Implementation and Evaluation of the Simultaneous Multithreading Model using SimpleScalar v4.0

Hansung Kim
Department of Electrical and Computer Engineering
Seoul National University
stephen422@snu.ac.kr

***Abstract*—The simultaneous multithreading model is implemented using an execution-driven timing simulator. The evaluation results presented in the original work in SMT are reproduced, and shows overall coherent results with some inconsistency.**

I. Introduction

The simultaneous multithreading model [1] has exhibited a major opportunity in accelerating performance of multithreaded workloads by exploiting the issue slot wastage present in superscalar processors. In this work, the multithreading model is implemented using the SimpleScalar simulator [4] version 4.0, and is evaluated using methods analogous to those utilized in the original studies, i.e. comparison with FGMT model [1] and evaluating various fetch partitioning and thread selection logic [2].

II. Details of the Implementation

A. Context-specific Hardware Resources

As SimpleScalar is originally designed to simulate a single-threaded processor, several modifications are necessary to adapt the processor into a multi-threaded one. One of the major modifications is the addition of multiple architectural states, or *contexts*, for each thread. The hardware structures that store architectural states include register files and the program counter. For convenience, the set of these context-specific resources are collectively declared as the form of a C structure called `context_t` in `sim-outorder.c`.

The physical memory object is also replicated for all contexts. This is not necessary in real processors, as the virtual memory system handles the segregation of address space between processes. However, for ease of implementation, each context holds separate physical memory that is tagged with its context ID. This scheme effectively disallows any form of memory sharing between contexts, but is sufficient to simulate *multiprogramming* workloads.

Separation of the architectural states is not enough for efficient implementation of SMT. The reorder buffer (RUU in the context of SimpleScalar) and load store queue are also replicated for each context. Sharing ROB and LSQ between context is possible, but may complicate the branch misspeculation recovery logic because only the instructions from the faulted context has to be selectively squashed from the pipeline. Squashing all earlier instructions in the pipeline in those cases may lead to severe performance disadvantage.

Along with ROBs and LSQs, various microarchitectural hardware states such as predicted program conuter (`spec_mode`), speculation mode flag (`spec_mode`), rename table (`create_vector`) and fetch delay counter (`ruu_fetch_issue_delay`) are also replicated. See the definition of `struct context_t` for details.

Ramsay [5] suggests that the replication of branch prediction table yields better performance. In this work, however, it is left to be shared between contexts for ease of implementation.

B. Modification to Cache Access Logic

Since each context holds separate physical memory, caches also need to account for which context it is that the memory block currently being accessed belongs to. To accomplish this, each cache block (represented as `struct cache_blk_t` in `cache.h`) is tagged with the context ID. The tagged ID is then taken into account for all cache hit and miss determination and block replacement logic (`cache.c`).

C. Fetch Partitioning and Thread Selection Policy

Tullsen et al. [1] discusses various partitioning schemes of the fetch unit and their effect on the instruction throughput. Among the suggestion, RR.1.8, RR.2.4 and ICOUNT.2.8 is implemented for evaluation. Because of an artifact in the implementation that causes deadlock under the RR.1.8 scheme for certain situations, an additional scheme called RAND.1.8 is also implemented. Its only difference from RR.1.8 is that it selects the thread to fetch in a random fashion instead of a round-robin scheme.

The 2.8 fetch schemes enable dynamic partitioning of the fetch bandwidth between the two selected thread. In essence, the first thread is fetched from as much as possible, and it is only switched to the second thread if the fetch unit encounters an I-cache miss or a branch, or the fetch buffer is full. An additional condition is engaged if the dynamic partitioning is disabled, i.e. in the 2.4 partitioning scheme. This is explained in the following code excerpted from function `ruu_fetch()`.

Listing 1. 2.8 fetch thread selection logic

```
/* SMT: 2.8 fetching: fetch as much as we can from the first
   thread, and then fetch the rest from the second thread. */
if (/* encountered a branch? */
```

