On the Performance of Fetch Engines Running DSS Workloads*

Carlos Navarro, Alex Ramírez, Josep-L. Larriba-Pey, and Mateo Valero

Universitat Politècnica de Catalunya
Jordi Girona 1–3, D6
08034 Barcelona (Spain)
{cnavarro, aramirez, larri, mateo}@ac.upc.es

Abstract This paper examines the behavior of current and next generation microprocessors' fetch engines while running Decision Support Systems (DSS) workloads. We analyze the effect of the latency of instructions being fetched, their quality and the number of instructions that the fetch engine provides per access. Our study reveals that a well dimensioned fetch engine is of great importance for DSS performance, showing gains over 100% between a conventional fetch engine and a perfect one. We have found that, in many cases, the I-cache size bounds the benefits that one might expect from a better branch prediction.
The second part of our study focuses on the performance benefits of a code reordering technique for the Database Management System (DBMS) that runs our DSS workload. Our results show that the reordering has a positive effect on the three parameters and can speed-up the DSS execution by 21% for a 4 issue processor, and 27% for an 8 issue one.

1 Introduction

Fetch engine performance is characterized by three different factors: the latency of instructions being fetched from memory, the number of instructions fetched per access, and the quality of those instructions.

Instruction latency is caused by the speed gap between processing and memory. While the pipeline needs new instructions every cycle, the memory can only supply them at a ratio of tens or even hundreds of cycles. Reducing the I-cache misses has been addressed using software and hardware techniques. On the software side we have code reordering techniques, like [7,11]. On the hardware side we have set associative caches, hardware instruction prefetching [17], and victim caches [8], among others.

The number of instructions fetched per cycle, fetch bandwidth, is a problem when more than one basic block has to be provided per cycle. Bandwidth is limited by the execution of non-contiguous instructions and the hardware width

* This research has been supported by CICYT grant TIC-0511-98, the Spanish Ministry of education grant PN98 43443683-1 (Carlos Navarro), the Generalitat de Catalunya grant 1998FI-003060-26 (Alex Ramirez) and CEPBA.

of the fetch engine. The execution of non-contiguous basic blocks has been addressed also, both in hardware and software. The hardware solutions comprise the branch address cache [20], the collapsing buffer [5], and the trace cache [10,15]. Also, in our previous work [12,13] we have addressed this problem with a software code reordering technique, the Software Trace Cache (STC), and the interaction between the software and hardware trace cache [14].

The quality of instructions is determined by branch prediction accuracy. Dynamic branch prediction schemes have been used during the last years to avoid stopping the fetch until branch resolution [16]. But the use of this prediction mechanisms also introduces wrong path instructions in the pipeline whenever a branch is mispredicted. A lot of work has been done during the last years to increase branch prediction accuracy [6,16,21].

DBMSs are highly structured codes, perform many procedure calls and have plenty of control statements to handle all types of error situations. This causes a great level of control flow activity in the code and larger instruction working sets when compared with other common integer codes[1,2]. Recent work has shown that those instruction working sets can be a problem for the instruction cache sizes used in current generation microprocessors [1].

Our results show that for DSS workloads, the fetch engine has a large influence on the IPC. In Figure 1 we compare the performance of a conventional fetch engine against the best fetch engine possible for 4 and 8 issue superscalars. This perfect fetch engine returns the full width of the next correct path instructions every cycle. That is, it does not have latency problems, it uses all the possible bandwidth, and it uses a perfect branch prediction. We can see that the perfect fetch engine shows speedups of 98% and 59% over the 16KB and 32KB engines for the 4 issue processor and of 116% and 70% for the 8 issue processor.

With all this evidence, we study how DBMSs exercise the different parts of the current processors' fetch engines, and how different improvements on the fetch engine can help their execution. We analyze the fetch impact on the overall performance, the isolated behavior of the three main fetch parameters, and their effect on performance. We also analyze the effects of a code reordering technique on each fetch engine component and its effect on performance.

The rest of the paper is structured as follows. Section 2 presents our experimental setup. Sections 3, 4, and 5 study the isolated effect on performance of fetch latency, quality and bandwidth. Section 6 addresses the effects of applying a code reordering scheme to our workload. Finally we conclude in Section 7.

2 Experimental Setup

Our DSS workload runs on top of the PostgresSQL 6.3 DBMS [18]. Our workload is modeled with a session that executes queries 3 and 17 of the Transaction processing council TPC-D benchmark [19].

