

Design of a Predictive Filter Cache for Energy Savings in High Performance Processor Architectures ¹

Weiyu Tang

Rajesh Gupta

Alexandru Nicolau

ICS Technical Report

Technical Report #-01-61

April 2001

Center for Embedded Computer Systems
Department of Information and Computer Science
University of California, Irvine
{wtang, rgupta, nicolau}@ics.uci.edu

Department of Information and Computer Science
University of California, Irvine

¹This work was supported in part by DARPA ITO under DIS and PACC program. A version of this report is published in Int'l Conf. on Computer Design, 2001.

Abstract

Filter cache has been proposed as an energy saving architectural feature. A filter cache is placed between the CPU and the instruction cache (I-cache) to provide the instruction stream. Energy savings result from accesses to a small cache. There is however loss of performance when instructions are not found in the filter cache.

The majority of the energy savings from the filter cache in high performance processors are due to the temporal reuse of instructions in small loops. In this paper, we examine subsequent fetch addresses at run-time to predict whether the next fetch address is in the filter cache. In case a miss is predicted, we reduce miss penalty by accessing the I-cache directly. Experimental results show that our next fetch prediction reduces performance penalty by more than 91% and maintains 82% energy-efficiency of a conventional filter cache. Average I-cache energy savings of 31% are achieved by our filter cache design with around 1% performance degradation.

Contents

1	Introduction	1
2	Related work	1
3	Next fetch prediction	2
4	Experimental results	5
5	Conclusion	11

List of Figures

1	Filter cache with/without CPU direct access to the I-cache.	1
2	Next fetch address prediction.	3
3	Instruction fetch from the filter cache.	4
4	Instruction fetch from the I-cache.	5

List of Tables

1	Memory hierarchy configuration.	6
2	Processor configuration.	6
3	Filter cache hit rate.	7
4	Filter cache miss rate.	8
5	Normalized delay.	9
6	Normalized energy.	9
7	I-cache prediction rate.	10
8	Effectiveness of CEP and NFP	11

1. Introduction

High utilization of the instruction memory hierarchy is needed to exploit instruction level parallelism. As a consequence, energy dissipation by the on-chip I-cache is high. For energy efficiency, a filter cache [9], which is shown in the left of Figure 1, is placed between the CPU and the I-cache to service I-cache accesses. Because of its small size, hits in the filter cache can result in energy savings. However, misses in the filter cache increase instruction fetch time. It has been observed that performance degradation can be more than 20%.

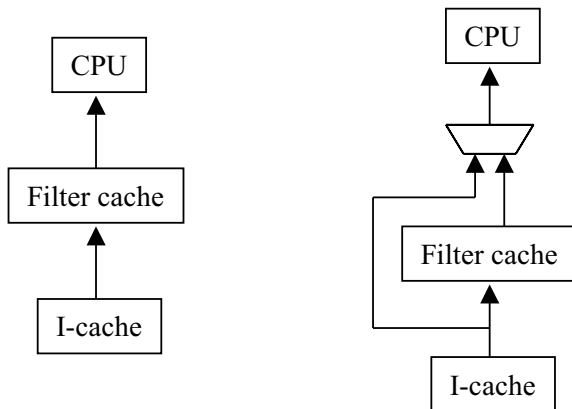


Figure 1. Filter cache with/without CPU direct access to the I-cache.

In this paper, we use the filter cache design where the CPU can access the I-cache directly as shown in the right of Figure 1. For each instruction fetch, we predict whether the next fetch will hit in the filter cache. If a filter cache miss is predicted, the CPU will access the I-cache directly. If this prediction is correct, the filter cache miss penalty is eliminated. If the prediction is wrong, energy consumption is increased.

In high-performance processors, multiple instructions (typically 4 to 8) are fetched simultaneously to support instruction level parallelism. A filter cache line (typical size 16B or 32B) can provide instructions for one or two fetches. Thus there is small or no spatial reuse by the filter cache. Most energy savings by the filter cache are due to the temporal reuse of instructions in small loops. For two consecutive fetches within a small loop, the difference between the fetch addresses is small. Thus we can predict whether the next fetch will hit in the filter cache based on the tags for the current fetch address and the predicted next fetch address.