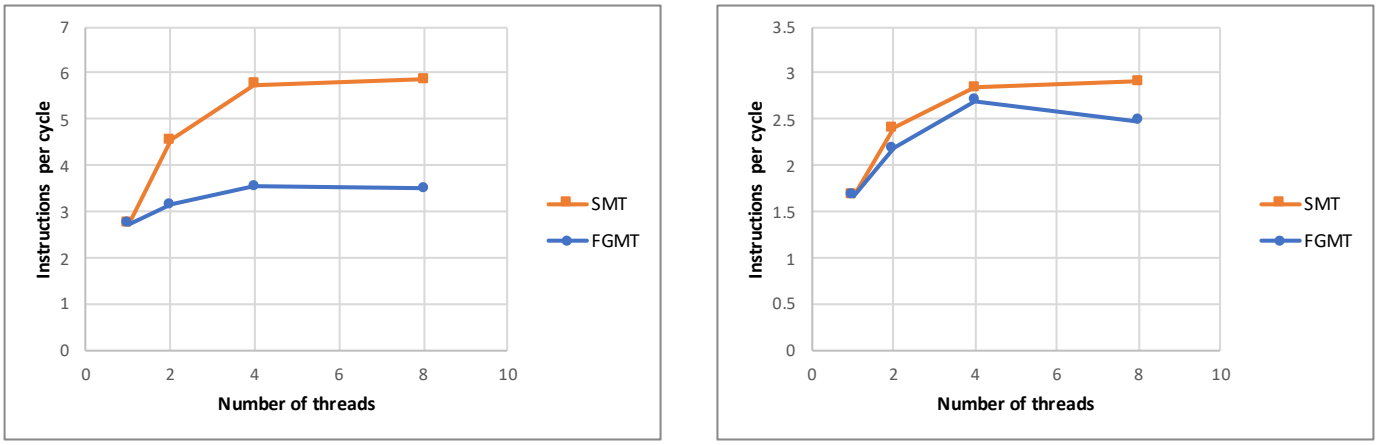

Fig. 1. Instruction throughput of the SMT and FGMT model for varying number of threads with bzip2 (a) and mcf benchmark (b).

```
    branch_cnt[ctx_id] > 0
    /* 2.4 fetching: depleted BW quota? */
    || (!dynamic_fetch_bw
        && ctx_id == selected[0]
        && i >= (ruu_decode_width * fetch_speed) / 2)
    /* I-cache blocked? */
    || contexts[ctx_id].ruu_fetch_issue_delay > 0
    /* IFQ queue filled? */
    || contexts[ctx_id].fetch_num >= ruu_ifq_size
    /* done flag set? */
    || done[ctx_id])
{
    /* already switched to the second thread? abort */
    if (ctx_id == selected[1])
        break;
    /* if not, try again with the second thread */
    ctx_id = selected[1];
    if (dynamic_fetch_bw)
        /* keep BW usage from incrementing */
        i--;
    else
        /* fixed partitioning: used up BW quota */
        i = (ruu_decode_width * fetch_speed) / 2 - 1;
    continue;
}
```

D. Fine-grained Multithreading Model

Fine-grained multithreading model differs from SMT in that only a single context is allowed to be issued from for each cycle. The target context for issue is updated for every cycle, usually in a round-robin selection scheme. In this work, FGMT is implemented with minimal modification from the SMT implementation: in the issue logic (`ruu_issue()`), each issue queue entry in the *ready list* is examined, and only those whose context ID tag matches the current target context is allowed to issue.

III. Evaluation

A. Workload

The implementation is evaluated against two multiprogramming workloads: SPECint2006 *401.bzip2* and *429.mcf*. bzip2 is run with input arguments *liberty.jpg 30*, and mcf with *inp.in*. Both are fast-forwarded by 2 billion instructions, and measured with the following 1 billion instructions. For each workload, the simulator is initialized with all contexts containing the same program data and program counters; i.e. all threads run the same program with idential initial state.

Threads	FGMT	SMT	Threads	FGMT	SMT
1	2.7292	2.7292	1	1.6742	1.6742
2	3.1431	4.5485	2	2.1843	2.4046
4	3.5503	5.7431	4	2.6964	2.8436
8	3.5100	5.8529	8	2.4860	2.9051

Fig. 2. IPC values for the bzip2 and mcf benchmark

B. Instruction throughput

The instruction throughput of the SMT model with increasing number of threads is presented in Figure 1 and 2. The instruction throughput, measured as the average commited instructions per cycle, consistently increased as the number of threads increased. For the bzip2 (liberty.jpg) workload, the instruction throughput with 8 threads (5.8529) achieved an IPC value higher by over a factor of two than that with 1 thread (2.7292). The amount of IPC increase for each step got smaller as the number of threads increased. This is most possibly due to the resource contention, meaning the limited number of hardware resources such as functional units and cache ports may cause contention between threads, which hinders performance advantage.

C. Comparison with Fine-grained Multithreading Model

The instruction throughput of the FGMT model is also shown in Figure 1. The speedup was consistently much smaller than SMT. This indicates the limitation of FGMT in terms of the capability in reducing issue slot waste: while FGMT may reduce vertical waste [1], it cannot reduce horizontal waste by only allowing single thread to issue in each cycle.

Furthermore, for mcf, the speedup over the single thread peaked at 4 threads and deteriorated at 8 threads. This result is in line with what Eggers [3] presented. Eggers explained this result with two factors: first, four threads were sufficient to eliminate vertical waste given the latency-hiding capability of

