Using dynamic cache management techniques to reduce energy in a high-performance processor *

Nikolaos Bellas, Ibrahim Hajj, and Constantine Polychronopoulos

Department of Electrical & Computer Engineering
and the Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 West Main Street, Urbana, IL 61801, USA
{nikos,hajj}@uivlsi.csl.uiuc.edu

Abstract

In this paper, we propose a technique that uses an additional mini cache, the *L0-Cache*, located between the instruction cache (I-Cache) and the CPU core. This mechanism can provide the instruction stream to the data path and, when managed properly, it can effectively eliminate the need for high utilization of the more expensive I-Cache. In this work, we propose, implement, and evaluate a series of run-time techniques for dynamic analysis of the program instruction access behavior, which are then used to proactively guide the access of the L0-Cache. The basic idea is that only the most frequently executed portions of the code should be stored in the L0-Cache since this is where the program spends most of its time.

We present experimental results to evaluate the effectiveness of our scheme in terms of performance and energy dissipation for a series of SPEC95 benchmarks. We also discuss the performance and energy tradeoffs that are involved in these dynamic schemes.

1 Introduction

As processor performance continues to grow, and high performance, wide-issue processors exploit the available *Instruction-Level Parallelism*, the memory hierarchy should continuously supply instructions and data to the data path to keep the execution rate as high as possible. Very often, the memory hierarchy access latencies dominate the execution time of the program. The very high utilization of the instruction memory hierarchy entails high energy demands for the on-chip I-Cache subsystem.

In order to reduce the effective energy dissipation per instruction access, we propose the addition of a small, extra cache (the L0-Cache) which serves as the primary cache of the processor, and is used to store the most frequently executed portions of the code, and subsequently provide them to the pipeline. Our approach seeks to manage the L0-Cache in a manner that is sensitive to the frequency of accesses of the instructions executed. It can exploit the temporalities of the code and can make decisions on-the-fly, i.e., while the code executes.

The problem that the dynamic techniques seek to solve is how to select basic blocks[1] to be stored in the L0-Cache while the program is being executed. If a block is selected, the CPU will access the L0-Cache first; otherwise, it will go directly to the I-Cache. In case of an L0-Cache miss, the CPU is directed to the I-Cache to get the instruction and, at the same time, to transfer the instruction from the I-Cache to the L0-Cache. A penalty of one clock cycle has to be paid in case of an L0-Cache miss. The L0-Cache is loaded with instructions from the I-Cache after a miss.

The paper is organized as follows: in section 2, we review previous work regarding energy and power minimization in the microarchitectural level, as well as dynamic management of the memory hierarchy for performance enhancement. Next, in section 3 we briefly describe the hardware that we use, and, in section 4, we explain the basic idea behind our scheme. Section 5 details our solution to dynamic selection of basic blocks to be cached in the L0-Cache, and gives several techniques that trade off delay and energy reduction. The experimental results for each technique are also given in section 5. The paper is concluded with section 6.

2 Related Work

The area of power minimization at the architectural and software levels is relatively new. The impact of memory hierarchy in minimizing power consumption, and the exploration of data-reuse in order to reduce the power required to read or write data in the memory is addressed in [1].

The filter cache [2] tackles the problem of large energy consumption of the L1 caches by adding a small, and thus more energy-efficient cache between the CPU and the L1 caches. The penalty to be paid is the increased miss rates and, hence, longer average memory access time.

In [3], the addition of a compiler-managed extra cache (the Loop-Cache) is proposed that amends the large performance degradation of the filter cache. In this scheme, the compiler generates code that exploits the new memory hierarchy by maximizing the hit rate of the L-Cache.

The work in [4] focuses on the excessive energy dissipation of high-performance, speculative processors that tend to execute more instructions than are actually needed in a program. The CPU stops execution in the pipeline when there is a large probability of wrong path execution, and it resumes only when the actual execution path is detected.

There has been an extensive research effort lately on techniques to improve the memory hierarchy performance through

[1] A basic block is a sequence of instructions with no transfers in and out except possibly at the beginning or end.

dynamic techniques. Along this line, the authors in [5] present techniques for dynamic analysis of program data access behavior, which are then used to guide the placement of data within the memory hierarchy. Data that are expected to have little reuse in the cache are bypassed and are not placed in the L1 D-Cache.