The rest of this paper is organized as follows. In Section 2, we briefly describe related work on cache energy savings and fetch prediction. We present in Section 3 “next fetch prediction” for filter cache. The experimental results are given in Section 4. The paper is concluded with future work in Section 5.

2. Related work

Loop-Cache [4], which is managed by the compiler, is proposed to amend the large performance degradation of the filter cache. The compiler generates code to maximize the hit rate of the

Loop-Cache.

Dynamic approaches using branch prediction have been proposed to determine when to access the filter cache [3]. These approaches exploit the fact that locality is high for frequently accessed basic blocks. The filter cache is accessed only when frequently accessed basic blocks are detected. One approach based on confidence estimation [6] shows good reduction in performance penalty, but the energy savings are much lower than that of a conventional filter cache.

[11] proposes a multi-level memory system architecture for DSP processors where caches and RAMs coexist to allow high frequencies while maintaining the DSP goals of low cost and low power.

In the set-associative instruction caches, cache way partitioning is exploited for energy savings. In a n -way set-associative cache, power dissipation by one cache way is approximately $\frac{1}{n}$ the power dissipation by the whole cache. In “way-prediction” [7], a table is used to record the most recently used way-footprint for each cache set. On a cache access, the table is first accessed to retrieve the way-footprint for the corresponding cache set. Then that particular way is speculatively accessed. On a way-prediction miss, the remaining ways are accessed in the next cycle. “Selective cache way” [2] proposes to turn off some cache ways for energy savings based on the application requirements.

Fetch prediction has been used in [10, 1, 8]. It exploits the fact that many branches tend to favor one outcome and the same control path may be taken repeatedly.

In trace cache [10], segments of the dynamic instruction stream are stored sequentially. If block A is followed by block B, which in turn is followed by block C, at a particular point in the execution of a program, there is a strong likelihood that they will be executed in that order again. After the first time they are executed in this order, they are stored in the trace cache as a single entry. Subsequent fetches of block A from the trace cache provide block B and C as well.

In Alpha 21264 [8] and UltraSparc II [1], each line in the I-cache is in the following format:

(tag, data, line-prediction, way-prediction).

“Line-prediction” and “way-prediction” are used to speculatively locate the next cache line in a set-associative I-cache. They are dynamically trained based on the program control flow. With the “way-prediction”, one way is accessed most of the time. Thus cache access time is shorter than the access time in a conventional set-associative cache where extra time is needed for way selection.

3. Next fetch prediction

To capture temporal reuse within small loops, we need to predict what the next fetch address is and whether the next fetch address and the current fetch address belong to a small loop.

To predict the next fetch address, we also exploit the facts that the same control path will be taken repeatedly. Each line in the filter cache has the following fields as shown in Figure 2:

(tag, data, next-address).

Suppose line L is accessed for address $addr_A$, followed by the access to line M for address $addr_B$. Then $addr_B$ is filled into the “next-address” field of line L . When line L is accessed the next time for address $addr_C$, the predicted next fetch address is $addr_B$.

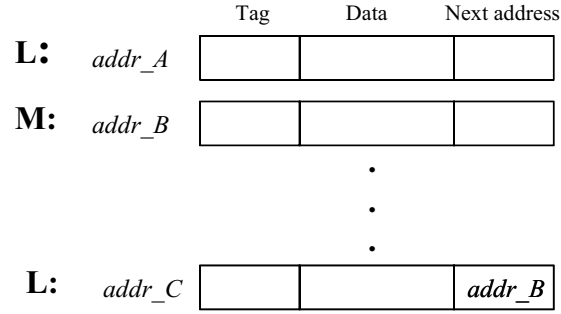


Figure 2. Next fetch address prediction.

If *addr_A* is equal to *addr_C*, it is likely that current control path is the same as the control path that was taken when line *L* was accessed last time. Thus *addr_B* is to be accessed next and the next fetch is likely to hit in the filter cache.

