ACM Computer Architecture News **19**, 2 (December 1991), 4-8.

Precise Instruction Scheduling without a Precise Machine Model

Henry G. Baker

Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436
(818) 986-1436 (818) 986-1360 FAX

A simple technique is presented which allows an optimizing compiler to more precisely compare the performance of alternative instruction sequences on a complex RISC architecture so that the better sequence can be chosen. This technique may be faster than current techniques, and has the advantage that minor modifications to the hardware do not require any changes to the compiler (not even recompilation), and yet have an immediate effect on instruction scheduling decisions.

INTRODUCTION

Modern reduced-instruction set computer ("RISC") architectures use pipelining and overlapping techniques in order to improve the performance of a serial instruction stream. While these techniques can dramatically improve the performance of many programs, the complexity of these architectures places great demands upon high-level language compilers to *schedule* instructions in such a way that they can be executed in an efficient manner on such RISC architectures. The job of such a compiler is to compare alternative orderings of instruction sequences to find one which is faster than most of the other alternatives. Of course, choosing the absolute *best* sequence is NP-complete [Hennessy83], so heuristics are used to choose a good sequence within the time allowed for compilation. The issue of compilation time is significant; some compiler vendors already acknowledge in their documentation that requesting optimized instruction scheduling can significantly increase compilation times.

The optimizing compiler writer cannot ignore the importance of instruction scheduling, because poor instruction scheduling can completely mask the effect of many other compiler optimizations. For example, optimizations on integer operations such as constant propagation and integer register allocations have the effect of reducing the number of integer operations executed, but if these operations are already scheduled between floating point operations of much longer latencies, then these integer optimizations will provide no performance improvement at all.

With a good machine model, the scheduling problem is not easy, but intuitive heuristics can achieve satisfactory results. If the machine utilizes register scoreboarding, but otherwise hides pipelining, then the compiler estimates the time required for an operation to complete, and incorporates this latency on the edges in the def-use graph constructed by the compiler [Hennessy83] [Bradlee91]. If, on the other hand, the machine exposes the pipeline to the compiler, then the latencies appear deterministic, but the scheduling requires that pipeline conflicts be avoided through techniques like *reservation tables* [Kogge81]. Of course, both scoreboarding and pipelining models hide the real truth. The conflicts that are exposed in the pipeline model are also present in the scoreboard model, but the model ignores this. Furthermore, there are additional conflicts that are hidden in the pipeline model in order to achieve a simpler programming model; these additional conflicts can be resolved only by freezing part or all of the machine, if the programming model is to be preserved. If the performance model used by a compiler deviates in important ways from the performance of the real hardware, then the compiler will not be able to properly optimize.

> The presence of pipeline interlocks ... may change the accuracy of the usual metrics employed to choose between two alternative instruction streams. ... For example, ignoring the effect of pipeline interlocks, [the timing of] memory-memory and register-register operations may differ [only a little]. However, when the interlocks are considered, using ... a memory reference tends to produce [less optimized] code. [Hennessey83].

Thus, an optimizing compiler must have a good machine model in order to intelligently schedule instructions, because it must be able to compare the relative cost of alternative instruction sequences and choose the better ones. With the newest generation of RISC architectures, however, these good machine models are becoming difficult to construct, because the multiple functional units, pipelines and caches interact in extremely complex and difficult-to-predict ways. In addition to any programmer-visible pipelining, an intelligent compiler should also be aware of a variety of "wait states" and "freeze conditions", which are extremely difficult to model and/or are not well-documented. For example, 24 pages in the TMS320C30 documentation [TI88] are devoted to explaining the conditions under which freezes may occur; it is unlikely that any compiler can reasonably model this level of detail. Even worse, the number of "wait states" and "freeze conditions" may vary over different members of the architecture family, and among the different versions (ECO levels, or "steppings", e.g., A-step, B-step, etc.) of the same processor chip. Part of the reason for this is that some wait states and freeze conditions are introduced to correct bugs due to certain unanticipated worst-case gate delays.[1] Even when an accurate CPU model is available, the characteristics of the memory system in which it is embedded can affect performance by up to an order of magnitude [Scott90] [Moyer91].

