

Emulating Unimplemented Instructions in a Simultaneous Multithreaded Processor

Suan Yong and Brian Forney
CS/ECE 752
Spring 2000

Abstract

Emulating unimplemented instructions can reduce the cost and power requirements of a processor by allowing functional units to be removed. But the handling of unimplemented instruction exceptions in modern processors wastes fetch bandwidth and reduces throughput due to squashed instructions. Simultaneous Multithreaded (SMT) processors can avoid the waste by using multiple thread contexts to handle unimplemented instruction exceptions. We investigate the effectiveness of SMT in improving the performance of emulating unimplemented instructions. We focus on emulation of the Alpha 21164's integer multiply instructions as a proof of concept.

1 Introduction

Simultaneous Multithreaded (SMT) processors exploit thread-level parallelism for greater performance. Hardware thread contexts are used to fetch from multiple threads concurrently. This keeps the processor busier than fetching from a single thread. Research in SMT processors have shown promising performance improvement (about 2.5 times, according to [Tullsen]). Industry also appears to be embracing SMT. Multithreaded processors, which are a precursor to SMT processors, have already begun shipping [Storino, Alverson]. SMT processors are under development with the Alpha 21464 [Lipasti] as an example of a planned SMT.

SMT processors, however, require additional hardware beyond what a superscalar processor demands. SMT processors require a larger register file, multiple program counters, separate instruction retirement and exception handling, and modified branch prediction. Other structures, like a return address stack, may also need to be extended [Tullsen]. Additional components add to the complexity of a processor, often leading to increased processor cost, power, and development cost.

Cost and power considerations are important for mobile and embedded computing. Mobile computers have limited power resources, which typically are less than users would like. Users of mobile systems would like these systems to behave like desktop computers. Embedded computing, like mobile computing, has cost and power constraints. Embedded systems function in a variety of applications, including automobiles, PDAs, airplanes, and data acquisition devices.

SMT processors' additional complexity limits their use for applications like mobile and embedded computing, where cost and power are important. One solution is to remove functional units from the processor. However, partial emulation of the ISA will be needed to ensure binary compatibility. Partial emulation is implemented as an unimplemented instruction exception. Exception handling via trapping requires instructions fetched after the excepting instruction to be squashed. For out of order and SMT processors, this means almost all instructions in the instruction window are squashed since exceptions are not handled until commit time.

Previous work [Zilles] on exception handling demonstrated speed-ups for multithreaded processors. The key technique from this work is exploiting hardware contexts to switch between a

main thread and an exception handling thread. We propose an extension of this work for handling unimplemented instructions. This extension decreases the performance tradeoff made by removing functional units and emulating their behavior with software. A small amount of additional hardware is needed, but the addition is smaller than the removed functional units, and is fixed regardless of the number of functional units removed.

This paper is organized as follows. Section 2 discusses related work. We explain our technique for improved emulation of unimplemented instructions in section 3. Section 4 contains our simulation and benchmark selection methodology. Experimental results are presented in section 5. We suggest future work in section 6. The paper concludes in section 7.

2 Related work

Our work builds on previous work in several areas. Zilles *et al* [Zilles] at the University of Wisconsin and Compaq studied exception handling in multithreaded processors. This work proposed the use of hardware contexts to handle exceptions without squashing fetched instructions. When a processor generates an exception, the thread that created the exception is paused, and a new hardware thread is allowed to handle the exception. The hardware context switch conserves instructions fetched for the excepting thread after the excepting instruction. Zilles *et al* used TLB miss handlers as a case study of this technique. Our work is most closely related to this work.

Work at the University of Michigan by Chappell *et al* [Chappell] proposed the use of subordinate threads in a multithreaded processor to help a master thread. The subordinate threads provide services to the master thread. Services include cache prefetching and branch prediction allowing software greater control over program behavior than hardware based mechanisms. Threads in this model are started by a special SPAWN instruction injected in the binary by the compilation system. Unlike Zilles' work, subordinate threads fetch from a separate memory called a microRAM maintained by the operating system. Our emulation threads are fundamentally different from these subordinate threads in that our emulation threads need to have a higher priority than the main thread, since progress of the main thread is dependent on the completion of the emulation thread. A further difference is that emulation threads are started by processor exceptions rather than compiler-inserted SPAWN instructions.

