The SImulator for Multi-threaded Computer Architecture (Release 1.2)

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

Overview

The SImulator for Multi-threaded Computer Architecture(SIMCA) is built on top of the SimpleScalar tool set [1] in an effort to evaluate the performance of the superthreaded architecture [4, 5], and to explore the different design alternatives. Our compiler can compile superthreaded source codes written in C or FORTRAN into superthreaded binary, and this binary runs on the SIMCA. All processes are automated.

The performance of SIMCA with no compiler optimization on an SGI Challenge Cluster with R1000 processors is about a 15 to 20 thousand instructions per second when the program is highly superthreaded, and about 15 thousand instructions per second when only one thread-unit is active. The main contribution of this simulator is that it resolves many questions on the details of the hardware design, and it serves as a guide for the actual hardware implementation.

1 Introduction

The superthreaded architecture [4, 5] uses the thread-pipelining model to execute multiple threads concurrently for better performance. Data dependence is resolved in runtime while control dependences are speculated. In order to evaluate this architecture thoroughly, we need a detailed simulator. We started the development of SIMCA with an approach called process-pipelining [3] which forks a new process to simulate each activation of a thread unit. This approach helps to hide the underlying details of the base simulator, and to reduce the chances of introducing errors. The programmers can concentrate on implementing the new features of the proposed architecture at the early steps of development, and worry about the performance of the simulator later. This experiment turned out successful in that we quickly produced a functioning version within 6 months with minimal human resources [2].

However, the process-pipelining approach can not boost the speed of the simulator measured by the number of instructions it can simulate each second. We adopted a method to use a fixed number of processes to simulate the corresponding number of thread units. The speed of the simulator more than doubled. But this method incountered difficulty to simulate those programs that open or close files frequently. The current version [2] uses a single process to simulate the whole super-threaded processor, and its performances is even better than the underlying SimpleScalar tool set, due to the novel features of the super-threaded architecture. This single-processed version maintains the simulation model developed by [3] and keeps many of the variable names used there.

This document is written to help you install, and use the simulator. For a complete description of the super-threaded architecture, please refer to [4]. We are releasing both the source code and the binaries at [2]. All non-commercial users are welcome to run, to modify, and to redistribute this simulator for their own research purpose. The rest of the copy rights are reserved. This document is organized as follows: Section 2 talks about installation of SIMCA. Section 3 helps you to hand-compile source code for superthreaded architecture. Section 4 describes the implementation of SIMCA in detail.

2 Obtaining and Installing SIMCA

SIMCA 1.2 only runs on big-endian machines at this time and has been tested on SUN SunOS 5.6 and SGI IRIX 6.2. All necessary files are deposited at http://www.cs.umn.edu/Research/Agassiz/simca.html. Five files are needed.

- *simpletools.tar.gz* : This file contains the SimpleScalar *gcc 2.6.3* source files. It is not modified to support the direct compilation of FORTRAN programs.
- *simpleutils.tar.gz* : This file contains the SimpleScalar assembler, (ss-gas) and SimpleScalar loader (ss-gld) source code. Ss-gas is modified to accept super-threaded instructions.
- *simca.tar.gz* : This is the simulator source code.
- *simca.bin.tar.gz* : This archive contains four different binary programs precompiled for the superthreaded processors with 2, 4, 8, and 16 thread processing elements. It also contains the parser tool called *replace*. This tool is used after the generation of assembly code to replace particular function calls with the corresponding superthreaded instructions. The extension of *sun* indicates that the corresponding binary is compiled for SUN OS, and the extension of *sgi* tells that the binary is for Silicon Graphics IRIX operating system.
- *st-spec.tar.gz* : This archive has seven programs that are hand-compiled to run on SIMCA.