The net result is that some processor vendors cannot predict the performance of an instruction sequence without actually running the sequence on a real chip—their own software simulators do not capture the full complexity of their machine, and/or this software is often one or more generations behind the chip architecture actually being delivered. It is therefore no wonder that optimizing compilers often do not wring the best performance out of RISC architectures—they do not have up-to-date and accurate machine models on which to base intelligent code generation decisions.

SELF-SIMULATION

The technique we propose for comparing the performance of alternative instruction sequences is embarrassingly simple—*have the compiler actually execute the instruction sequences and see how fast they run!* Since a compiler usually schedules only within a basic block, and since basic blocks tend to consist of at most a few tens of instructions, each such experiment should require less than 1 µsecond to execute on a modern RISC architecture. Of course, it may require several µseconds to set up the experiment, but it is probable that estimating the duration of the instruction sequence by any other means would take as long. If one has access to an MIMD parallel architecture (e.g., the Alliant Computer Systems' i860™-based FX2800 series), then a number of such experiments may be run in parallel.[2]

We call our technique "scheduling through self-simulation", and it obviously works only for "self-hosted" compilers—not cross-compilers. But since the vast majority of compilers are self-hosted, the technique should have wide applicability. Because the program is being timed on the same machine on which it will later execute, any scheduling decisions will be based on completely up-to-date information about *this* processor chip and memory system, and not upon some obsolete simulation program which has not been brought up to current revision level.

---

[1]Freeze conditions that stop the entire machine are more easily modelled by a compiler than freeze conditions that only affect certain functional units. Modern architectures at least uphold a standard programming model; on older architectures, the legality of microcode sequences often varied from individual machine to individual machine, with only the standard microcode sequence guaranteed to work correctly on all machines [Baker79] [Baker80] !

[2]Such parallel self-simulations would truly represent a "race condition"!

Our notion of self-simulation is quite similar to that of Massalin's "superoptimizer", which finds the optimal sequence for a basic block by exhaustively executing all possible instruction sequences [Massalin87]. We are not suggesting that all possible instruction sequences be tested and timed, nor are we even suggesting that all correct instruction sequences be timed, since these searches are hopelessly exponential. We suggest that the compiler generate a small handful of good candidate sequences—using a crude machine performance model—and then time these sequences precisely to reduce the chances of picking a sequence which is substantially worse than average.[3] For example, it appears useful to compare different associations and commutations when compiling an arithmetic expression; the number of different associations is typically small, and the architecture may favor one kind of association over others. For example, the Intel i860™ floating point multiply instruction is commutative in its operands with respect to the resulting product, but requires a long "setup time" on its first operand. This instruction is therefore most efficient when its first operand has "settled" for an entire clock period.

ARCHITECTURAL IMPLICATIONS

Self-simulation places certain demands upon a RISC architecture. Obviously, one must be able to accurately time very short instruction sequences, which requires an extremely high resolution timer. A 16-bit counter attached directly to the processor oscillator would provide the required resolution with an order of magnitude more range than we need for the types of experiments envisioned. The time to read this clock should be a small constant number of cycles, so that the timing of the instruction sequence can be accurate. The required circuitry would require a miniscule amount of space on a modern processor chip, which would make this clock quickly accessible as a machine register. The TMS320C30 [TI88] has an on-chip memory-mapped 32-bit timer which is appropriate for this purpose; the ATT DSP32C [ATT88] does not have a built-in timer, but its serial port could be used to time very short (<32 ticks) sequences.[4]

A much bigger problem in self-simulation is the setting up of an appropriate context in which to execute the experiment. The instruction and data caches have to be loaded with appropriate contents, and the registers and pipelines have to be initialized. Given the lengths of the sequences we envision, it is not necessary to replace the entire instruction or data cache, but only to make sure that the instructions and data needed for the experiment are in the cache.[5] Similarly, not every bit of programmer-visible state in the machine will matter to the experiment, so only the registers and pipelines that matter need be initialized. In most RISC architectures, the timing of an instruction is oblivious to the actual values of its arguments—assuming that they do not cause an exception—and in these cases the registers may not have to be initialized at all.

A potentially more significant problem is the ability of the underlying architecture to quickly change from writing a portion of memory (during the construction of the experiment) to the execution of that portion of memory.[6] The hardware protection scheme of some computer systems—e.g., Multics—completely rules out the possibility of immediately executing constructed code. Other systems—e.g., IBM 7090, 370—offer a "hook" in the form of an "execute" instruction, which, while sufficing for some applications, its execution of only a single instruction makes the accurate timing of instruction *sequences* impossible.