We set up the Alpha version of the Postgres 6.3 database on digital Unix v4.0 compiled with the -O3 flags of the Digitals's C compiler. A TPC-D database is used with scale factor 0.1 (100MB of raw data).

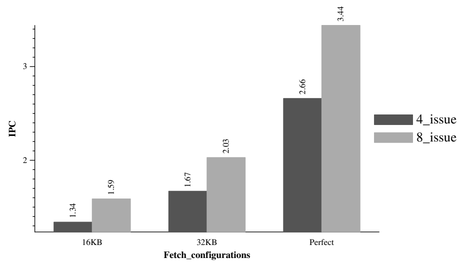

Fig.1. Comparison between two conventional fetch engines and the perfect one for 4 and 8 issue superscalar processors. The experiment shows results for cache sizes of 16KB and 32KB. The conventional fetch engine uses a 2048 entries Bimodal predictor.

The simulator used in this study is based on the sim-outorder timing simulator included in the Simplescalar-Alpha v3.0 tool set [4]. Table 1 shows the configuration we have used for our baseline architecture. The I-cache and branch prediction setups are varied across the study. Since the complete execution requires over a billion instructions, we have used sampling in order to reduce simulation times. Simulations are performed using the following sequence: 50 Million detailed simulation sample followed by a 150 Million fast simulation sample where we just emulate the ISA.

We model a branch misprediction in the following way. When a branch misprediction is detected, the execution engine spends a fixed number of cycles arranging the window and other internals before re-starting the fetch in the correct path. This number of cycles is what we call *misprediction recovery penalty* and it is a simulation parameter. After those cycles, fetch is re-started and the issue of correct path instructions restarts 2 cycles later[9].

In this work we are using two different processor widths, 4 and 8. The fetch engine for the 4 issue processor predicts a single branch per cycle, while for the 8 issue processor it can predict a second branch if the first was not taken.

3 Effect of Instruction Latency on Performance

The first parameter we explore is the latency of the instructions being fetched from memory, and the impact that it has on the overall performance. Our analysis will be structured as follows: First we will show the perceived latency for

Parameters	Values	Parameters	Values
Core		Misprediction recovery penalty	1
Issue width	4,8	*Memory*	
Window size	128	Inst L1	several sizes/ 32 byte/ 2 way
L/S queue size	128	Data L1	64K/ 32 byte/ 2 way
FUs	Unbound fully pipelined	Combined L2	2MB/ 64 byte/ 2 way
Branch prediction		Latencies	1/ 7 / 80
Direction	Several schemes	Instruction TLB	16 4K entries/ 4 way
Address	BTB 4K entries/ 4 way	Data TLB	32 4K entries/ 4way
RAS	128 entries		

Table1. Baseline processor

several cache sizes, and finally we will show how those latencies affect the overall execution performance.

Given that with the current technology trends our L2 latency could be quite aggressive, we will also provide results for a L2 latency of 15 cycles. This latency is more feasible for future processor clock speeds, even with integration of L2 in the processor die. For example, in [3], where a 1GHz processor with integrated 2MB L2 cache is evaluated, the latencies for L2 are larger than 15 cycles.

Our metric for the perceived instruction latency, will be the Average Memory Access Time for instructions (AMAT_i). This metric will show the average latency that our fetch engine perceives per instruction. AMAT_i is computed as: $Icache\ miss\ rate * L2\ latency + (Icache\ miss\ rate * L2\ instruction\ miss\ rate) * memory\ latency$. With an AMAT_i equal to 1, the fetch engine would never block waiting for instructions.

Figure 2(a) shows the AMAT_i for several I-cache sizes when we use both 7 and 15 cycles L2 latencies. We observe that with an 8KB I-cache the fetch engine must wait more than 1.5 cycles per instruction or even 2.08 cycles for a 15 cycles L2 latency. We can also see that for 15 cycles, even a 64KB I-cache shows a latency that still is far from the desired 1. Our results show that the working set starts to fit in caches with sizes of 128KB or larger. This is not strange because the instruction footprint of our queries is larger than 230KB.

Figure 2(b) shows the IPC for different I-cache sizes using perfect branch prediction for 7 cycles L2 latency. The perfect I-cache shown in that Figure is used as an upper bound. Results are shown for both 4 and 8 issue processors. We can see that the decrease in performance for I-caches smaller than 128KB is important. This is due to the increase in exposed latency and is even more evident if we focus on the 8 issue processor. For a 4 issue processor and caches of 8 and 16KB it is better to double I-cache size than to double the issue width.