Various emulation techniques exist. The techniques can be divided into two categories: full emulation of an ISA and emulation of unimplemented instructions, or partial emulation of an ISA. Complete emulation of ISAs has been used in various systems including the IBM/360 to run IBM 1401, 7070, and 7090 applications, the Digital VAX for Digital PDP-11 binary compatibility, the Apple PowerMac, HP's PA-RISC line of workstations, and Compaq's Alpha systems using FX!32 [Altman].

Binary translation of ISAs is a subarea of general complete ISA emulation and has received increased interest recently. Static and dynamic translators, like HP's Aries [Zheng] for PA-RISC to IA-64 translation, and Java VMs, are examples. These systems create a new binary either off-line or during program execution. However, unlike interpreters, the translations are cached temporarily or permanently.

Partial translation of an ISA traditionally occurs using trapping. Typically, the processor will generate an unimplemented instruction exception and trap into the operating system.

3 Emulating SMT

In this section, we elaborate on the algorithm for emulation of unimplemented instructions in an SMT and on the hardware changes needed to support better emulation of instructions when SMT hardware is available.

3.2 Emulation end functional unit

To finish the emulation, the emulation thread needs a mechanism to return its computed value to the main thread and stop the emulation thread. The *emulation end* functional unit performs this task (see Figure 2).

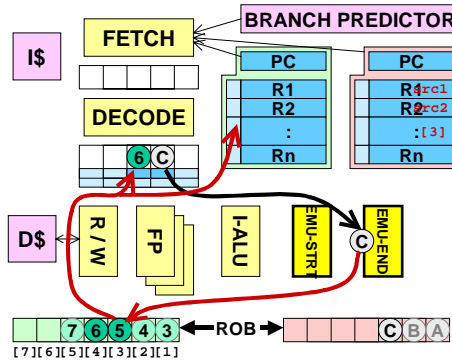


Figure 2: SMT processor core, showing *emulation end* functional unit.

When it is finished, the emulation thread will execute a return-from-exception instruction. This instruction will dispatch to the *emulation end* functional unit, passing it the ROB tag of the unimplemented instruction and the output value of the emulation thread. The *emulation end* functional unit will then update the main thread’s ROB entry.

When executing the return-from-exception, the *emulation end* functional unit performs several actions. First, the output will be broadcast on a result bus to update the ROB, reservation stations, and main thread’s register file. Second, the emulation thread context will be put to sleep and, if needed, reclaimed for another thread.

One factor that complicates the *emulation end* functional unit is speculative execution. Suppose the return-from-exception was after a branch. Based on the branch prediction scheme, the return-from-exception may be speculatively executed. If a simple approach of broadcasting the result from the *emulation end* functional unit occurs before the return is known to be non-speculative, an incorrect result will be placed in the ROB entry for the emulated instruction. This will lead to incorrect behavior.

Two possible solutions exist: broadcast the result of the emulation thread when the return-from-exception is retired, or require the return to be non-speculative. A broadcast from the ROB upon commit of the return requires an additional result bus and comparators. This is less than elegant and erodes the economic cost, power, and complexity advantage of emulating instructions. Using a non-speculative return is relatively inexpensive. Support for memory ordering is common in dynamically scheduled out of order processors [Zilles, Yeager]. This support could be extended to a non-speculative return.

3.3 Extra plumbing

Additional hardware beyond the two functional units is needed. The *emulation start* unit requires a bus to copy the two operands and the destination tag to the emulation thread’s register file, with a corresponding expansion of the muxes for three predetermined registers in each register file. The *emulation end* unit requires any additional hardware that adding a functional unit would normally require, including expanded muxes and broadcast bus.