If *addr_A* is not equal to *addr_C*, a different control path is taken. Thus *addr_B* is unlikely to be accessed next and the next fetch is likely to miss in the filter cache. As *addr_A* and *addr_C* are mapped to the same cache line, the following condition is satisfied:

$$tag(addr_A) \neq tag(addr_C) \tag{1}$$

On the other hand, *addr_A* and *addr_B* are consecutively fetched. If they belong to a small loop that fits in the filter cache, it is likely that the following condition is satisfied:

$$tag(addr_A) == tag(addr_B) \tag{2}$$

Based on Equation 1 and 2, the following equation is used for next fetch prediction:

$$tag(addr_B) == tag(addr_C) \tag{3}$$

The next fetch is predicted to hit in the filter cache if:

the tag for the current fetch address (addr_B) is equal to the tag for the predicted next fetch address (addr_C).

It is not necessary to use all the tag bits for the comparison in Equation 3. The lowest four tag bits are used in our implementation. In our experiments, we have found that using lowest four tag bits achieves 97% the prediction accuracy of using all the tag bits.

Instead of coupling the “next-address” field with the filter cache line, a separate “next-address” prediction (NP) table is used as shown in Figure 3 and 4. The advantage of a separate table is that “next-address” prediction can proceed when the next fetch is directed to the I-cache as shown in Figure 4.

The following hardware is needed for the prediction:

- NP table, 4-bit per entry, table size equal to the number of lines in the filter cache;

- A register named *Last_line*, which holds the line number for the filter cache line accessed last time;
- A comparator to determine whether the tag of current fetch address matches the value in the corresponding entry of the NP table.

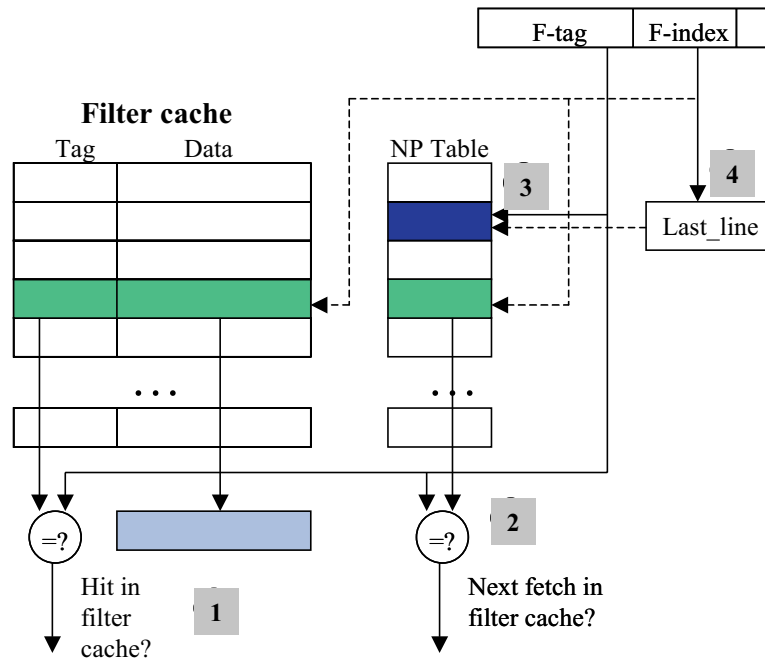


Figure 3. Instruction fetch from the filter cache.

Figure 3 shows relevant operations when the instruction fetch is directed to the filter cache as listed below:

1. Filter cache access, which is the same as cache access in a conventional cache;
2. Next fetch prediction, which is used to determine whether the next fetch will hit in the filter cache;
3. NP table update, where most recently used tags are saved for future prediction;
4. Register *Last_line* update, which is necessary for the next fetch prediction.