The techniques proposed in [5] can also be used in our scheme to manage the caching of instructions in the L0-Cache. They can detect the most frequently executed portions of the code dynamically, and, then, direct the L0-Cache to store only those portions. However, these techniques require the addition of extra hardware in the form of extra tables or counters to keep statistics during execution. The extra hardware dissipates energy, and can offset the possible energy gains from the usage of the L0-Cache. To make the dynamic techniques attractive for low energy, we need to use hardware that already exists in the CPU. The hardware we will use in this work is the *branch prediction* mechanism.

3 Branch prediction and confidence estimation–A brief overview

As the processor speed increases and instruction-level parallelism becomes widely used, conditional branches pose an increasingly heavy burden for the growth of uniprocessor performance. Branch prediction is the most popular method to increase parallelism in the CPU, by predicting the outcome of a conditional branch instruction as soon as it is decoded. Provided that the branch prediction rate is high, the pipeline executes from the correct path and avoids unnecessary work most of the time.

The branch prediction problem can actually be divided into two subproblems. The prediction of the direction of the branch and the prediction of the target address if the branch is predicted to be taken. Both subproblems should be solved for the branch prediction to be meaningful. In this work, we are only interested in the prediction of the branch direction.

3.1 Previous work on branch prediction

Successful branch prediction mechanisms take advantage of the non-random nature of branch behavior [6]. Most branches are either taken or not-taken. Moreover, the behavior of a branch usually depends on the behavior of the surrounding branches in the program.

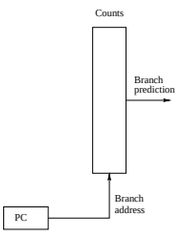

Figure 1: Bimodal branch predictor. Each entry in the table is a 2-bit saturated counter.

Bimodal branch predictor. The *bimodal* branch predictor in Fig. 1 takes advantage of the bimodal behavior of most branches. Each entry in the table shown in Fig. 1 is a 2-bit saturated counter which determines the prediction. Each time a branch is taken, the counter is incremented by one, and each time it falls through it is decremented by one (Fig. 2). The prediction is done by looking into the value of the counter: if it less than 2, the branch is predicted as not taken; otherwise, it is predicted as taken. By using a 2-bit counter, the predictor can tolerate a branch going into an unusual direction once.

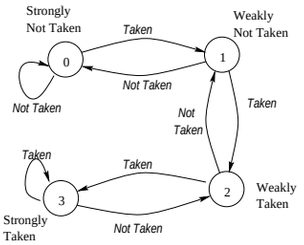

Figure 2: FSM for the 2-bit saturated counters.

The table is accessed through the address of the branch using the program counter (PC). Ideally, each branch has its own entry in the table, but for smaller tables multiple branches may share the same entry. The table is accessed twice for each branch: first to read the prediction, and then to modify it when the actual branch direction has been resolved later in the pipeline.

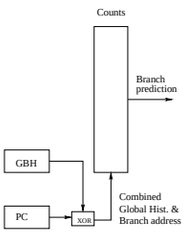

Figure 3: Global branch predictor with index sharing.

Global branch predictor. In the bimodal brach prediction method, only the past behavior of the current branch is considered. Another scheme is proposed in [7] which also considers the behavior of other branches to predict the behavior of the current branch. This is called global prediction, and the hardware implementation is similar to the implementation of the bimodal method (Fig. 3). The difference is that the table with the counters is accessed with the *Global Branch History* (GBH) register, which contains the outcome of the n most recent branches. A single shift register, which records the direction taken by the n most recent branches, can be used. This information is combined with the address of the branch under consideration (via XOR or concatenation) to index the table of counters. This predictor is called global branch predictor with index sharing.

McFarling branch predictor. Finally, McFarling [8] combines two predictors to achieve better results. In Fig. 4, a McFarling predictor is shown which consists of three tables. The tables PR1 and PR2 contain the counters for the two independent predictors, and the selector counter determines which predictor will be used to give the prediction. The two predictors can be any of the predictors we discussed in the previous paragraphs. McFarling found out that the combination of a local [9] and a global predictor with index sharing gives the best results.