To install the simpletools and simpleutils, please follow the document to install SimpleScalar [1]. You can unpack *simca.tar.gz, simca.bin.tar.gz*, and *st-spec.tar.gz* in any directory you like by typing the following:

zcat simca.tar.gz | tar xvf -

zcat st-spec.tar.gz | tar xvf -

Please add the directory where you put the simulator to your PATH environment variable by adding this line to your *.cshrc* file.

setenv PATH ${PATH}:your-SIMCA-location

The *CC* variable in the Makefiles of SPEC programs should be modified to reflect the location of your compiler. For example, if your simpletools and simpleutils are installed under

/home/my_account_name/simplescalar/,

the installation of these two packages will put the compilers under

/home/my_account_name/simplescalar/ssbig-na-sstrix/bin

Then the *CC* variable should have the value of */home/my_account_name/simplescalar/ssbig-na-sstrix/bin/gcc*. Compile the binaries of the SPEC programs by typing *make* in each program directory, you are then ready to run the simulation. The simulator can be started by typing

simca.x_program_binary <program_parameters>

Here *x* can be 2, 4, 8, or 16 to represent different number of thread units. A sample output file is also included in the *simca.tar.gz* package. The file name is *sample-output.txt*.

3 Organization of the Simulator

SIMCA requires the support of SimpleScalar's modified *gcc, gas*, and *gld*, which are referred to as *ss-gcc*, *ss-gas*, and *ss-gld* respectively in this document. New instructions which are required by the Super-threaded architecture are added and support is provided in *ss-gas*.

3.1 Running Programs Written in C

All benchmark codes are hand-compiled to comply with superthreading concept. A particular superthreading function call should be inserted wherever a suerthreaded instruction is needed. Figure 2 shows an example C-code segment with a loop that changes the value of each element in an array base on the value of its neighboring element with a smaller index value. This code segment has loop-carried data dependences. The corresponding hand-compiled code is shown in Figure 3. Here the prefix *st_* indicates that this function call corresponds to a superthreaded instruction. The suffix *_b, _h, _w*, and *_d* tells the size of the data (byte, two bytes, four bytes, and eight bytes) at the referenced address. All the functions that help to execute a program in the the superthreaded mode have to be declared as in Table 1. File *stmacros.h*, which is in the package of *st-spec.tar.gz* should be included for any file that has any superthreaded code. These modified benchmark source codes can be fed to *ss-gcc* with *-S* flags to produce assembly code. The assembly code is then parsed by the *replace* tool. This tool recognizes all the special function calls. It will replace all the function calls by the corresponding super-threaded instructions. Finally the assembly code can be passed to *ss-gas* and *ss-gld* to produce the binary code accepted by SIMCA. Figure 1 shows the complete process.

In the super-threaded code, the function call ST_BEGIN signals the beginning of a super-threaded region. The parameter *11* helps to identify the region in the source code level by the programmer. ST_END_REGION ends the corresponding region. This kind of region is similar to a DOACROSS loop in parallel FORTRAN codes. ST_LFORK indicates that the current thread unit can fork a successor thread and the address that the successor thread should start execution with is the instruction after ST_BEGIN. Other function calls have the same semantics as the superthreaded instructions they represent.

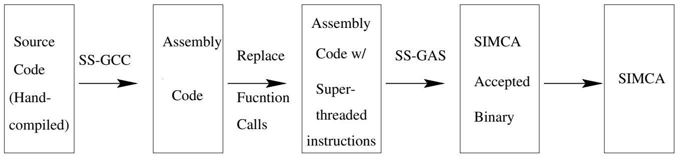

Figure 1: The SIMCA Approach

```
int i, tempi; float a[10], tempf;
a[0] = 8; for (i=1; i<10;i++) {
    a[i] = a[i-1]+4;
}
```

Figure 2: An example code segment written in C.

```
    ST_BEGIN(11);      /* Continuation Stage */
    tempi = i;
    ST_STORE_TS_W(&i, tempi+1);
    if (tempi >= 9) {
       goto ST_12; }
    /* fork next thread */
    ST_LFORK();
/* TSAG Stage */
    ST_WAIT_TSAG_DONE();
    ST_ALLOCATE_TS_W(&a[tempi]);
    ST_TSAG_DONE();
/* Computation Stage */
    tempf = a[tempi-1]+4;
    ST_STORE_TS_F(&a[tempi], tempf);
ST_12: /* WB Stage */
    ST_END_REGION();
```

Figure 3: The superthreaded code for the code segment shown in Figure 2.