Figure 2(c) shows results for a 15 cycles L2 latency. We can see that the slope for both 4 and 8 issue is steeper than the one of Figure 2(a). In this case, the drops in IPC are significant even for large caches. For instance, we have speedups of 18% and 23% in 4 and 8 issue processors when we go from a 64KB to a 128KB I-cache.

4 Effect of Instruction Quality on Performance

In this section we want to quantify if the branch prediction accuracy has the same importance as the I-cache performance. In order to inspect that, we have tested the behavior of several branch predictors. In particular we have used Bimodal and Gshare branch predictors of 10, 12, 14, 16, and 18 bits and Hybrid branch predictors of 12 and 14 bits.

The branch prediction accuracy of those predictors is between 98.34% and 88.55%. In general, the branch prediction accuracy is good, compared to the prediction rates obtained on several SpecInt95 benchmarks [6].

Figure 3(a) shows the IPC we can expect from a 4 issue processor running the branch predictors we have tested. Results show an interesting effect. On one

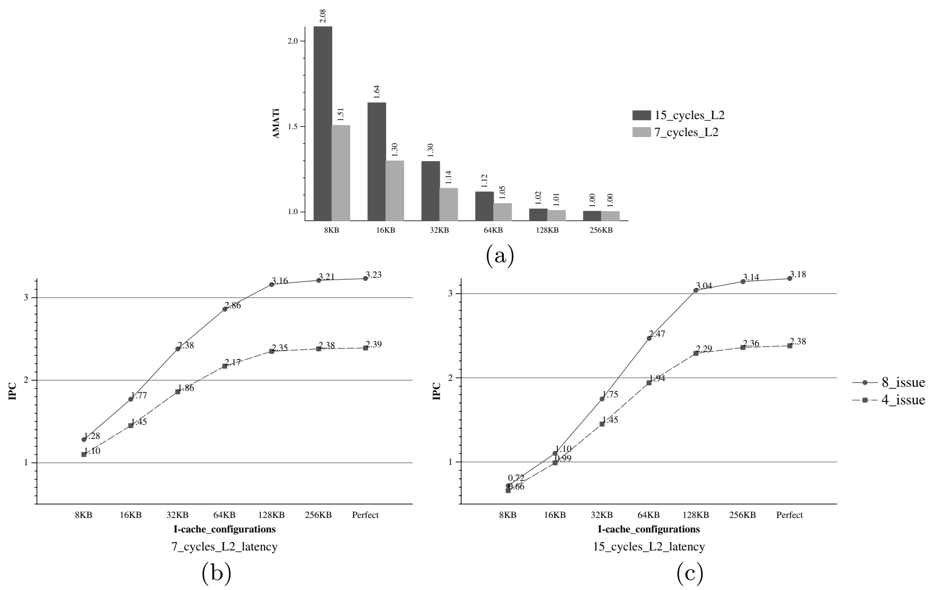

Fig. 2. (a) AMATi in cycles for different L1 I-cache sizes and L2 latencies. Charts (b),(c): IPC for several I-cache configurations. Chart (b) shows results for 7 cycle L2 latency. Chart (c) shows results for 15 cycle L2 latency.

hand, the difference in branch prediction accuracy between the worst performing branch predictor (G10) and the oracle one are clearly reflected in the IPC for the results with a perfect I-cache. That is, the quality of instructions directly enhances the overall performance. On the other hand, for the system with an 8KB I-cache, the benefits of the same increase in instruction quality are very small. We can see that the impact on IPC of the increasing branch prediction accuracy grows when we increase the I-cache size. So the perceived latency is hiding the benefits of the increase in quality. Thus, for our workloads, spending a lot of hardware in complex branch predictors is not cost effective unless $AMAT_i$ is small enough.

The results we have seen so far are for simulations where the misprediction recovery penalty was set to 1. This leads to small branch misprediction penalties. One question that could arise is, how the effect of latency over instruction quality is modified by an increasing branch misprediction penalty.

Figure 3(b) shows results for branch misprediction recovery cycles equal to 6 leading to penalties of at least 8 cycles. In that Figure, we can see that, the effect observed in the last section still exists when we have larger branch misprediction penalties. For branch misprediction recovery of 6 cycles it is still better to duplicate an 8KB I-cache than to have a perfect branch predictor.

