Author Retrospective for
Software Trace Cache

Alex Ramirez[1,2], Ayose J. Falcón[3], Oliverio J. Santana[4], Mateo Valero[1,2]
[1] Universitat Politècnica de Catalunya, BarcelonaTech
[2] Barcelona Supercomputing Center
[3] Intel Labs, Barcelona
[4] Universidad de Las Palmas de Gran Canaria
{aramirez@ac.upc.edu}

ABSTRACT
In superscalar processors, capable of issuing and executing multiple instructions per cycle, fetch performance represents an upper bound to the overall processor performance. Unless there is some form of instruction re-use mechanism, you cannot execute instructions faster than you can fetch them.

Instruction Level Parallelism, represented by wide issue out of order superscalar processors, was the trending topic during the end of the 90's and early 2000's. It is indeed the most promising way to continue improving processor performance in a way that does not impact application development, unlike current multicore architectures which require parallelizing the applications (a process that is still far from being automated in the general case). Widening superscalar processor issue was the promise of never-ending improvements to single thread performance, as identified by Yale N. Patt et al. in the 1997 special issue of IEEE Computer about "Billion transistor processors" [1].

However, instruction fetch performance is limited by the control flow of the program. The basic fetch stage implementation can read instructions from a single cache line, starting from the current fetch address and up to the next control flow instruction. That is one basic block per cycle at most.

Given that the typical basic block size in SPEC integer benchmarks is 4-6 instructions, fetch performance was limited to those same 4-6 instructions per cycle, making 8-wide and 16-wide superscalar processors impractical. It became imperative to find mechanisms to fetch more than 8 instructions per cycle, and that meant fetching more than one basic block per cycle.

The Trace Cache [2] [3] [4] quickly established itself as the state of the art in high performance instruction fetch. The trace cache relies on a trace building mechanism that dynamically reorders the control flow of the program, and stores the dynamic instruction sequences in sequential storage, increasing fetch width.

However, it is a complex hardware structure that adds not only another cache memory to the on-chip storage hierarchy, but also requires a branch predictor capable of issuing multiple predictions per cycle to index the contents of the trace cache, and distinguish between multiple dynamic sequences of instructions.

ICS 25th Anniversary Volume. 2014.
ACM 978-1-4503-2840-1/14/06.
http://dx.doi.org/10.1145/2591635.2594508

The Software Trace Cache is a compiler transformation, or a post-compilation binary optimization, that extends the seminar work of Pettis and Hansen PLDI'90 to perform the reordering of the dynamic instruction stream into sequential memory locations using profile information from previous executions. The major advantage compared to the trace cache, is that it is a software optimization, and does not require additional hardware to capture the dynamic instruction stream, nor additional memories to store them. Instructions are still captured in the regular instruction cache. The major disadvantage is that it can only capture one of the dynamic instruction sequences to store it sequentially. Any other control flow will result in taken branches, and interruptions of the fetch stream.

Our results show that fetch width using STC was competitive with that obtained with the hardware TC, and was applicable to a wide range of superscalar architectures, because it does not require hardware changes to enable fetching from multiple basic blocks in a single cycle, even if only one branch prediction can be issued. Any front-end architecture built on a BTB that ignores branches as long as they are not taken, will automatically treat such branches as NOP instructions in terms of fetch [5].

This was only the beginning. The impact of this optimization on fetch and superscalar processor architectures went much further than the original ICS paper in 1999

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors. Compilers. Code generation. Optimization.

General Terms
Performance, Experimentation.

Keywords
Binary trabslation, ILP. Superscalar processors. Instruction fetch.

1. IMPACT
After the first STC presentation, the first author, Alex Ramirez, was invited for a summer internship in Digital Equipment Corporation's Western Research Lab.((by the time the internship actually happened, DEC and its Research Labs had been acquired by Compaq)) in Palo Alto (CA) in order to implement the STC algorithm in their SPIKE binary optimizer.

This work actually expanded two consecutive summers (1999 and 2000). The first one was required to finalize the implementation, and perform initial simulation studies. The second one allowed finalizing the implementation and performing tests on real workloads and systems.

Using SPIKE we optimized the performance of both a commercial DBMS (Oracle) and the operating system kernel, improving TPC benchmark results on Alpha platforms.

Similar code layout optimizations [6] were implemented in the IBM FDPR binary optimizer, and the STC algorithm is implemented in the Control Flow branch of the GNU gcc compiler [7].