3.2 Running Programs Written in FORTRAN

We have included the *g77* compiler in our distribution. Hence, FORTRAN programs can be compiled using the same *ss-gcc* binary to run on SIMCA. FORTRAN programs use pass-by-reference convention for function and procedure call parameters. Hence, the *Replace* tool has to add an additional instruction to load the values in order for the subsequent *STORE-TS* instruction to function normally. This is an additional overhead, but it is the price we pay to automate this process.

The program shown in Figure 4 is the FORTRAN counterpart for the C program shown in Figure 2. And the program shown in Figure 5 is the superthreaded FORTRAN code. We can see that the only difference between the FORTRAN and the C programs is the handling of parameters. Since FORTRAN passes parameters by reference, we do not need to add an "&" sympol to obtain the address of the variable for the parameter of ALLOCATE_TS, and the first parameter of STORE_TS. However, we cannot force the second parameter of STORE_TS to be the value instead of the address of the variable without the assistance of another function. Adding a function call to return the value of the variable will incur unnecessary overhead for the superthreading process. Hence, we resolve this issue by adding a load instruction right before the *store_ts* instruction in the processing of the assemply code. This only adds one instruction as the overhead. By doing so, the source-to-source compiler like the *Agassiz* [6] can treat programs written in these two languages in the same way.

3.3 Parameters of SIMCA

Parameters are supported by using a configuration file named *simca.config* for the super-threaded part. The supported parameters are (x should be replaced with the value you want):

- The number of memory buffer ports. Use "MB_PORTS: x" in the config file.
- The number of ports for the communication ring. Use "RING_PORTS: x" in the config file.

```
REAL A(10), TEMP1
REAL TEMP1
INTEGER TEMP2

A(1) = 8
DO WHILE I¡= 10
A(I) = A(I-1)+4
ENDDO
```

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

```
REAL A(10), TEMP1
INTEGER TEMP2

A(1) = 8
I = 2
CALL ST_BEGIN(1)
TEMP2 = I
I = I + 1
CALL ST_STORE_TS_W(I, I)
IF ((10-I)*1 .GE. 0) THEN
   CALL ST_LFORK(1)
ENDIF
CALL ST_WAIT_TSAG_DONE()
CALL ST_ALLOCATE_TS_W(A(TEMP2))
CALL ST_TSAG_DONE()
TEMP1 = A(TEMP2-1)+4
CALL ST_STORE_TS_W(A(TEMP2),TEMP1)
CALL ST_END_REGION()
END
```

Figure 5: The superthreaded code for the code segment shown in Figure 4.

- The processing speed of communication unit (in number of messages per cycle). Use “CU_SPEED: *x*” in the config file.

- The number of ports for the inter-thread-unit communication ring. Use “RING_PORTS: *x*” in the config file.

- The size of the communication unit. Use “CU_SIZE: *x*” in the config file.

- Associativity of memory buffer. Use “MB_ASSOC: *x*” in the config file. 0 refers to fully-mapped, while other positive integers stands for *x*-way set associative.

- The delay to fork the next thread. Use “FORK_DELAY: *x*” in the config file.

- The wire delay for a message to be delivered through the communication ring. Use “RING_DELAY: *x*” in the config file.