If we have a design with a misprediction recovery time of 12 cycles (chart found at [9]), it is always better to go to the perfect branch prediction than to duplicate the cache size. However, it is interesting to note that I-cache size is still

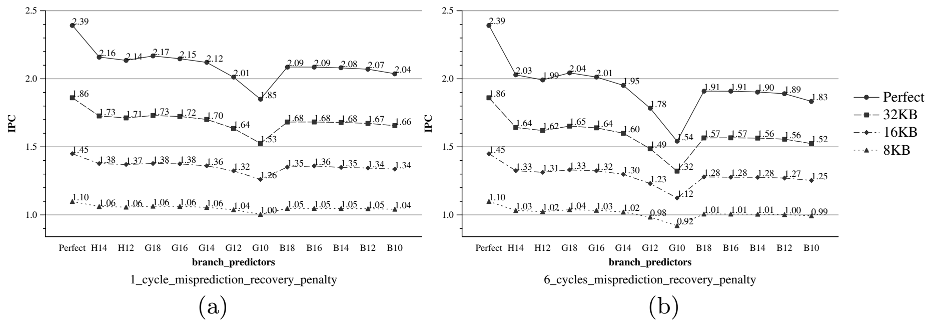

Fig.3. IPC for each of the branch predictors tested. Charts (a) is for misprediction recovery penalties of 1 cycles, while chart (b) is for misprediction recovery penalties of 6 cycles. All simulations are for a 4 issue processor and L2 latency of 7 cycles.

having a bounding effect on the performance benefits of better branch prediction accuracy.

5 Effect of Fetch Bandwidth on Performance

Instruction fetch bandwidth will be a major limiting factor to high performance in next generation microprocessors. Nevertheless, fetch bandwidth could be also a problem for 4 or 8 issue processors.

Several experiments have been performed to characterize the impact that fetch bandwidth has on the performance. In particular we have tested the impact of increasing the fetch bandwidth for a 4 issue processor to 8 instructions per cycle. The complete analysis can be found at [9] and has not been shown here due to space limitations.

Our results show that for 4 and 8 issue processors, large increases in bandwidth do not reflect in such improvements in IPC. Thus, for our workload and issue widths, increasing the fetch bandwidth is not critical. Instead, its is more important to fetch the right path instructions.

6 Code Reordering

In the previous sections we have shown the high impact that the fetch engine design can have in the overall execution performance of our DSS workload. From this evidence, it seems that for our workloads, it can be interesting to use a supplementary technique in order to help I-caches. This technique can be a code reordering scheme.

Code reordering techniques have been used for a long time in order to reduce the I-cache misses [11]. In our previous work, [12,13,14] we have shown that DSS workloads are a good target for these techniques. In particular we presented a reordering technique, the Software Trace Cache (STC), aimed at increasing the fetch bandwidth for future aggressive wide superscalars. This technique has been proven as efficient increasing the number of non taken branches and reducing the I-cache misses.

In this section we want to characterize how the application of the STC reordering affects each of the fetch parameters and how the overall performance can benefit from it. In order to do so, we have used the same PC translation mechanisms used in [13] to simulate the reordering, and we have performed simulations for three different I-cache sizes: 8, 16 and 32 KB.

First, we will study the impact of the STC reordering on the latency. Figure 4(a) shows the AMAT$_i$ for both reordered and non reordered workloads. We can see that for the same I-cache size, the reordered workload always shows a lower latency. We can see that for a 32KB I-cache, the reordered workload has an AMAT$_i$ of the same order as a 64KB one with a non reordered workload.

