Evolutionary Compilation to Long Instruction Superscalar Microarchitectures for Exploiting Parallelism At All Levels

Thomas M. Conte
Department of Electrical and Computer Engineering
North Carolina State University
conte@ncsu.edu, http://www.ece.ncsu.edu/tinker

The problem

Independence between tasks occurs at a wide range of distances. Exploiting it is relatively easy when all run-time behavior is entirely predictable. The run time behavior includes (1) *branch direction and target location*, (2) *load latencies (cache hit/miss)*, and (3) *memory dependencies.* Consider the canonical solutions today, shown in Table 1.

Table 1: Exploiting parallelism today

Time	*Distance*	*Predictability solutions*
At compile time	One source file	Profiling, static estimates, static memory disambiguation
At instruction fetch time	Size of hardware window	Branch prediction hardware, dependence prediction

VLIW seeks to perform all prediction before run-time, if possible. Superscalar hopes it can get reasonable performance even if the compiler disappears. Let's assume the solution includes dynamic scheduling hardware (superscalar), but with explicit parallelism expressed in the ISA (i.e., an EPIC ISA). The compiler's predictions via profiling are input dependent, and are assumed to hold true for the entire life of the program. Common sense tells us that programs are used differently even by the same user (say as the user transitions to a "power user"). The hardware predictions are very accurate, but the potential parallelism exploited by them is limited to a small distance (the size of the hardware window). So the compiler has the right scope (large), but poor prediction. The hardware has great prediction, but terrible scope. There has been a war raging between the two approaches. Instead of warring, it is tempting to try to combine the two. I propose that we add some levels to the current, rather stunted hierarchy of parallelism exploitation.

Our suggestion

Consider Table 1 again. What other times does the machine handle code? The answers and their associated levels are shown in Table 2.

Table 2: Some new times to exploit parallelism.

Time	*Distance*	*Predictability solutions*
At compile time	One source file	Profiling, static estimates, static memory disambiguation
At program exit	**Entire executable**	**Recent profile data from current run**
At page faults	**Size of a page**	**Recent profile data from last run**
At Icache misses	**Size of a miss repair fetch unit**	**Recent profile data from last replacement**
At instruction fetch	Size of hardware window	Branch prediction hardware, dependence prediction

A program can now *evolve* its code over time based on the most recent past behavior of branches, loads and memory dependencies. (There is actually another level, rescheduling code while it remains dormant between executions—'flossing' the executable, if you will.) A key challenge is the collection of profiles *all the time.* That implies it shouldn't impact performance at all. This rules out software profiling. We introduced techniques to allow branch predictors [1] or performance monitors [2],[3] to collect profile data in real time in hardware, without significant overhead. These hardware buffers are actually performing prediction of events, just like branch and memory dependence predictors. But they are predicting events for future runs of the code, rather than current runs. This addresses the problem of the user becoming a "power user". The system can now react to changes in the compiler's assumptions and re-compile the affected code.

How can parallelism be extracted at page fault time? Techniques such as DAISY [4] have demonstrated it is possible to convert from Java to scheduled VLIW code at page fault time. Sumedh Sathaye looked at techniques to reschedule VLIW instructions at page fault time for code compatibility [5],[6] and for performance [6].

How can parallelism be extracted at Icache miss time? Nair and Hopkins [7] suggested a technique they called DIF. We proposed a hardware technique that is similar in spirit but different in implementation called *miss path scheduling* [8]. These are just early studies, and more work needs to be done here.

The architecture of an evolutionary processor

What does this mean for architecture? One view is that the compiler will dissolve into the operating system (i.e., page-fault-time or program-exit-time scheduling) and the memory hierarchy (i.e., page-fault-time or miss-path scheduling). The microarchitecture needs new profiling hardware to record events for the future. It may need new structures to allow the compilation to proceed with minimal overhead. Furthermore, the ISA should contain as many hints to the microarchitecture as possible, whereas the microarchitecture should exploit dynamic parallelism as much as possible. To introduce a new buzzword, let's call this a *Long Instruction Super-Scalar* architecture (*LISS*). Open questions include: (1) If code is being rescheduled, does the instruction set format have to remain rigid (what is an ISA for)? (2) What does this mean for code compatibility? (3) Can the OS or hardware do more than just reschedule code (i.e., code transformations as in [9])? (4) Can users ever trust such a system to not introduce new bugs into their code? (As for the last question-- we wisely didn't tell users about the complexity of speculative superscalar microarchitectures.)

References

[1] T. M. Conte, B. A. Patel, and J. S. Cox, "Using branch handling hardware to support profile-driven optimization," in *Proc. 27th Annual International Symposium on Microarchitecture*, (San Jose, CA), Dec. 1994.

[2] T. M. Conte, K. N. Menezes and M. A. Hirsch, "Accurate and practical profile-driven compilation using the profile buffer," in *Proc. the 29th Annual International Symposium on Microarchitecture*, (Paris, France), Nov. 1996.

[3] K. N. P. Menezes, "Hardware-based profiling for program optimization," Ph.D. thesis, Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, North Carolina, 1997.

[4] Eric Altman and Kemal Ebcioglu, "DAISY: Dynamic compilation for 100 architectural compatibility," *Proc. 24th International Symposium on Computer Architecture*, (Denver, CO), June, 1997.

[5] T. M. Conte and S. W. Sathaye, "Dynamic rescheduling: A technique for object code compatibility in VLIW architectures," in *Proc. 28th Annual International Symposium on Microarchitecture*, (Ann Arbor, MI), Nov. 1995.

[6] S. W. Sathaye, "Evolutionary compilation for code compatibility and performance," Ph.D. thesis, Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, North Carolina, 1998.

[7] R. Nair and M. E. Hopkins, "Exploting instruction level parallelism in processors by caching scheduled groups," *Proc. 24th International Symposium on Computer Architecture*, (Denver, CO), June, 1997.

[8] S. Banerjia, S. W. Sathaye, K. N. Menezes and T. M. Conte, "MPS: Miss path scheduling for multiple-issue processors," *IEEE Transactions on Computers (to appear)*, 1998.

[9] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad "Fast, Effective Dynamic Compilation", Proc. SIGPLAN Conf. On Programming Languages Design and Implementation, June 1996.