Figure 4 shows relevant operations when the instruction fetch is directed to the I-cache as listed below:

1. I-cache access;
2. Next fetch prediction;
3. NP table update;

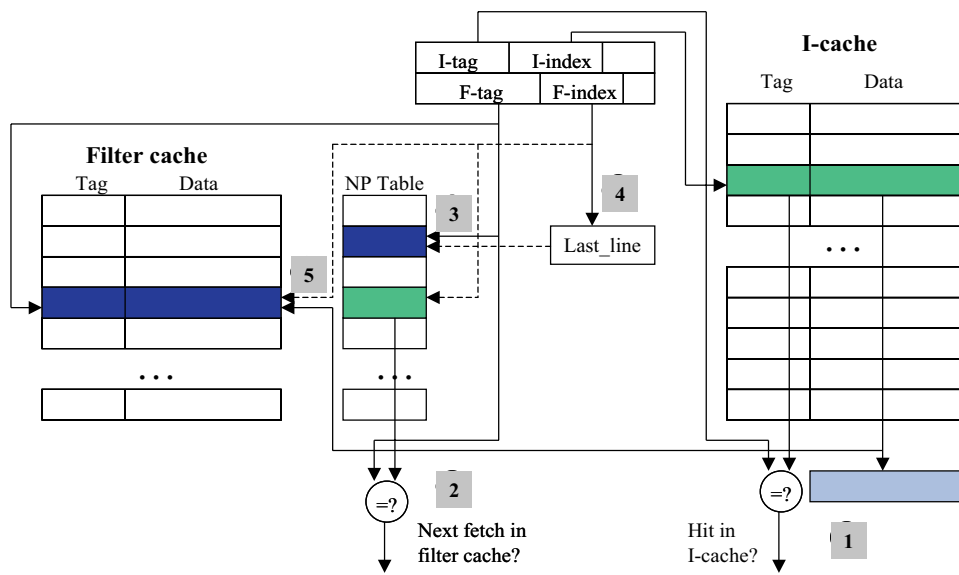


Figure 4. Instruction fetch from the I-cache.

4. Register *Last_line* update;
5. Filter cache update.

Most of the operations are similar to the corresponding operations for instruction fetch from the filter cache.

When the fetch is directed to the I-cache, the prediction is that instructions being fetched are not in the filter cache. These instructions are sent to the filter cache for future hits in the filter cache.

4. Experimental results

We use the SimpleScalar toolset [5] to model an out-of-order superscalar processor. The processor and memory hierarchy parameters shown in Table 1 and 2 roughly correspond to those in current high-end microprocessors.

There are 2 banks in each way of the I-cache and each cache line spans two banks. Any bank can provide 4 instructions (4B instruction size) in a fetch. Cache banking improves cache access time and cuts the per I-cache access power dissipation by half. As the fetch width is 4, there will be no additional energy savings if the number of banks in a cache way is larger than 2. Consequently, we select 16B as the line size for the filter cache. For line size larger than 16B, multiple I-cache accesses are needed to fill a filter cache line, which will dramatically increase the number of I-cache accesses. This will offset the energy savings from the use of filter cache and will result in much higher I-cache power dissipation. Because one filter cache line can only provide instructions for one fetch, there is no spatial reuse in the filter cache and only temporal reuse can be exploited.

The power parameters are obtained using Cacti [12] for the 0.18 μ m technology. SPEC95 benchmarks are simulated. For each benchmark, 100 million instructions are simulated.

Parameter	Value
Filter cache	256B or 512B, direct-mapped, 16B line
L1 I-cache	64KB, 4-way, 32B line, two banks per way, 1-cycle latency
L1 D-cache	64KB, 4-way, 32B line, 1-cycle latency, 2 ports
L2 cache	512KB, 4-way, 64B line, 8-cycle latency
Memory	30-cycle latency

Table 1. Memory hierarchy configuration.

Parameter	Value
branch pred.	combined, 4K 2-bit chooser, 4k-entry bimodal, 12-bit, 4K-entry global
	7-cycle miss prediction penalty
BTB	4K-entry, 4-way
RUU	64
LSQ	16
fetch queue	16
fetch speed	2
fetch width	4
int. ALUs	2
flt. ALUs	2
int. Mult/Div	2
flt. Mult/Div	2

Table 2. Processor configuration.