4 Methodology

We now discuss our methodology in this section. We use emulation of the integer multiply functional unit as a proof of concept. (Our original plan was to emulate both integer multiply and divide, but the Zilles' SMT simulator uses the Alpha 21164 instruction set, which does not include an integer divide instruction.) Our methodology included modifying Zilles' *sim-multi* simulator, evaluating two multiply algorithms, and selecting appropriate benchmarks.

4.1 Simulator

We started with a version of Zilles' SMT simulator that did not include his work on exception handling. The simulated machine supports 4 threads, a decode window size of 32 instructions, 64KB each of L1 data and instruction caches, and 1 MB of L2 cache. It's functional units are: one simple integer ALU, one load and one store unit, an assortment of 6 floating point units, and one integer multiply unit, which is disabled when emulation is turned on. Other specifications of the simulated machine are as described in [Zilles]. Our simulator performs emulation of integer multiply instructions (mull, mulq, umulh) in four modes:

- “squash” mode: instructions fetched after a multiply instruction are squashed when the instruction is emulated. This mode mimics the behavior of emulation in a non-multithreaded processor.
- “pause” mode: when an emulation thread is activated, the original thread's fetch state is paused, so that no further instructions in the original thread are fetched until emulation completes.
- “ooo2” mode: this approach allows the original thread to continue fetching and executing after the emulation thread has been started; at most one emulation thread can be active.
- “ooo4” mode: similar to “ooo2”, except we allow up to three emulation threads to be active simultaneously.

The multiply instruction handler, which was compiled with Compaq's cc compiler, is 12 static instructions in length, with a single highly-predictable loop. It takes its two operands from the a0 and a1 argument registers, stores the computed result in the v0 return value register, and uses only one temporary register. Its return instruction was replaced with a bogus syscall instruction which our simulator intercepts and interprets as a “return-from-exception” instruction.

When an unimplemented multiply instruction is encountered, we first look for a free thread context; if none is available the instruction is kept in the reservation station. When a free thread context or sequencer is obtained, we copy the operands of the multiply instruction to the sequencer's argument registers, reset its program counter to the start of the multiply routine, and keep a pointer to the excepting multiply instruction (represented in the simulator by a *dynamic_inst* object). Zilles' simulator performs system calls non-speculatively by squashing subsequent instructions in the pipeline at the decode stage, and performing the system call at commit time. We employ the same mechanism in “squash” mode to start the emulation thread non-speculatively. For the other three modes, the emulation thread is started at execution time; the fetch unit is disabled only in the “pause” mode.

When a return-from-exception instruction (actually a bogus syscall instruction) is encountered, the result of the computation is copied to the destination of the multiply instruction in the original thread, and the original thread's fetch state is set to enabled (if it was disabled). The emulation thread is then disabled, and marked “available”.

Because we allow emulation threads to be started speculatively (except in “squash” mode), when an in-flight multiply instruction is squashed, we must also disable the emulation thread and

squash all its in-flight instructions. While this is easily done in the emulator, it may be non-trivial to build into a real processor. The tradeoffs of alternative approaches (e.g. allowing the multiply thread to run to completion and discard its results) may be worth studying.

A further complication we encountered is the possibility of deadlock. In the modes where we do not squash instructions, it is possible for instructions in the original program to occupy all of the reservation stations, so that the emulation thread has no window through which to execute. This is currently handled in an *ad hoc* manner by limiting the occupancy of each thread to 30 (of the 32) slots in the decode buffer (the simulator needs at least two slots per cycle to decode into). This solution is far from optimal, as an emulation executing through a 2-instruction-wide window isn't likely to achieve much parallelism. We look at some possible directions for further study in the future work section.