3.4 Outputs of SIMCA

The output of the simulator is written to standard output. It prints the settings used to run the simulation, and the information about number of threads initiated, number of threads squashed, number of entries touched in memory buffer, the average time needed to forward memory buffer entries to successor threads, the average time taken to resolve data dependences, the average time taken to finish write-back, and the average life-span of each thread instance. The number of instructions that are squashed due to failed thread speculation refers to the committed instructions. It does not include squashed instructions due to branch-misspeculation. The statistic *sim_num_insn* in the output file refers to the total number of instructions that are committed, and are not squashed in any thread units. Another statistic *sim_total_insn* refers to the total number of instructions that are executed but not necessarily committed in the execution. It does include the number of instructions executed in the squashed threads. Hence, the following relation holds:

- *sim_num_insn + Number of Committed Instruction Squashed due to thread misspeculation + squashed instructions due to branch misspeculation = sim_total_insn*

Due to the difficulty to track the actual boundary of different thread-pipelining stages, the average length of each stage shown in the result file is not accurate. For the explanation of other outputs, please refer to [1].

4 Implementation Details

4.1 Architecture Model

This distribution of SIMCA is trying to simulate the micro-architecture shown in Figure 6 for each thread processing element. It has four copies of the binary. The file *simca.x* refers to the simulator with *x* thread processing elements. Every thread processing element is itself a superscaler processor. Each unit has a 128-entry memory buffer, 8-entry communication unit, 2 memory buffer write ports, and 2 communication-unit to memory-buffer ports by default. All units share a 2-level cache hierarchy, main memory, I-TLB, and D-TLB. Instruction cache and data-cache are separated in level-one cache. Level-two cache is shared by both instruction and data. Level-one cache has a one-cycle delay, and level-two cache has a total delay of 6 cycles. First chunk of memory data arrives 18 cycles after memory access, and each additional chunk cost 2 cycles.

4.2 Semantics of the Superthreaded Instructions

The following is the alphabetical list of superthreaded instructions and their format.

- abfutr : Abort the successor thread, and propagate this request. The current thread becomes the last active thread in this superthreaded region.

- altsb rs : Allocate a TSAG entry in local memory buffer, and forward this request to the successor thread (if any). The target-store distance vec-

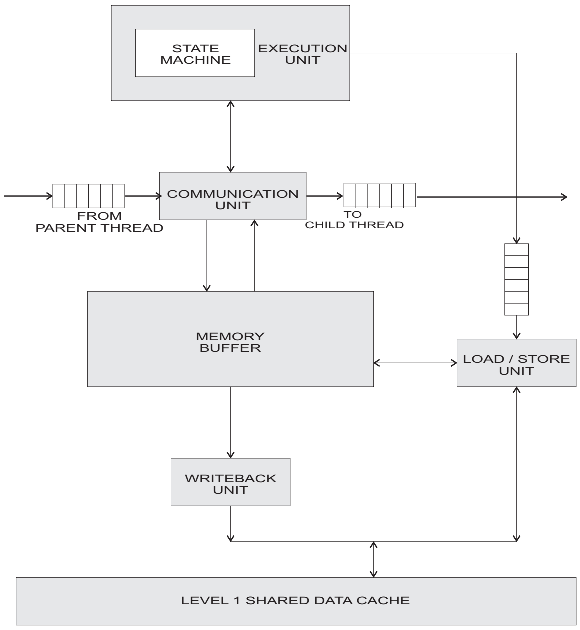

Figure 6: The microarchitecture of a thread processing element in SIMCA

tor is changed to reflect the source of this propagation. The address is equal to the value in register *rs*. The data takes one byte of space.