We have evaluated the following three schemes:

- **CON**– **CON**ventional filter cache with no direct path from the CPU to the I-cache;
- **CEP**– filter cache where there is a direct path from the CPU to the I-cache and **C**onfidence **E**stimation **P**rediction proposed in [3] is used to determine when to access the filter cache; (when a branch is encountered, if it is predicted strong “taken”/“untaken”, the CPU will access the filter cache for subsequent instructions until another branch is encountered; otherwise, the CPU will access the I-cache for subsequent instructions.)
- **NFP**– filter cache where there is a direct path from the CPU to the I-cache and **N**ext **F**etch **P**rediction is used to determine when to access the filter cache.

Benchmarks	CON	CEP	NFP
compress	0.891	0.212	0.825
gcc	0.392	0.138	0.284
go	0.446	0.155	0.332
ijpeg	0.799	0.492	0.603
li	0.417	0.200	0.267
perl	0.188	0.116	0.133
applu	0.572	0.305	0.525
apsi	0.538	0.435	0.447
fpppp	0.026	0.014	0.020
hydro2d	0.347	0.258	0.266
su2cor	0.452	0.249	0.381
swim	0.217	0.165	0.079
tomcatv	0.314	0.223	0.232
turb3d	0.770	0.674	0.629
wave	0.202	0.163	0.126
avg	0.438	0.253	0.343

Table 3. Filter cache hit rate.
(filter cache size = 256B)

Table 3 shows filter cache hit rate, calculated as the ratio of the number of hits in the filter cache versus total number of instruction fetches. CON has the highest hit rate because it doesn’t use any prediction. The hit rate of NFP is higher than that of CEP for all benchmarks except *swim*, *turb3d* and *wave*. The advantage of NFP over CEP is more evident in integer benchmarks, where there are many branches with low confidence estimation. The filter cache is not accessed in CEP if the confidence estimation for branches is low.

Table 4 shows filter cache miss rate, calculated as the ratio of the number of misses in the filter cache versus total number of instruction fetches. The sum of hit rate and miss rate is not equal to 1 in CEP and NFP because some fetches go to the I-cache directly.

The miss rate of CEP and NFP is much higher than that of CON. And the miss rate of NFP is much lower than that of CEP for all benchmarks. CEP predicts based on confidence estimation of branches and has no knowledge of code size and filter cache size. Thus the miss rate may be

Benchmarks	CON	CEP	NFP
compress	0.109	0.054	0.046
gcc	0.608	0.204	0.085
go	0.554	0.180	0.083
ijpeg	0.201	0.120	0.073
li	0.583	0.288	0.118
perl	0.812	0.497	0.092
applu	0.428	0.200	0.038
apsi	0.462	0.288	0.057
fpppp	0.974	0.850	0.072
hydro2d	0.653	0.385	0.080
su2cor	0.548	0.281	0.085
swim	0.783	0.518	0.079
tomcatv	0.686	0.387	0.098
turb3d	0.230	0.203	0.021
wave	0.798	0.457	0.134
avg	0.562	0.327	0.077

Table 4. Filter cache miss rate.
(filter cache size = 256B)

high even though the confidence estimation prediction is accurate. For example, *fpppp* has loops that are accessed frequently. Confidence estimation can identify the loops and the filter cache is accessed for instructions in the loop body. However, the loop size is often larger than the filter cache size and most instructions will be replaced before temporal reuse. As a consequence, the miss rate for *fpppp* is very high, 0.974 for CON and 0.85 for CEP. On the other hand, NFP has knowledge of the filter cache size and the predicted next fetch address. Thus NFP can make more accurate prediction on whether the next fetch will hit in the filter cache.

Table 5 shows normalized delay for filter caches of size 256B and 512B. The baseline system configuration for comparison has no filter cache. For every benchmark, the delay by CON is the highest and the delay by NFP is the lowest. We observe that high miss rate (seen in Table 4) results in high performance degradation.

For some benchmarks such as *apsi*, the normalized delay is lower than 1. Instruction fetches are delayed on miss-fetches in the filter cache. Several instructions are committed during a miss-fetch cycle and branch history may be changed. This somehow improves the branch prediction accuracy for some benchmarks. Comparing to 1 cycle filter cache miss-fetch penalty, the branch miss-prediction penalty is 7 cycles. Thus more accurate branch prediction can result in performance improvement.