4.2 Multiply algorithm

Although investigating different multiply algorithms was not the goal of the project, we considered two algorithms. The first was a simple shift and add algorithm. The second was Booth's algorithm. We profiled both algorithms multiplying the same set of numbers on a SPARC system. (SPARC was used because good profiling tools on the available Alpha system were not installed.) The simple shift and add algorithm required fewer cycles than Booth's algorithm. Booth's algorithm averaged 3.7 times more cycles than the simple shift-and-add algorithm. Since the difference was so great, we did not investigate the modified Booth's algorithm any further. Booth's algorithm is typically faster in hardware than the straightforward and simple shift and add algorithm. We surmise that Booth's is slower in software due to parallelism achievable in hardware yet unavailable in software.

4.3 Selection of benchmarks

To evaluate our technique, we looked for benchmarks containing integer multiplies. Additionally, we were interested in benchmarks that are closer to real world mobile and embedded computing. We considered the SPEC CPU2000 [SPEC CPU2000] and MediaBench [MediaBench] suites. SPEC CPU2000 is a new version of the venerable SPEC processor benchmarks. They target CPU performance using integer and floating-point applications. The MediaBench suite contains multimedia applications, and perform such operations as image, video, and sound compression and decompression, encryption, and speech recognition.

To supplement these established benchmarks, we created two microbenchmarks to stress the emulation of the integer multiply unit. Both benchmarks contained a tight loop of multiplies and a few other basic arithmetic operations needed to prevent removal of the multiplies by the compiler's optimizer. One benchmark (*depend*) contains multiplies that form a long dependence chain (with two independent multiplies in each iteration), while the other (*independ*) contained relatively-independent multiplies.

We choose benchmarks primarily on the two criteria above: number of multiplies and relevance to mobile and embedded computing. Each benchmark was profiled using *sim-profile* [Burger], which required access to a Compaq Alpha system due to tool limitations. For the SPEC CPU2000, the train input set was used. Of the benchmarks that compiled properly and completed successfully under *sim-profile*, the ones that contained more than 100 integer multiplies were chosen. (Some of the MediaBench suite would not compile and some of the SPEC CPU2000 would not run to completion under *sim-profile* due to limited system call support for signals.) We then created *sim-eio* [Burger] traces (on a Solaris x86 system) to stub system calls and allow the simulation to be run on SPARC systems.

5 Experimental results

Benchmark	(a) number of original instructions	(b) number of multiplies	(c) average number of emulation instructions per multiply	(d) scaling factor
depend	21,500,000	3,583,286	191.00	0.9695
independ	21,500,000	7,129,408	109.06	0.9731
crafty	26,400,000	134,228	104.25	0.3464
cjpeg	17,617,982	14,215	92.91	0.0697
djpeg	4,649,037	23,439	32.56	0.1410
epic	9,439,277	243,691	64.79	0.6258
gzip	447,300,000	24,344	37.57	0.0020
mesa	24,500,000	800,964	29.62	0.4919
parser	24,400,000	30,217	11.14	0.0136
swim	126,526,737	9,678	110.09	0.0084
twolf	11,600,000	27,145	126.31	0.2281
vortex	30,300,000	22,803	90.70	0.0639

Table 1: Characteristics of benchmarks.

Table 1 summarizes some characteristics of the benchmarks used. Column (a) shows the number of dynamic instructions executed per benchmark. Only a handful of these (cjpeg, djpeg, epic, swim) represent the complete execution. For the other cases we chose a cut-off point based on the execution time of the simulation and the number of multiply instructions encountered. In two cases (parser, twolf), a bug in our simulator limited the number of instructions simulated.

Column (b) shows the number of dynamic multiply instructions in the original program. Column (c) shows the average number of instructions required to emulate each multiply instruction. Because both of these numbers vary so much, we compute a scaling factor to normalize the results presented below (in Figure 3). This scaling factor, shown in column (d) of Table 1, is computed as the proportion of the dynamic instructions that were part of the integer multiply emulation threads.

Figure 3 graphs the slowdown of each of the four emulation modes relative to a base execution, where multiply instructions are executed in a hardware integer multiply unit, scaled by the scaling factor described above. Note that for the two microbenchmarks (*depend* and *independ*), we have further scaled the numbers down by 20 (to fit in the graph).