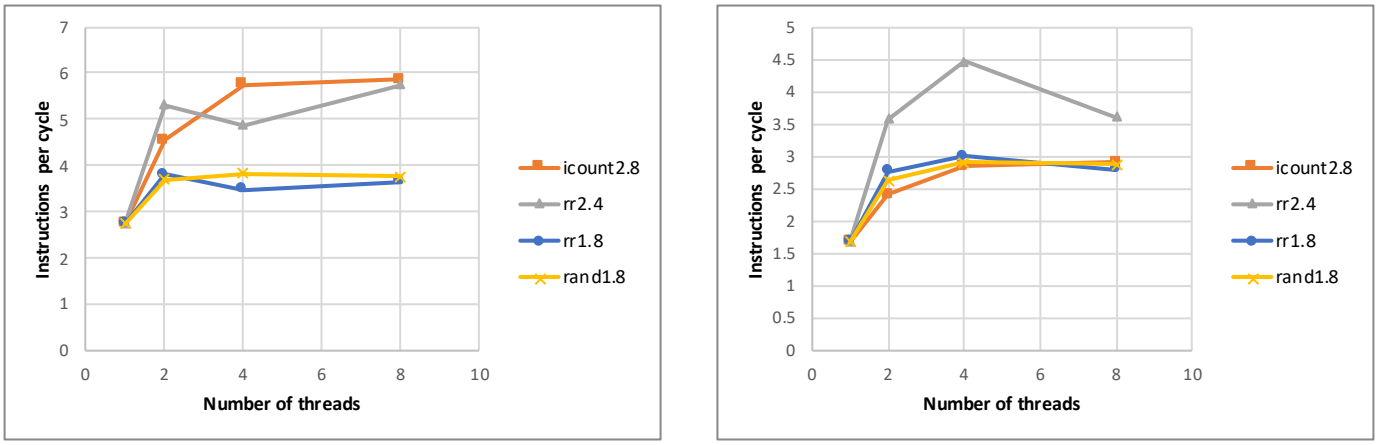

Fig. 3. Instruction throughput of the SMT model as a function of different fetch policies with bzip2 (a) and mcf benchmark (b).

out-of-order processor and lockup-free caches; second, FGMT cannot hide the additional conflicts from interthread competition for shared resources, because it can issue instructions from only one thread each cycle.

It should be noted that the FGMT implementation in this work uses the "simpler" thread selection scheme that selects thread without considering the availability of that thread in advance. If a more adaptive scheme is used that only selects from available (i.e. non-blocked) threads, the speedup value may increase for FGMT.

D. Effect of the Fetch and Thread Selection Policy on Performance

Figure 3 shows the instruction throughput comparison between various fetch policies. Between 1.8 and 2.4 partitioning, 2.4 consistently exhibits better performance over the single thread fetch scheme. This is in contrary to [2], where the 1.8 scheme wins over 2.4 with lower number of threads due to "thread shortage" in the 2.4 scheme. This potentially indicates that for this workload, there were little to no cycles where a single thread was able to fetch more than 4 instructions. Therefore, it may have been more beneficial to choose one more thread to fetch from, rather than allocating more of the already superflous fetch bandwidth to each thread.

Between the dynamic partitioning scheme (ICOUNT.2.8) and fixed scheme (RR.2.4), the result is mixed. ICOUNT.2.8 performed better with higher number of threads for the bzip2 benchmark, while RR.2.4 consistently performed better for the mcf. This is likely due to incorrectness of the fetch logic implementation, which is addressed in the following section.

E. Limitations and Incorrectness Issues

This SMT implementation leaves out opportunities for a more complete and efficient implementation of SMT, e.g. separate branch predictor table as suggested by Ramsay [5] and M-Sim [6], and two-stage extension of the register access stage and the necessary bypass logic modification to account for the larger register file as in Eggers [3].

The abnormal IPC advantage of RR.2.4 for the mcf benchmark, along with an unfixed bug that causes fetch deadlock under RR.1.8 running *test-printf*, indicates that there remains incorrectness issues within this implementation of the SMT model. The bug seems to recur more frequently with round-robin fetch policies, i.e. RR.1.8 and RR.2.4. Using randomization such as with RAND.1.8 is one way to cope with the issue by including nondeterminism to the simulator states. This is a temporary fix, however, and should be fixed and verified its correctness in the future work.

REFERENCES

[1] Tullsen, Dean M., Susan J. Eggers, and Henry M. Levy. "Simultaneous multithreading: Maximizing on-chip parallelism." ACM SIGARCH Computer Architecture News. Vol. 23. No. 2. ACM, 1995.

[2] Tullsen, Dean M., Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor." In ACM SIGARCH Computer Architecture News, vol. 24, no. 2, pp. 191-202. ACM, 1996.

[3] Eggers, Susan J., Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. "Simultaneous multithreading: A platform for next-generation processors." IEEE micro 17, no. 5 (1997): 12-19.

[4] Burger, Doug, and Todd M. Austin. "The SimpleScalar tool set, version 2.0." ACM SIGARCH computer architecture news 25, no. 3 (1997): 13-25.

[5] Ramsay, Matt, Chris Feucht, and Mikko H. Lipasti. "Exploring efficient SMT branch predictor design." In Workshop on Complexity-Effective Design, in conjunction with ISCA, vol. 26. 2003.

[6] "M-Sim: A Flexible, Multithreaded Architectural Simulation Environment", Technical Report CS-TR-05-DP01, Department of Computer Science, State University of New York at Binghamton, October 2005.