Table 6 shows the normalized energy for the filter caches of size 256B and 512B. The energy of NFP is close to the energy of CON. And the energy of NFP is much lower than the energy of CEP. High hit rate (seen in Table 3) and low delay (seen in Table 5) results in low energy. As there is no spatial reuse in the filter cache, the normalized energy shown in Table 6 is higher than reported in [9, 3]. For some benchmarks such as *perl*, *fpppp*, *swim* and *wave*, the normalized energy is close to or even more than 1. When there are small or no energy savings, it is beneficial to turn off the filter cache completely for these benchmarks to avoid performance degradation.

Benchmarks	256B			512B		
	CON	CEP	NFP	CON	CEP	NFP
compress	1.042	1.022	1.014	1.000	1.000	1.000
gcc	1.130	1.042	1.011	1.107	1.037	1.010
go	1.091	1.031	1.010	1.062	1.025	1.010
jpeg	1.009	1.007	1.017	1.003	1.002	1.002
li	1.130	1.080	1.027	1.077	1.068	1.022
perl	1.225	1.114	1.019	1.193	1.106	1.016
applu	1.101	1.040	1.001	1.005	1.002	1.000
apsi	1.034	1.015	0.994	1.018	1.009	0.996
fpppp	1.179	1.150	1.001	1.174	1.149	1.003
hydro2d	1.178	1.066	1.013	1.170	1.081	1.017
su2cor	1.131	1.034	0.994	1.101	1.024	0.996
swim	1.552	1.373	1.054	1.449	1.262	1.078
tomcatv	1.152	1.057	1.007	1.138	1.049	1.005
turb3d	1.009	1.005	1.000	1.006	1.004	1.001
wave	1.285	1.163	1.014	1.197	1.125	1.003
avg	1.150	1.080	1.012	1.113	1.063	1.011

Table 5. Normalized delay.

Benchmarks	256B			512B		
	CON	CEP	NFP	CON	CEP	NFP
compress	0.172	0.807	0.235	0.090	0.757	0.145
gcc	0.699	0.893	0.782	0.617	0.877	0.700
go	0.642	0.874	0.733	0.514	0.833	0.591
jpeg	0.269	0.549	0.460	0.248	0.535	0.426
li	0.672	0.844	0.800	0.631	0.819	0.700
perl	0.915	0.947	0.933	0.849	0.881	0.877
applu	0.509	0.734	0.537	0.173	0.512	0.202
apsi	0.545	0.622	0.616	0.363	0.484	0.432
fpppp	1.086	1.083	1.046	1.137	1.133	1.071
hydro2d	0.746	0.800	0.800	0.745	0.815	0.778
su2cor	0.636	0.797	0.684	0.636	0.804	0.672
swim	0.884	0.903	0.987	0.783	0.791	0.941
tomcatv	0.782	0.834	0.835	0.763	0.848	0.822
turb3d	0.300	0.388	0.431	0.197	0.295	0.218
wave	0.899	0.898	0.943	0.788	0.837	0.878
avg	0.658	0.796	0.731	0.577	0.751	0.643

Table 6. Normalized energy.

For a 256B filter cache, average 26.9% energy savings are achieved using NFP with 1.2% performance degradation. For a 512B filter cache, average 35.7% energy savings are achieved using NFP with 1.1% performance degradation. Given such small performance degradation, NFP is suitable for high-performance processors.

Benchmarks	CEP	NFP
compress	0.735	0.130
gcc	0.657	0.631
go	0.665	0.585
jpeg	0.388	0.323
li	0.512	0.615
perl	0.386	0.775
applu	0.494	0.437
apsi	0.277	0.496
fpppp	0.136	0.908
hydro2d	0.357	0.654
su2cor	0.470	0.534
swim	0.317	0.842
tomcatv	0.390	0.670
turb3d	0.124	0.351
wave	0.380	0.740
avg	0.419	0.579

Table 7. I-cache prediction rate.
(filter cache size = 256B)

Table 7 shows the I-cache prediction rate, calculated as the ratio of the number of instruction fetches, which access the I-cache directly based on prediction, versus total number of instruction fetches. For benchmarks with small or even negative energy savings, the I-cache prediction rate by NFP is high. For example, the energy savings rate by *wave* is 0.057 and the I-cache prediction rate by it is 0.74. On the other hand, for benchmarks with large energy savings, the I-cache prediction rate by NFP is low. For example, the energy savings rate by *compress* is 0.765 and the I-cache prediction rate by it is 0.13. The I-cache prediction rate by NFP is a good indicator of energy savings by the filter cache. If a high rate is detected, which means the potential energy savings are small, the filter cache can be turned off to avoid performance degradation.