- altsh rs : Same as altsb, but the data takes two bytes of space.
- altsw rs : Same as altsb, but the data takes four bytes of space (one word).
- altsd rs : Same as altsb, but the data takes eight bytes of space (double word).
- bstr : Signals the start of a super-threaded region. Store the address of the next instruction into the global buffer used for fork-label. All succeeding threads will start execution with this address. It also copies the state of the current thread unit into the global state.
- estr : End of a super-threaded region. This instruction forces the thread unit to write back to the cache any locally written entries in its memory buffer. When write-back is done, the current thread has finished all superthreaded task, and is allowed to release the thread unit if it as a non-aborted successor thread. Otherwise the current thread continues executing instructions after $estr$.
- glock $immed$: acquire lock, lock ID is *immed.*
- hfrk rs : The generic fork instruction for this architecture. *Hfrk* enables the successor thread unit, forwards necessary local memory buffer entries to successor threads. It takes the address stored in register rs as the fork-label, which is the address for all successor threads to start execution. If the address stored in rs is different from what is saved as the fork-label, it indicates a new super-threaded region and local registers are copied to global state.
- lfrk : The fork instruction optimized for DOACROSS loop model. It simply enables the successor thread unit, forwards necessary local memory buffer entries to successor threads. Must be used with *bstr* to function.
- rlock $immed$: release lock, lock ID is *immed.* When a thread acquires a lock successfully, all memory references bypass its memory buffer and go directly to the cache. *glock* and *rlock* are used for mutual exclusion purposes.
- rltsb rs : Release a TSAG entry in local memory buffer, and forward the valid data in local memory buffer (if any) to the successor thread. If no valid data is present, just forward this request. The target-store address vector is modified to reflect the source of this request in the propagation process. The address of the data is stored in register *rs*. The data takes one byte of space.
- rltsh rs : Same as rltsb, but the data takes two bytes of space.
- rltsw rs : Same as rltsb, but the data takes four bytes of space (one word).
- rltsd fs : Same as rltsb, but the data takes eight bytes of space (double word).

- sttsb rd, rs : Save the data in register rs to the address of *rd* in local memory buffer. Forward this request to to the successor thread (if any). The target store distance vector is set to reflect the source of this request in the propagation process. The data takes one byte of space.

- sttsh rd, rs: Same as sttsb, but the data takes two bytes of space.

- sttsw rd, rs: Same as sttsb, but the data takes four bytes of space (one word).

- sttsd rd, fs: Used for floating-point data. Same as sttsb, but the data takes eight bytes of space (double word).

- tsagd : Send the TSAG_DONE flag to the successor thread.

- wtsagd : Wait for the TSAG_DONE flag from the predecessor thread. If the current thread is activated before the parent thread finishes its TSAG stage, the current thread will stall all of its subsequent memory operations, and wait for the *TSAG_DONE* message from its parent to continue. Otherwise, this instruction has no effect.

4.3 Function Calls in the Source-Code

In the source code, one special function call corresponds to a super-thread instruction, as shown in Table 1. ST_ALLOCATE_TS and ST_RELEASE_TS both have four versions. ST_ALLOCATE_TS_B is for *altsb*, and ST_RELEASE_TS_B is for *rltsb*. Here *_B* is used for data of one-byte. *_H, _W*, and *_D* can be used to replace *_B* for different data types. The person who does hand-compilation is responsible for identifying the data type and placing the correct versions of function calls. ST_STORE_TS has a fifth version, besides the stand four. This unusual one is the ST_STORE_TS_F, which is used for single-precision floating point data. Because we are storing and forwarding raw data using STORE_TS, we don't want the compiler to covert the floating point data into integer data before calling STORE_TS. However, both ST_STORE_TS_W and ST_STORE_TS_F correspond to the instruction *sttsw*. FORTRAN programs do not have this problem, since they always pass the parameters by using the address of the variable.

ST_FORK and ST_LABEL are used as a pair. The latter indicates where the successor thread should start execution, while the former tells where to start the forking process. The parameter to both of them is an integer label-id. The *replace* tool will match the label-id for these two function calls, and manually insert a label with syntax "ST_x:". Here *x* matches the label-id number. This manually inserted label will be loaded to a general register, and this general register will be used as the operand for *hfrk*. This pair of function calls are designed for overlapped execution with general purposes. ST_BEGIN and ST_LFORK are introduced to spawn new threads more efficiently. ST_BEGIN declares the beginning of the current superthreaded region and marks the instruction right after itself the starting point of all of the subsquent threads in the current region. This will provide the opportunity for the later threads to start the initialization process earlier.