The important observation to be made about this graph is the relative slowdown among the four emulation modes. In most cases the performance of the four modes is as one might expect (or hope): “squash” does the worst, “pause” is better, and the two “ooo” modes are better yet. However, the “ooo4” is not always better than “ooo2”: in five of the cases it improved very little or not at all, and in three cases it actually performed worse.

Figure 4 sheds some light on why this happens. For *crafty*, *cjpeg*, *parser*, *twolf*, and *vortex*, there is almost never more than one active multiply thread at any time. For these benchmarks, multiply instructions are sufficiently sparse that limiting emulation threads to one is adequate. For the cases where “ooo4” did worse than “ooo2” (*independ*, *epic*, and *swim*), we speculate that this might be due to the main thread occupying most of the decode buffer. For the *independ* microbenchmark, however, the slowdown may be due to the multiply threads themselves competing with each other.

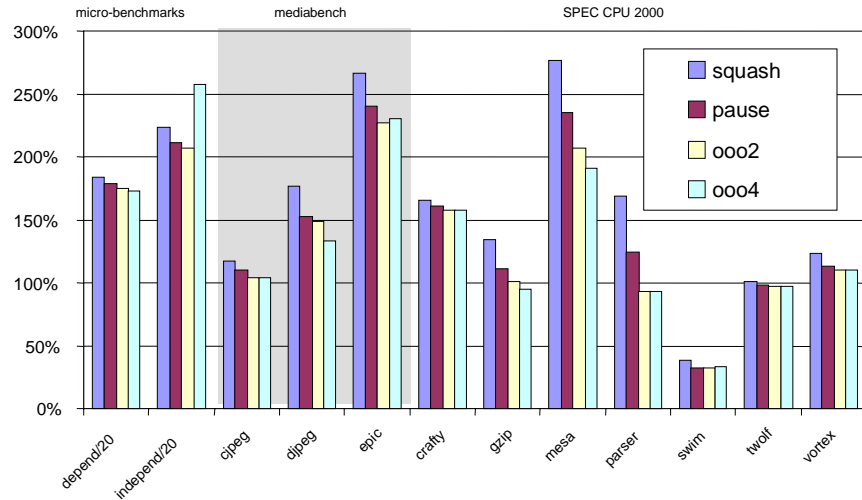


Figure 3: Emulation slowdown, scaled to proportion of multiply instructions in program

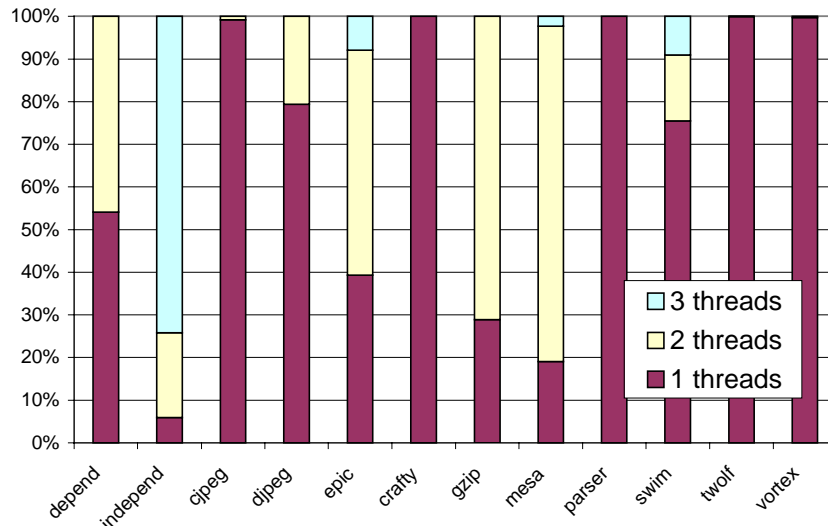


Figure 4: percentage of execution time with n emulation threads active in "ooo4" mode, when at least one is active.