Each entry in the selector counter contains a 2-bit saturated counter. This counter determines which predictor will be used for the prediction and is updated after the direction of the branch has been resolved. The counter is biased towards the predictor that usually gives correct prediction for that particular branch.

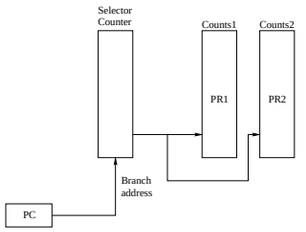

Figure 4: McFarling branch predictor.

3.2 Previous work on confidence estimation

In many cases computer architects want to assess the quality of a branch prediction and determine how confident the machine is that the prediction will be correct. The relatively new concept of *confidence estimation* has been introduced recently to quantify this confidence and keep track of the quality of branch predictors [10].

The confidence estimators are hardware mechanisms that are accessed in parallel with the branch predictors when a branch is decoded, and they are modified when the branch direction is resolved. They characterize a branch prediction as "high confidence" or "low confidence" depending upon the history of the branch predictor for the particular branch. For example, if the branch predictor predicted a branch correctly most of the time, the confidence estimator would designate this prediction as "high confidence," otherwise as "low confidence." We should note that the confidence estimation mechanism is orthogonal to the branch predictor used. In other words, we can use any combination of confidence estimators and branch predictors.

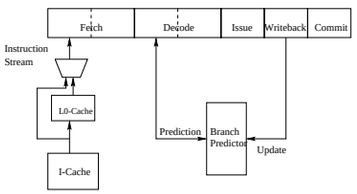

Figure 5: Pipeline microarchitecture

Figure 5 shows the pipeline with the extra cache and the branch predictor. A branch is decoded at the front end of the pipeline, but its direction is only resolved when it is executed.

4 Basic idea of the dynamic management scheme

The dynamic scheme for the L0-Cache should be able to select the most frequently executed basic blocks for placement in the L0-Cache. It should also rely on existing mechanisms without much extra hardware investment if it is to be attractive for energy reduction.

The branch prediction in conjunction with the confidence estimator mechanism is a reliable solution to this problem. During program execution, the branch predictor accumulates the history of branches and uses this history to guess the branch behavior in the future. Since the branch predictor is usually successful in predicting the branch direction, we can assume that predictors describe accurately the behavior of the branch during a specific phase of the program. Confidence estimators provide additional information about the steady-state behavior of the branch.

For example, a branch that was predicted "taken" with "high confidence" will be expected to be taken during program execution in that particular phase of the program. If it is not taken (i.e., in case of a misprediction), it will be assumed to behave unusually. Of course, what is "usual" or "unusual" behavior in the course of a program for a particular branch can change. Some branches can change behavior from mostly taken to mostly untaken during execution. Moreover, many branches, especially in integer benchmarks, can be in a gray area, and not have a stable behavior with respect to direction, or can follow a complex pattern of behavior.

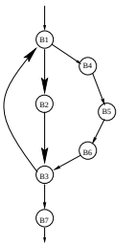

Figure 6: An "unusual" branch direction leads to a rarely executed portion of the code.

If a branch behaves "unusually," it will probably drive the thread of control to a portion of the code that is not very frequently executed. The loop shown in Fig. 6 executes the basic blocks B_1, B_2, and B_3 most of the time, and it seldom executes B_4, B_5, and B_6. The branch at the end of B_1 will be predicted "not-taken" with "high confidence." If it is taken, it will drive the program to the rarely executed branch, i.e., it will behave unusually. A similar situation exists for B_3 and B_7.

These observations lay the foundation for the dynamic selection of basic blocks in the L0-Cache scheme. In our approach, we attempt to capture the most frequently executed basic blocks by looking into the behavior of the branches. The basic idea is that, if a branch behaves "unusually," our scheme disables the L0-Cache access for the subsequent basic blocks. Under this scheme, only basic blocks that are executed frequently tend to make it to the L0-Cache. Hence, we avoid *cache pollution* problems in the L0-Cache, i.e., storing there infrequently accessed portions of the code, that replace more frequently accessed code, and that could create conflict misses in the small L0-Cache.

