conf. on Parallel Architecture and Compiler Techniques* (PACT'96), 1996.

[3] Jenn-Yuan Tsai, Zhenzhen Jiang, Zhiyuan Li, David J. Lilja, Xin Wang, Pen-Chung Yew, Bixia Zheng, Stephen J. Schwinn, and Robert Glamm. "Integrating Parallelizing Compilation Technology and Processor Architecture for Cost-Effective Concurrent Multithreading". *Journal of Information Science and Engineering, Special Issue on Compiler Techniques for High-Performance Computing*, March 1998.

[4] D. Burger, T. Austin and S. Bennett. *Evaluating Future Microprocessors: the Simple Scalar Tool Set.* http://www.cs.wisc.edu/˜ mscalar/simplescalar.html

[5] J. E. Veenstra and R. J. Fowler. *MINT Tutorial and User Manual.* Technical Report 452, Department of Computer Science, University of Rochester, June 1993.

[6] M. Brorsson, F. Dahlgren, H. Nilsson, and P. Stenström. “The CacheMire Test Bench - A Flexible and Effective Approach for Simulation of Multiprocessors”. In the *Proceedings of 26th Annual Simulation Symposium*, pp. 41-49, March, 1993.

[7] FAST-MIPS v5.0. http://www-mount.ee.umn.edu/mcerg/fast-mips/.

[8] J. Hennessy and D. A. Patterson. *Computer Architecture: A quantitative Approach*, Second Edition, 1996. Morgan Kaufmann Publishers.

[9] R. C.Covington, S. Madala, V. Mehta, J. R. Jump, and J. b. Sinclair. “The Rice Parallel Processing Testbed”. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 4-11, 1988.