6 Future Work

Further study on emulating instructions in an SMT processor is needed. Our simulation artificially constrained the main thread's use of the decode stage to avoid deadlock. More intelligent means should be sought. One possible approach is to squash enough instructions to open the decode stage to the emulation thread.

Two overheads associated with our approach could be studied. We allow speculative execution of unimplemented instructions. When a branch mispredict occurs for the path that

started the emulation thread, the emulation thread must be squashed. This can be wasteful if mispredicts occurs frequently. A full study of how often mispredicts occur and solutions to the mispredict problem would be helpful. The other overhead is contention for functional units between the main thread or threads and emulation threads. We were not able to study how often dispatch is stalled due to full reservation stations for functional units. Scalability of our approach is an area of further study.

In our experiments, we only considered a very limited set of applications. More applications should be considered. Multithreaded applications, which we did not consider, are of important note.

Finally, we did not consider the operating system or multiple, concurrent applications. SMT processors have a limited number of hardware thread contexts, typically eight or less. When the hardware contexts are exhausted, the operating system or other privileged software must save the hardware context to memory and schedule a new thread. The use of additional threads for emulation may incur more software context switches, thus degrading performance.

7 Conclusion

We have shown that using an SMT processor can reduce the performance cost of emulating unimplemented instructions by not squashing instructions already fetched. Our experiments show that allowing the original thread to continue executing while the emulation thread runs, as exhibited by the “ooo2” and “ooo4” modes, improves performance over the “pause” mode. However, allowing more than one concurrent emulation thread to run (the “ooo4” mode) isn’t always better, because (i) some programs do not have enough multiply instructions to take advantage of the extra threads, and (ii) there may be contention for resources among multiply instructions.

Acknowledgements

We would like to thank Craig Zilles for use of his simulator, an overview of the simulator’s function, and access to neufchatel.cs.wisc.edu, a Solaris X86 system used by the Multiscalar group. Our thanks also go to Mark Hill for use of paul.cs.wisc.edu, a Compaq Alpha system available to students and researchers outside of the Multifacet research group.

References

- [Altman] E. R. Altman, D. Kaeli, Y. Sheffer, “Welcome to the Opportunities of Binary Translation,” IEEE Computer, March 2000, pp. 40-45.
- [Alverson] R. Alverson, D. Callahn, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, “The Tera computer system,” 17th Annual International Symposium on Computer Architecture, May 1990.
- [Burger] D. C. Burger, T. M. Austin, “The Simple Scalar Tool, Version 2.0,” Technical Report CS-TR-97-1342, University of Wisconsin-Madison, June 1997.
- [Chappell] R. Chappell, J. Stark, S. Kim, S. Reinhardt, Y. Patt, “Simultaneous Subordinate Microthreading (SSMT),” Proceedings of the 26th Annual International Symposium on Computer Architecture, May 1999.
- [Lipasti] M. Lipasti and J. P. Shen, “Advanced Computer Architecture I lecture notes,” spring 2000, University of Wisconsin-Madison.
- [MediaBench] <http://www.cs.ucla.edu/~leec/mediabench/>
- [SPEC CPU2000] <http://www.spec.org/osg/cpu2000/>

- [Storino] S. Storino, A. Aipperspach, J. Borkenhagen, R. Eickemeyer, S. Kunkel, S. Levenstein, G. Uhlmann, "A Commercial Multi-Threaded RISC Processor," Proceedings of the 23rd Annual International Symposium on Computer Architecture, May 1996.
- [Tullsen] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor"
- [Yeager] K. C. Yeager, "The MIPS R10000 Superscalar Microprocessor," IEEE Micro, 16, 2, April 1996, pp. 28-40.
- [Zheng] C. Zheng and C. Thompson, "PA-RISC to IA-64 Transparent Execution, No Recompilation." IEEE Computer, March 2000, pp. 47-52.
- [Zilles] C. B. Zilles, J. S. Emer, and G. S. Sohi, "The Use of Multithreading for Exception Handling," Proceedings of Micro-32