ST_WAIT_TSAG_DONE is used by the current thread to wait for the predecessor to signal the end of its TSAG stage, it corresponds to *wtsagd*. ST_TSAG_DONE, which corresponds to *tsagd*, is to signal the successor when the current thread has finished TSAG_STAGE. If the parent thread already executed the it tsagd instruction before the child thread is actually spawned, the execution of *wtsagd* instruction by the child thread has no effect.

ST_ACQUIRE_LOCK and ST_RELEASE_LOCK also take one parameter to identify the ID of the lock they refer to. Currently 32 locks (0 - 31) are supported. The instructions corresponding to ST_FORK, ST_LFORK, ST_BEGIN, ST_WAIT_TSAG_DONE, ST_TSAG_DONE, ST_ACQUIRE_LOCK, and ST_RELEASE_LOCK are treated as synchornization instructions.

4.4 Known Problems

The current version only works on big-endian architectures. Although the SIMCA is coded to support

Instruction	Function Call Prototype
Fork-Label	*ST_LABEL*(label-id)
abfutr	*ST_ABORT_FUTURE*()
altsb	*ST_ALLOCATE_TS_B*(address)
altsh	*ST_ALLOCATE_TS_H*(address)
altsw	*ST_ALLOCATE_TS_W*(address)
altsd	*ST_ALLOCATE_TS_D*(address)
bstr	*ST_BEGIN*(region-id)
estr	*ST_END_REGION*()
glock	*ST_ACQUIRE_LOCK*(lock-id)
hfrk	*ST_FORK*(label-id)
lfrk	*ST_LFORK()*
rlock	*ST_RELEASE_LOCK*(lock-id)
rltsb	*ST_RELEASE_TS_B*(address)
rltsh	*ST_RELEASE_TS_H*(address)
rltsw	*ST_RELEASE_TS_W*(address)
rltsd	*ST_RELEASE_TS_D*(address)
sttsb	*ST_STORE_TS_B*(address, data)
sttsh	*ST_STORE_TS_H*(address, data)
sttsw	*ST_STORE_TS_W*(address, data)
sttsw	*ST_STORE_TS_F*(address, data)
sttsd	*ST_STORE_TS_D*(address, data)
tsagd	*ST_TSAG_DONE*()
wtsagd	*ST_WAIT_TSAG_DONE*()

Table 1: Function call prototypes for super-threaded instructions

the case of using the double precision variables for resolving data dependences, this scenario is not tested. In addition, when the number of cycles exceeded the limit of a 32-bit unsigned integer, the simulator stalls. We are working on these problem right now.

5 Acknowledgements

This project is supported in part by the National Science Foundation Grant No. MIP-9610379, CDA-9502979, and CDA-9414015. Many thanks to Chris Amlo, who has contributed a large amount of effort to help debugging this simulator.

References

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

[2] J. Huang. The SImulator for Multithreaded Computer Architecture (SIMCA), Release 1.2. http://www-mount.cs.umn.edu/ Research/ Agassiz/ simca.html

[3] J. Huang, D. J. Lilja. "An Efficient Strategy for Developing a Simulator for a Novel Concurrent Multithreaded Processor Architecture", in the *Six Int'l Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Montreal, Canada, July, 1998.

[4] J. Tsai, J. Huang, C. Amlo, D.J. Lilja, and P.-C. Yew. "The Superthreaded Processor Architecture". To appear in the IEEE *Transaction on Computers, Special Issue on Multithreaded Architectures*, September, 1999.

[5] J. Tsai, P.-C. Yew. "The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation". *PACT'96*, pp. 35-46.

[6] B. Zheng, J.-Y. Tsai, B.Y. Zang, T. Chen, B. Huang, J.H. Li, Y.H. Ding, J. Liang, Y. Zhen, P.-C. Yew, C.Q. Zhu. Designing the Agassiz Compiler for Concurrent Multithreaded Architectures. In *Workshop on Languages and Compilers for Parallel Computing*, San Diego, August, 1999.