Although 16-wide superscalar processors never happened (so far), we have seen other wide-issue superscalars such as the IBM Power7, and Intel SandyBridge or Haswell, where fetch performance is a potential bottleneck. These architectures have implemented solutions to enable fetching of multiple basic blocks per cycle, from the standard instruction cache, and would greatly benefit from optimizations such as the STC.

Also, fetch performance and fetch scheduling become even more important in simultaneous multithreaded architectures (which also exploit wide-issue superscalar pipelines), such as the IBM Power5 (and its successors), or the AMD Bulldozer. Again, improving the layout of the code could lead to both simpler microarchitectures and better performance.

The hardware trace cache has been actually implemented in the Intel Pentium 4 and Pentium D NetBurst architectures, from Willamette to Presier. However it was widely acknowledged that the main purpose was to store and fetch decoded micro-ops rather than increasing fetch width. As such, it is more closely related to the Decoded Instruction Area [8] or the rePLay framework [9] than to the STC.

2. RELATED AND FOLLOW-UP WORK

There were several papers that derived from this original Software Trace Cache paper. A slightly different version of the code layout algorithm was developed focusing on database workloads, which traverse a very large instruction working set, with lots of subroutine calls and low branch predictability in ICPP'99 [10], with a journal archive version in IJPP'02 [11]. An improved version of the fetch stage microarchitecture avoiding redundancy between software traces in the instruction cache, and dynamic traces in the trace cache was published in HPCA'00 [12].

The final, archive version of the algorithm with new heuristics to totally automate the process, was published in IEEE Transactions on Computers in 2004 [12].

Also, several papers explored in depth analysis of the impact of the STC optimizations in the SPIKE optimizer. A detailed analysis on the instruction cache and fetch performance on Oracle database was published in ISCA'01 [13], showing not only improvements on the instruction cache, but also a mush tighter packaging of useful instructions into cache lines, and a reduction in taken branches that leads to improved fetch width. A detailed analysis on the impact on branch prediction accuracy for various prediction strategies was published in PACT'00 [14], showing that since most branches behave the same way (usually not taken), their interference in the branch prediction tables was constructive, not destructive, achieving the same effect as an Agree prediction strategy.

Finally, based on all these observations, we developed a new fetch stage microarchitecture and a specialized branch predictor that exploit the maximum benefits of these layout optimizations: the instruction stream fetch engine, in MICRO'02 [15].

This fetch architecture for wide superscalar instruction fetch was picked up by the next PhD students in the group, and produced several related papers. A modification of the stream fetch engine for SMT processors was evaluated in HPCA'04 [16], showing relevant impact on the choice of the fetch policy to be used when fetching from a single thread on each cycle. A stream predictor capable of issuing multiple predictions per cycle was shown in ISHP'05 [17]. And a mechanism to store decoded instructions (like the Pentium4 trace cache) in the standard instruction cache was presented in PACT'06 [18] (archive version published in IEEE ToC'09 [19]).

3. DISCUSSION

There have been multiple works, before and after the STC, that have shown the benefits of feedback directed compilation. However, profile feedback is standard practice only in benchmark runs. It has not been adopted by end users in daily usage of computers. This may indicate that we need to change the way we use the compiler and how profile data is generated and stored, to make it easier for users to use them.

At the same time, and for the same reasons, it is very important to develop heuristic analysis techniques that can statically obtain an approximation of the information that would be obtained via profiling, such as the "Branch prediction for free" paper by Ball and Larus in PLDI'93 [20].

Combining profile feedback and heuristics, it is possible to reproduce at compile time many of the complex dynamic mechanisms that we have in out of order execution, leading to significant savings at reduced performance penalties. Even if maximum performance is required, and thus we resort to complex dynamic solutions in hardware, the combination of software and hardware optimizations still leads to improved results.

The combined results of all the STC related research show that code layout optimizations have an impact that goes far beyond a mere reduction in the instruction cache misses.

Even if 16-wide superscalar processors have not been produced yet, there is a significant benefit of wide superscalar instruction fetch in terms of power consumption. As shown by our instruction stream fetch engine, it takes approximately the same effort to fetch 16 instructions in one cycle, than to fetch just 4. That means that we could be fetching instructions only once every 4 cycles, or that we could decouple the clock frequency of the processor front-end from the processor back-end.

All in all, the STC was the seed for a very exciting series of research projects spanning compiler algorithms, superscalar processor microarchitecture, branch prediction, multithreaded processors, and instruction decoding. With the demise of wide issue superscalar processors in favor of multicore processors, interest in this kind of techniques has decreased. However, given the power efficiency difficulties that future processor and multicore architectures are facing, it is likely that techniques originally intended for wide superscalar processors can be revisited focusing on their energy saving potential.