Instructions are transferred to the L0-Cache only in case of an L0-Cache miss. A whole block is then transferred from the I-Cache (8 or 16 bytes in our experiments). We assume that no prefetching mechanism exists in the memory hierarchy system.

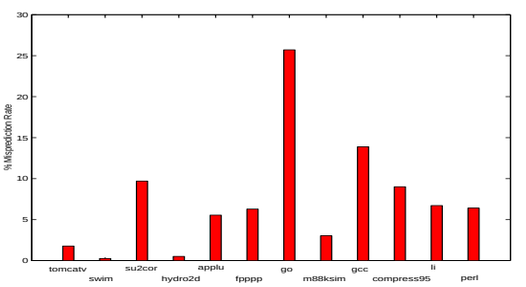

Figure 7: Misprediction rate for the SPEC95 benchmarks.

Not all branches can be characterized as "high confidence." What is more, branch predictors are not always accurate in predicting the direction of a branch. In Fig. 7, the branch

misprediction rates for most of the SPEC95 benchmarks are shown. The first six benchmarks are floating point, and the last six are integer. We use a McFarling branch predictor in which each one of the three tables used has 2048 entries. The bimodal and the global branch predictor with index sharing are used as the component predictors. Branch predictors are quite successful with numerical or computation-intensive code, yet they sometimes fail for integer benchmarks like *099.go* and *126.gcc*.

In section 5 we propose various dynamic methods for the selection of basic blocks that span the range of accuracy and complexity. We make the realistic assumption that the processor is already equipped with a branch prediction mechanism. We are assuming a McFarling predictor for all experiments.

5 Dynamic techniques for selecting basic blocks for the L0-Cache

5.1 Experimental setup

To gauge the effect of our L0-Cache in the context of a realistic processor operation, we simulated the MIPS2 instruction set architecture (ISA) using the *MINT* [11] and the *SpeedShop* [12] tool suites. MINT is a software package for instrumenting and simulating binaries on a MIPS machine. We built a MIPS2 simulator on top of MINT which accurately reflects the execution profile of the R-4400 processor. Table 1 describes the memory subsystem base configuration as (cache size / block size / associativity / cycle time / latency to L2 cache in clock cycles / transfer bandwidth in bytes per clock cycles from the L2 Cache). Both I-Cache and D-Cache are banked both row-wise and column-wise to reduce the access time and the energy per access [13]. We use the tool *cacti*, described in [13], to estimate the access time of the on-chip caches, as well as the optimal banking that minimizes the access time.

We have developed our cache energy model based on the work by Wilson and Jouppi [13] in which they propose a timing analysis model for SRAM-based caches [14]. Our model uses run-time information of the cache utilization (number of accesses, number of hits, misses, input statistics, etc.) gathered during simulation, as well as complexity and internal cache organization parameters (cache size, block size, associativity, banking, etc.). A 0.8 μm technology with 3.3 volts supply voltage is assumed. These models are used for the estimation of energy in both the I-Cache and the L0-Cache. The detailed description of the energy models can be found elsewhere [14].

The utilization parameters are available from the simulation of the memory hierarchy. The cache layout parameters, such as transistor and interconnect physical capacitances, can be obtained from existing layouts, from libraries, or from the final layout of the cache itself. We use the numbers given in [13] for a 0.8 μm process.

Table 1: Memory subsystem configuration in the base machine.

Parameter	Configuration
L1 I-Cache	32KB/32/1/1/4/8
L1 D-Cache	32KB/32/2/1/4/8

5.2 Simple method without using confidence estimators

The branch predictor can be used as a stand-alone mechanism to provide insight on which portions of the code are frequently executed and which are not. A mispredicted branch is assumed to drive the thread of execution to an infrequently executed part of the program.