---

[3]For high-volume ROM-able signal processing applications, like those developed for the TMS320C30, it may very well make economic sense to try *all* sequences, including Massalin's exhaustive search for non-intuitive sequences.

[4]Other current chips may already have appropriate clocks as "undocumented hardware debugging circuitry".

[5]The appropriate loading of the instruction cache may be problematical in some architectures, where the instruction cache can only be loaded through actually executing the instructions! Andrew Appel [Appel91] suggests that the sequence be executed twice—the first time to load the instruction/data caches, the second time to time the sequence.

[6]These issues are more extensively addressed in [Keppel91].

Even when possible, a change of a portion of memory from "data" to "instruction" requires a change to the page map, which may require flushing the translation lookaside buffer, as well as the data and instruction caches. In the Intel i860XR [Intel89], for example, the portion of the data cache in which the instruction sequence is constructed must be flushed and the entire instruction cache invalidated before the experiment can be performed. Unfortunately, the cache flush and instruction invalidation can be performed only in supervisor mode. A slightly better alternative is to locate the experimental instruction sequence on a "non-cacheable" page, which eliminates the need for the cache flush, but not the need for the instruction cache invalidate. As a result, the cost of an experiment could grow to hundreds or thousands of μseconds with an inappropriate architecture. Luckily, this problem is already being highlighted by advanced "object-oriented" programming languages, which incrementally compile methods (subprograms) "on-the-fly", as the classes of the arguments become known [Deutsch84] [Chambers89a,b].

The TMS320C30 [TI88] can bypass its instruction cache and execute instructions out of internal RAM; since this RAM is the same speed and latency as the instruction cache, it would seem that using this RAM for storing instruction sequences would be ideal. It is likely, however, that the instruction sequence being timed will also require access to data in this internal RAM, and the timed sequence would therefore run slowly due to memory conflicts.[7] On the other hand, the 64-word instruction cache can be wholly invalidated by a single instruction (there is no supervisor mode), so the TMS320C30's instruction cache can be easily and efficiently loaded.

Some instruction cache problems can be finessed by performing the experiments on a different processor from the one executing the optimizing compiler. If the controlling processor could start and stop the clock of the self-simulating processor, as is possible with "in circuit emulators", then the controlling processor could place its experiments in different locations in memory which happen not to be in the instruction or data caches, and therefore these locations can be loaded while the self-simulating processor is stopped. So long as no change is necessary in the virtual memory map, the translation buffer need not be reloaded. The self-simulating processor could then execute the sequence quite quickly without incurring the disastrous overheads.[8]

The newer Intel i860XP [Intel91] has a "snoopy" instruction cache, and can also declare data cache pages to be "write-through"; the combination of these features should allow timing experiments wholly within user mode Unfortunately, the i860XP's instruction cache apparently snoops on (and therefore is invalidated by) only *externally* generated bus cycles, such as those from another processor. Thus, one can either utilize a second processor for the experiments, or utilize some sort of external DMA device which copies the instruction sequence into the executable area. Because the DMA device forces instruction cache snooping, it can invalidate the relevant portion of the i860XP instruction cache more efficiently than the i860XP itself can, because the i860XP can only invalidate the *whole* cache, while a snoop can invalidate a *single* cache line! Many high-speed DMA devices—e.g., disk and network controllers—have a self-test "loop-back" mode which is ideal for this sort of DMA activity. Unfortunately, accessing these devices is also likely to involve supervisor mode.

CONCLUSIONS

We have described a technique called *self-simulation* which can be used by an optimizing compiler to compare more accurately the performance of alternative instruction sequences. Because self-simulation times the sequences on its own actual hardware, there is less possibility of the compiler

---

[7]The effect of memory conflicts on speed is one of the major reasons for performing these precise timings!

[8]Apollo Computer's first workstation utilized *two* microprocessors—one to execute the user's program, and one to handle the page faults—due to the MC68000's inability to properly handle page faults. This use of multiple processors can be called the "dumb ethnic" strategy, due to its resemblence to a jokes about the number of ethnic persons required to install a light bulb.

becoming "out-of-sync" with the current processor chip revision level. The hardware and software architectural requirements for efficient self-simulation are not strenuous; however, these requirements are not completely met by many current architectures. Luckily, a number of other programming techniques have the same requirements, so it is likely that future architectures will be more amenable to self-simulation. Processor architects who are disturbed by our conclusions should keep them in mind during their next design.