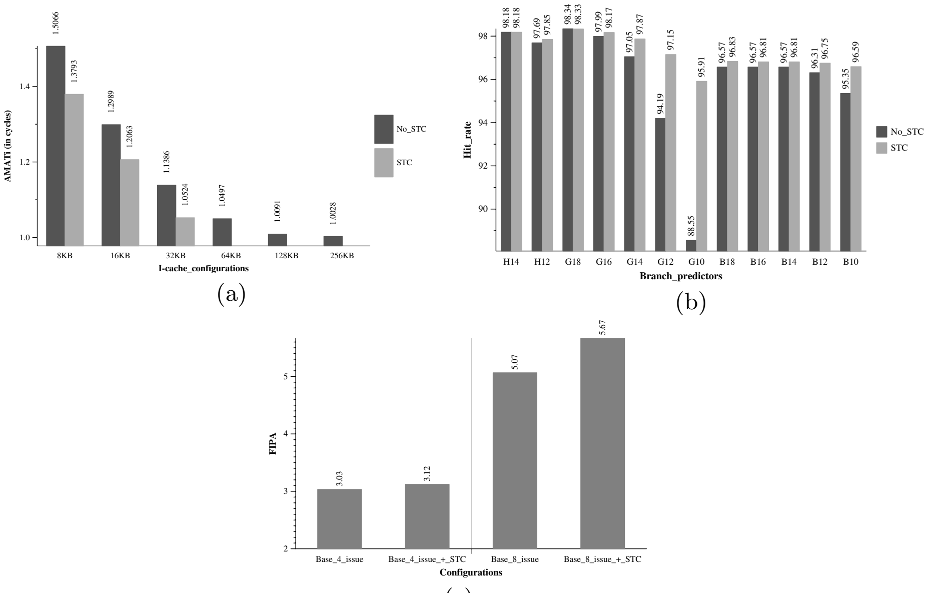

Fig.4. (a) AMAT$_i$ for both reordered and non reordered workload. Results for STC are shown up to 32KB I-cache. (b) Branch prediction hit rates for both reordered and non reordered workload. The branch predictors are the same used in Section 4. (c)Fetched Instructions per Access (FIPA) for both reordered and non reordered workload. The 2 leftmost bars are for 4 issue, while the right ones are for 8 issue.

The second aspect we want to analyze is the impact that the STC has in the quality of the fetched instructions. Figure 4(b), shows the branch prediction hit rate for both reordered and non reordered workloads, for the same branch predictors used in Section 4. Results show that the branch prediction accuracy improves always. This is not strange because the STC reorders the basic blocks in order to leave the biased branches in a not taken form. This leads to a reduction in the destructive conflicts in the branch prediction tables.

The third parameter we want to analyze is the effect that the reordering has in the effective fetch bandwidth. This effect has been shown in [12,13] for aggressive superscalar processors. Here we want to study it for our 4 and 8 issue superscalars. Figure 4(c) shows the number of fetched instructions per access to the fetch engine (FIPA) for the baseline engine on the 4 and 8 issue superscalar processors. In this simulations the branch prediction scheme used is the H12 and the I-cache evaluated is 32KB.

The results show that, while the increase in bandwidth for 4 issue superscalar is small, the increase for 8 issue is bigger (about 12%). This fact, coupled with the branch prediction accuracy results, means that we are not only fetching more instructions but more accurate instructions.

Finally we want to see the overall effect that our STC technique has over the execution performance. Figure 5 shows the performance increases that a conventional fetch engine would expect if the code was reordered with the STC. Results show that for the three configurations we have tested with STC, we always have similar performance to a system with double I-cache size.

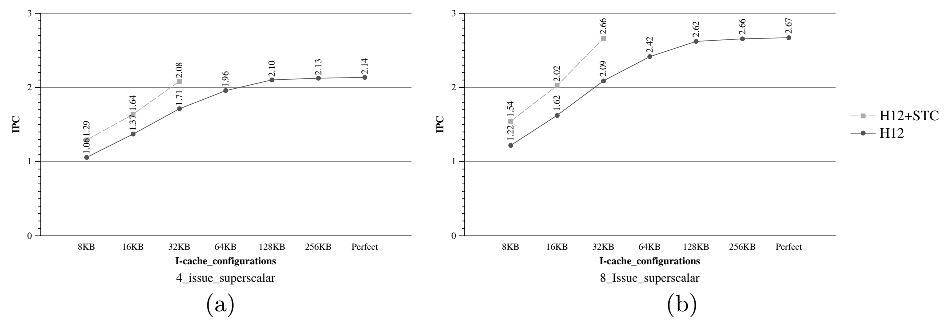

Fig.5. Comparison between the IPC of a non-reordered and a STC reordered workload. Chart (a) show results for 4 issue while chart(b) shows results for 8 issue. Bottom line shows the IPC for the baseline issue processor with the H12 branch predictor and several cache sizes. Upper line shows the IPC of the same configuration if the code is reordered using the STC.

7 Concluding Remarks

In this paper we have analyzed the high performance impact that the fetch engine behavior has on the overall performance of a current superscalar processor while running DSS workloads. We have seen that a bad tuning of the engine can lead to high penalties in the IPC.