4. ACKNOWLEDGEMENTS

The authors want to thank those industrial partners who expressed interest in the STC-derived technologies (Sun Microsystems, IBM, Intel, Digital Equipment Corporation / Compaq), and specially Luiz Barroso and Kourosh Garachorloo who hosted the first author for two consecutive summers in Compaq's Western Research Lab.

5. REFERENCES

[1] Y. N. Patt y et al., «One Billion Transistors, One Uniprocessor, One Chip,» *IEEE Computer,* pp. 51-58, Sept. 1997.

[2] A. Peleg y U. Weiser, «Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line». United States Patente 5,381,533, 1995.

[3] E. Rottenberg, S. Bennett y J. E. Smith, «Trace cache: a low latency approach to high bandwidth instruction fetching,» de *ACM/IEEE international symposium on Microarchitecture (MICRO 29)*, 1996.

[4] D. H. Friendly, S. J. Patel y Y. N. Patt, «Alternative fetch and issue policies for the trace cache fetch mechanism,» de *30th annual ACM/IEEE international symposium on Microarchitecture (MICRO 30)*, 1997.

[5] B. Calder y D. Grunwald, «Reducing branch costs via branch alignment,» de *6th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994.

[6] K. Pettis y R. C. Hansen, «Profile guided code positioning,» de *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 1990.

[7] «Improving GCC's Infrastructure (Control Flow Graph),» [En línea]. Available: http://www.gnu.org/software/gcc/projects/cfg.html.

[8] O. J. Santana, A. Falcon, A. Ramirez y M. Valero, «Branch predictor guided instruction decoding,» de *15th international conference on Parallel architectures and compilation techniques (PACT '06)*, 2006.

[9] S. J. Patel y S. S. Lumetta, «rePLay: A hardware framework for dynamic optimization,» *IEEE Transactions on Computers,* 2001.

[10] A. Ramirez, J. L. Larriba-Pey, C. Navarro, X. Serrano y M. Valero, «Optimization of Instruction Fetch for Decision Support Workloads,» de *International Conference on Parallel Processing (ICPP)*, 1999.

[11] A. Ramirez, J. L. Larriba-Pey, C. Navarro, M. Valero y J. Torrellas, «Software Trace Cache for Commercial Applications,» *Internatinal Journal of Parallel Programming,* vol. 30, nº 5, pp. 373-395, 2002.

[12] A. Ramirez, J. L. Larriba-Pey y M. Valero, «Trace Cache Redundancy: Red & Blue Traces,» de *Sixth International Symposium on High-Performance Computer Architecture (HPCA)*, 2000.

[13] A. Ramirez, J. L. Larriba-Pey y M. Valero, «Software Trace Cache,» *IEEE Transactions on Computers,* vol. 54, nº 1, pp. 22-35, 2005.

[14] A. Ramirez, L. A. Barroso, K. Gharachorloo, R. Cohn, J. L. Larriba-Pey, G. P. Lowney y M. Valero, «Code layout optimizations for transaction processing workloads,» de *28th Annual International Symposium on Computer Architecture*, 2001.

[15] A. Ramirez, J. L. Larriba-Pey y M. Valero, «The Effect of Code Reordering on Branch Prediction,» de *International Conference on Parallel Architectures and Compilation Techniques*, 2000.

[16] A. Ramirez, O. J. Santana, J. L. Larriba-Pey y M. Valero, «Fetching instruction streams,» de *35th Annual International Symposium on Microarchitecture*, 2002.

[17] A. Falcon, A. Ramirez y M. Valero, «A Low-Complexity, High-Performance Fetch Unit for Simultaneous Multithreading Processors,» de *10th International Conference on High-Performance Computer Architecture*, 2004.

[18] A. Falcon, A. Ramirez y m. Valero, «Tolerating Branch Predictor Latency on SMT,» de *5th International Symposium on High Performance Computing (ISHPC)*, 2005.

[19] O. J. Santana, A. Falcon, A. Ramirez y M. Valero, «Branch predictor guided instruction decoding,» de *15th International Conference on Parallel Architecture and Compilation Techniques*, 2006.

[20] J. S. Oliverio, F. Ayose, R. Alex y V. Mateo, «DIA: A Complexity-Effective Decoding Architecture,» *IEEE Transactions on Computers,* vol. 58, nº 4, pp. 448-462, 2009.

[21] T. Ball y J. R. Larus, «Branch prediction for free,» de *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'93)*, 1993.