Table 2: Energy results for the simple method.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	0.185	0.177	0.100	0.121
swim	0.123	0.134	0.099	0.118
su2cor	0.238	0.208	0.161	0.172
hydro2d	0.125	0.137	0.091	0.115
applu	0.329	0.253	0.292	0.232
fpppp	0.574	0.365	0.566	0.361
go	0.609	0.509	0.572	0.488
m88ksim	0.435	0.315	0.382	0.288
gcc	0.556	0.445	0.515	0.420
compress95	0.437	0.349	0.338	0.290
li	0.453	0.363	0.403	0.322
perl	0.513	0.396	0.451	0.355

Table 3: Delay results for the simple method.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	1.050	1.032	1.011	1.006
swim	1.028	1.017	1.015	1.008
su2cor	1.056	1.030	1.021	1.012
hydro2d	1.020	1.013	1.006	1.004
applu	1.108	1.056	1.089	1.045
fpppp	1.235	1.118	1.230	1.116
go	1.159	1.091	1.138	1.079
m88ksim	1.184	1.103	1.155	1.085
gcc	1.180	1.107	1.158	1.093
compress95	1.193	1.115	1.126	1.074
li	1.189	1.118	1.159	1.093
perl	1.225	1.138	1.188	1.114

Our strategy is as follows: If a branch is mispredicted, the machine will access the I-Cache to fetch instructions. If a branch is predicted correctly, the machine will access the L0-Cache. In a misprediction, the pipeline will flush, and the machine will start fetching instructions from the correct address by accessing the I-Cache. In a correct prediction, the machine will start fetching instructions from the L0-Cache as soon as the branch is resolved. This might well be several instructions after the branch in a high-performance, superpipelined processor has executed.

Therefore, the instruction fetch unit (IFU) can be in either of two states at any given time: fetch from the L0-Cache, or fetch from the I-Cache. Only when a branch is resolved can the state of the IFU be modified. Therefore, a prediction in the IFU, which will be proven correct when the branch will be executed, does not trigger an L0-Cache access immediately, but only after the actual execution of the branch. In the meantime, the instructions that follow the branch will be fetched from wherever they used to be fetched.

Tables 2 and 3 show the normalized energy and delay results for the SPEC95 benchmarks. We denote the energy dissipation and the execution time of the original configuration that uses no extra caches as unity, and normalize everything else with respect to that. Our model accounts for all possible stalls in the R-4400 CPU, which is used as the base machine. The delay increase is due to the relatively high miss ratio on the L0-Cache.

Numeric code has better energy gains and smaller performance degradation than integer code. This is because there is a smaller number of basic blocks in numeric code that contribute significantly to the execution time of the program, and, thus, there is less contention in the L0-Cache. Also, the branch predictor has a smaller prediction rate for integer benchmarks; thus, the L0-Cache will not be utilized as frequently as in the case of FP benchmarks. However, the energy gains for non-numeric code is also very significant, much more than the method presented in [3].

5.3 Using a confidence estimator

As we show in Fig. 7, branch predictors are not always able to give a correct prediction. Therefore, we need a confidence estimation mechanism which, coupled with the branch predictor, gives a better insight into the behavior of the branch.

In this scheme, the confidence of each branch is determined dynamically. We use the prediction of each one of the component predictors (the bimodal and the global) to determine the confidence. If both predictors are strongly biased in the same direction (both "strongly taken" or both "strongly not-taken"), we signal a "high confidence" branch. In any other case, we signal a "low confidence" branch. This methodology uses a minimal amount of extra hardware and has been shown to be reliable in [15].

We manage the access of the L0-Cache as follows: If a "high confidence" branch was predicted incorrectly, the I-Cache is accessed for the subsequent basic blocks. Moreover, if more than two "low confidence" branches have been decoded, the I-Cache is accessed. In any other case, the machine accesses the L0-Cache.

The first rule for accessing the I-Cache is due to the fact that a mispredicted "high confidence" branch behaves "unusually" and probably drives the program to an infrequently executed portion of the code. The second rule is due to the fact that a series of "low confidence" branches will also suffer from the same problem since the probability that they are all predicted correctly is low. The number of successive "low confidence" branches is a parameter of the method. If a larger number of "low confidence" branches is used as a parameter, more basic blocks will be accessed from the L0-Cache.