The impact of the three main parameters of the fetch engine has been studied. In particular we have seen that, due to the size of the working set of our workload, the impact of I-cache misses is the most important. The fetch engine can not tolerate the latencies of L2 caches. This effect can even hide the benefits of a better branch predictor. So, for these workloads it is more important to spend chip area in hiding L2 cache latency (bigger I-cache for example) than adding a more complex branch predictor.

Finally, we have shown how a code reordering technique (STC) affects the behavior of the different fetch parameters. It is specially interesting the fact that the branch prediction accuracy is improved without extra hardware cost. Also, we have seen that the STC can speed-up the DSS execution in more than 21% for a 4 issue processor, and 27% for an 8 issue one.

References

1. A. Ailamaki, D. Dewitt, M. Hill, and D. Wood. Dbms on modern processors: Where does time go? *Proc. of the 25th Int. Conf. on Very Large Data Bases*, 1999.
2. L. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. *Proc. of the 25th Intl. Symp. on Comp. Architecture*, 1998
3. L. Barroso, K. Gharachorloo, A. Nowatzyk, and B. Verghese. Impact of chip-level integration on performance of oltp workloads. *Proc. of the 6th Intl. Conf. on High Performance Comp. Architecture*, January 2000.
4. D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. *Technical Report TR-1308, Comp. Sciences Dept., Univ. of Wisconsin-Madison*, 1996.
5. T. Conte, K. Menezes, P. Mills, and B. Patell. Optimization of instruction fetch mechanism for high issue rates. *Proc. of the 22th Intl. Symp. on Comp. Architecture*, pages 333–344, June 1995.
6. T . Heil, Z. Smith, and J. E. Smith. Improving branch predictors by correlating on data values. *Proc. of the 32th Intl. Symp. on Microarchitecture*, 1999.
7. W. Hwu and P. Chang. Achieving high instruction cache performance with an optimizing compiler. *Proc. of the 16th Intl. Symp. on Comp. Architecture*, 1989.
8. N. J. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *Proc. of the 17th Intl. Symp. on Comp. Architecture*, pages 364–373, June 1990.
9. C. Navarro, A. Ramírez, J. Larriba-Pey, and M. Valero. Fetch Engine Design Decissions for DSS Workloads *Tech. report UPC-DAC-2000-9* , Feb. 2000.
10. A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized arround trace segments independent of virtual address line. *U.S. Pat. 5.381.533*, Jan. 1995.
11. K. Pettis and R. Hansen. Profile guided code positioning. *Proc. ACM SIGPLAN'99 Conf. on Programming Language Design and Implementation*, pages 16–27, 1990.
12. A. Ramírez, J. Larriba-Pey, C. Navarro, X. Serrano, J. Torrellas, and M. Valero. Code reordering of decision support systems for optimized instruction fetch. *Proc. of the Intl. Conf. on Parallel Processing*, pages 238–245, September 1999.
13. A. Ramírez, J. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero. Software trace cache. *Proc. of the 13th Intl. Conf. on Supercomputing*, pages 119–126, 1999.
14. A. Ramirez, J. Larriba-Pey, and M. Valero. Trace cache redundancy: Red & blue traces. *Proc. of the 6th Intl. Conf. on High Performance Comp. Architecture*, 2000.
15. E. Rottenberg, S. Benett, and J. E. Smith. Trace cache: a low latency aprroach to high bandwith instruction fetching. *Proc. of the 29th Intl. Symp. on Microarchitecture*, pages 24–34, December 1996.

16. J. E. Smith. A study of branch prediction strategies. *Proc. of the 8th Intl. Symp. on Comp. Architecture*, 1981.
17. J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. *Supercomputing 92*, pages 588–597, November 1992.
18. M. Stonebreaker and G. Kemnitz. The postgres next generation database management system. *Communications of the ACM*, October 1991.
19. Transaction Processing Performance Council (TPC). Tpc benchmark d (decision support) http://www.tpc.org. Standard Specification, Revision 1. 2. 3, 1993–1997.
20. T. Y. Yeh, D. T. Marr, and Y. N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. *Proc. of the 7th Intl. Conf. on Supercomputing*, pages 67–76, July 1993.
21. T. Y. Yeh and Y. N. Patt. Two-level adaptive branch prediction. *Proc. of the 24th Intl. Symp. on Microarchitecture*, pages 51–61, 1991.