ACKNOWLEDGEMENTS

Many thanks to A. Appel and D. Keppel for their helpful comments on early drafts of this paper.

REFERENCES

Appel, Andrew. Private communication, July, 1991.

ATT. *WE® DSP32C Digital Signal Processor Advance Data Sheet*. ATT Microelectronics, Allentown, PA, May 1988.

Baker, Henry, and Parker, Clinton. *Micro SPL*. Synapse Computer Services, Sept. 1979.

Baker, Henry, and Parker, Clinton. "High Level Language Programs Run Ten Times Faster in Microstore". Tech. Rept., Synapse Computer Services, 1980.

Bradlee, David G., *et al.* "The Marion System for Retargetable Instruction Scheduling". *Proc. ACM PLDI'91, Sigplan Not. 26*,6 (June 1991),229-240.

Brooks, F.P. "The Execute Operations — A Fourth Mode of Instruction Sequencing". *CACM 3*,3 (March 1960),168-170.

Chambers, C., and Ungar, D. "Customization: Optimizing Compiler Technology for SELF, A Dynamically-Typed Object-Oriented Programming Language". *Proc. ACM PLDI'89, Sigplan Not. 24*,7 (July 1989),146-160.

Chambers, C., Ungar, D., and Lee, E. "An Efficient Implementation of SELF, A Dynamically-Typed Object-Oriented Programming Language". *Proc. OOPSLA'89, Sigplan Not. 24*,10 (Oct. 1989),49-70.

Deutsch, L.P., and Schiffman, A.M. "Efficient Implementation of the Smalltalk-80 System". *Proc. 11'th ACM POPL*, Salt Lake City, UT, Jan. 1984, 297-302.

Ellis, John R. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, Cambridge, MA, 1986.

Gibbons, P.B., and Muchnick, S.S. "Efficient instruction scheduling for a pipelined architecture". *Proc. ACM Symp. on Compiler Constr., Sigplan Not. 21*,7 (July 1986),11-16.

Hennessy, John, and Gross, Thomas. "Postpass Code Optimization of Pipeline Constraints". *ACM TOPLAS 5*,3 (July 1983),422-448.

Intel Corp. *i860™ [XR] 64-Bit Microprocessor Programmer's Reference Manual*. #240329-002, 1989.

Intel Corp. *i860™ Microprocessor Family Programmer's Reference Manual*. #240875-001, 1991.

Intel Corp. *i860™ 64-bit Microprocessor Simulator and Debugger Reference Manual, Ver. 3*. #240437-003, Jan. 1990.

Keppel, David. "A Portable Interface for On-The-Fly Instruction Space Modification". *Proc. 4'th ACM ASPLOS, Sigplan Not. 26*,4 (April 1991),86-95.

Knuth, Donald E. *The Art of Computer Programming Vol. I: Fundamental Algorithms, 2nd Ed*. Addison-Wesley, Reading, MA, 1973,634p.

Kogge, P.M. *The Architecture of Pipelined Computers*. McGraw-Hill, New York, 1981.

Massalin, Henry. "Superoptimizer—A Look at the Smallest Program". *Proc. ACM ASPLOS'87, Sigplan Not. 22*,10 (Oct. 1987),122-126.

Morris, W.G. "CCG: A Prototype Coagulating Code Generator". *Proc. ACM PLDI'91, Sigplan Not. 26*,6 (June 1991),45-58.

Moyer, Steven A. "Performance of the iPSC/860 Node Architecture". IPC-TR-91-007, Inst. for Parallel Comp., Eng. & Applied Sci., U. of Va., May 1991.

Scott, D.S., and Withers, G.R. "Performance and Assembly Language Programming of the iPSC/860 System". Tech. Report, Intel Supercomputer Systems Div., Beaverton, OR, 1990.

Texas Inst. *TMS320C30: The Third Generation of the TMS320 Family of Digital Signal Processors*. Texas Instruments, Feb. 1988.

Wirth, Niklaus. "From Programming Language Design to Computer Construction". *CACM 28*,2 (Feb. 1985),160-164.

Xerox Corp. *ALTO: A Personal Computer System Hardware Manual.* Xerox PARC, Palo Alto, CA, Jan. 1977.