Tables 4 and 5 show the normalized energy and delay results for the SPEC95 benchmarks.

Table 4: Energy results for the method that uses the confidence estimator.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	0.181	0.174	0.096	0.119
swim	0.123	0.134	0.099	0.118
su2cor	0.208	0.188	0.139	0.149
hydro2d	0.125	0.137	0.090	0.114
applu	0.369	0.293	0.338	0.276
fpppp	0.572	0.361	0.564	0.357
go	0.642	0.548	0.609	0.529
m88ksim	0.432	0.311	0.379	0.284
gcc	0.546	0.432	0.505	0.406
compress95	0.416	0.329	0.308	0.264
li	0.435	0.344	0.386	0.303
perl	0.503	0.382	0.440	0.340

Table 5: Delay results for the method that uses the confidence estimator.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	1.046	1.029	1.008	1.006
swim	1.029	1.017	1.015	1.008
su2cor	1.059	1.034	1.025	1.014
hydro2d	1.019	1.013	1.006	1.004
applu	1.104	1.053	1.089	1.045
fpppp	1.237	1.120	1.232	1.117
go	1.149	1.085	1.130	1.074
m88ksim	1.185	1.104	1.156	1.089
gcc	1.186	1.111	1.163	1.096
compress95	1.192	1.114	1.119	1.071
li	1.194	1.122	1.164	1.097
perl	1.232	1.142	1.194	1.117

5.4 Another method using a confidence estimator

The methods described in the previous sections tend to place a large number of basic blocks in the L0-Cache, thus degrading performance. In modern processors, one would prefer a more selective scheme in which only the really important basic blocks would be selected for the L0-Cache.

We use the same setup as before, but the selection mechanism is slightly modified as follows: the L0-Cache is accessed only if a "high confidence" branch is predicted correctly. The I-Cache is accessed in any other case.

Table 6: Energy results for the modified method that uses the confidence estimator.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	0.202	0.183	0.119	0.141
swim	0.129	0.140	0.105	0.124
su2cor	0.256	0.248	0.205	0.219
hydro2d	0.138	0.151	0.105	0.130
applu	0.558	0.498	0.532	0.483
fpppp	0.602	0.405	0.595	0.401
go	0.800	0.758	0.783	0.748
m88ksim	0.473	0.361	0.419	0.334
gcc	0.694	0.627	0.667	0.598
compress95	0.563	0.498	0.486	0.452
li	0.601	0.529	0.560	0.498
perl	0.602	0.508	0.552	0.447

Table 7: Delay results for the modified method that uses the confidence estimator.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	1.046	1.024	1.009	1.005
swim	1.028	1.017	1.015	1.008
su2cor	1.041	1.023	1.015	1.008
hydro2d	1.019	1.012	1.005	1.003
applu	1.082	1.043	1.069	1.035
fpppp	1.222	1.113	1.218	1.110
go	1.073	1.044	1.063	1.038
m88ksim	1.171	1.096	1.142	1.081
gcc	1.117	1.072	1.103	1.056
compress95	1.146	1.087	1.093	1.056
li	1.149	1.092	1.123	1.073
perl	1.190	1.119	1.159	1.098

This method selects some of the very frequently executed basic blocks, yet it misses some others. Usually the most frequently executed basic blocks come after "high confidence" branches that are predicted correctly. This is especially true in FP benchmarks.

Again, Tables 6 and 7 present the normalized energy and delay results. As before, the delay results consider all the possible stalls in the R-4400 processor.

As expected, this scheme is more selective in storing instructions in the L0-Cache, and it has a much lower performance degradation, at the expense of lower energy gains. It is probably preferable in a system where performance is more important than energy.

5.5 Comparison of dynamic techniques

The normalized energy and delay results of the three different schemes we proposed as well as the L0-Cache without any support (filter cache) are shown graphically in Figs. 8 and 9, respectively. A 512 bytes L0-Cache with a block size of 16 bytes is assumed in all cases. The graphical comparison of the results can be used to extract useful information about each one of the three methods.