Table 8 shows the effectiveness of CEP and NFP compared to CON. The delay reduction by NFP is nearly twice the reduction by CEP. NFP achieves 82% of the energy savings by CON and CEP achieves only 59% of the energy savings by CON. In terms of energy-delay product, NFP is lower than CON, while CEP is 17.6% higher than CON. We conclude that NFP is better than CEP in delay reduction and energy saving for filter caches.

Note that the energy-delay product of the NFP is slightly better than that of the CON because the lower delay by the NFP has compensated its higher energy. In addition, the NFP is beneficial

		256B	512B	avg
delay reduction	CEP	0.467	0.442	0.455
	NFP	0.920	0.902	0.911
energy savings	CEP	0.596	0.589	0.593
	NFP	0.791	0.844	0.818
energy* delay	CEP	1.130	1.221	1.176
	NFP	0.949	0.972	0.961

Table 8. Effectiveness of CEP and NFP
(Normalized to CON).

for energy-efficiency of the whole system. With CON, the energy-delay product of other processor components such as register files will increase dramatically because of the high delay of CON. Hence energy-efficiency of CON on I-cache may not translate into energy-efficiency of the whole system.

5. Conclusion

In this work, we presented a prediction technique to determine whether the next fetch will hit in the filter cache to reduce the performance penalty. The idea is that in high-performance processors, most energy savings of the filter cache are due to the temporal reuse of instructions in small loops. The tags for the current fetch address and the predicted next fetch address can determine whether they belong to the same small loop. As the prediction uses the filter cache size, it is more accurate than currently used dynamic prediction techniques. Moreover, next fetch prediction needs minimal hardware. The performance degradation is around 1% with this technique and average 31% I-cache energy savings are achieved.

Branch prediction can improve the accuracy of next fetch prediction. We are investigating techniques to combine branch prediction with next fetch prediction to further reduce the performance penalty.

References

- [1] K.B. Normoyle et al. UltraSparc-III: expanding the boundaries of system on a chip. *IEEE Trans. Micro*, 18(2):14–24, 1998.
- [2] D. H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *Int'l Symp. Microarchitecture*, pages 248–259, 1999.
- [3] N. Bellas, I. Hajj, and C. Polychronopoulos. Using dynamic cache management techniques to reduce energy in a high-performance processor. In *Int'l Symp. on Low Power Electronics and Design*, pages 64–69, 1999.
- [4] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis. Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors. In *Int'l Symp. on Low Power Electronics and Design*, pages 70–75, 1998.
- [5] D. Burger and T. Austin. The simplescalar toolset, version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, 1997.

- [6] D. Grunwald, A. Klauser, S. Manner, and A. Plezskun. Confidence estimation for speculation control. In *Int'l Symp. Computer Architecture*, pages 122–131, 1998.
- [7] K. Inoue, T. Ishihara, and K. Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *Int'l Symp. on Low Power Electronics and Design*, pages 273–275, 1999.
- [8] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.
- [9] J. Kin, M. Gupta, and W. Mangione-Smith. The filter cache: An energy efficient memory structure. In *Int'l Symp. Microarchitecture*, pages 184–193, 1997.
- [10] E. Rotenberg, S. Bennett, and J. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Int'l Symp. Microarchitecture*, 1996.
- [11] S. Agarwala et al. A multi-level memory system architecture for high-performance dsp applications. In *IEEE Int'l Conf. on Computer Design*, pages 408–413, 2000.
- [12] S. Wilton and N. Jouppi. An enhanced access and cycle time model for on-chip caches. Technical Report 93/5, Digital Western Research Laboratory, 1994.