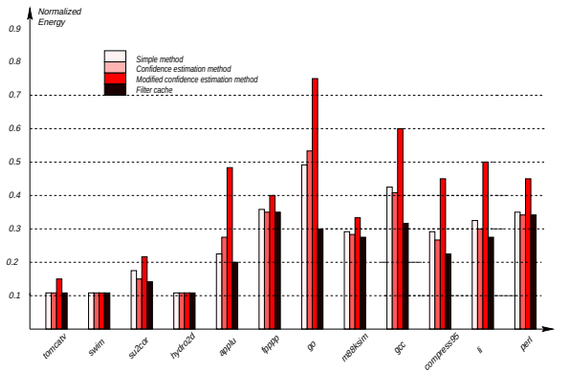

Figure 8: Normalized energy dissipation for the three dynamic methods and the filter cache. These are the same numbers that appeared in the tables of the previous sections.

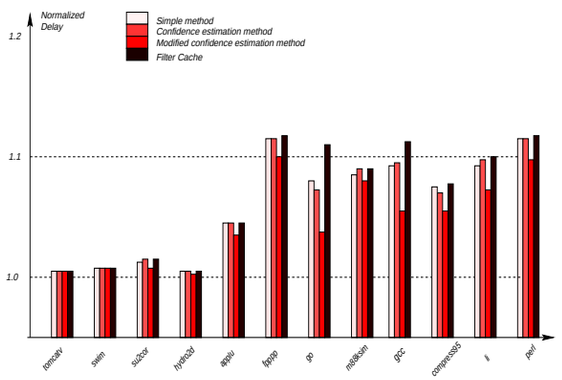

Figure 9: Normalized delay for the three dynamic methods and the filter cache. These are the same numbers that appeared in the tables of the previous sections.

The last "dynamic" method is the most successful in reducing the performance overhead, but the least successful in energy gains. The modified method that uses the dynamic confidence estimator poses stricter requirements for a basic block to be selected for the L0-Cache than the original dynamic confidence method. The filter cache has larger energy gains at a cost of larger performance overhead. The dynamic management of the L0-Cache attempts to regulate the traffic in the L0-Cache so that the excessive number of conflict misses of the filter cache is reduced.

The numeric benchmarks show the largest potential for energy gains without a severe performance penalty. The dynamic techniques have a larger impact on the integer benchmarks as is shown in the two graphs. Since a large percentage of branches are "low confidence" in integer benchmarks, the machine can be very selective when it picks up basic blocks for the L0-Cache. This is why different dynamic techniques have so different energy and delay characteristics for the integer benchmarks. Regulation of the L0-Cache utilization is more flexible in these programs.

Although the proposed dynamically-managed L0-Cache is not as energy efficient as the filter cache, it does not suffer from performance degradation as the filter cache does, and therefore it may be more suitable for high-performance processors designs.

6 Conclusion

In this work, we presented methods for "dynamic" selection of basic blocks for placement in an extra, small cache that is inserted between the L0-Cache and the pipeline. Since the small L0-Cache becomes the primary cache of the CPU, we need to place instruction within it selectively in order to reduce the possibility of a miss. Moreover, the "dynamic" techniques need minimal extra hardware for the final solution to have important energy gains with respect to the original scheme.

We are currently investigating further improvements in the scheme, by using "dynamic" techniques with different confidence estimation and branch prediction mechanisms. In addition we are looking into L0-Caches with associativity larger than one. Associativity becomes important for small caches since the miss rate drops dramatically.

References

[1] J. Diguet, S. Wuytack, F. Catthoor, and H. De Man, "Formalized methodology for data reuse exploration in hierarchical memory mappings," in *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 30–35, Aug. 1997.

[2] J. Kin, M. Gupta, and W. Mangione-Smith, "The filter cache: An energy efficient memory structure," in *Proceedings of the International Symposium on Microarchitecture*, pp. 184–193, Dec. 1997.

[3] 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 *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 70–75, Aug. 1998.

[4] S. Manne, D. Grunwald, and A. Klauser, "Pipeline gating: Speculation control for energy reduction," in *Proceedings of the International Symposium of Computer Architecture*, pp. 132–141, 1998.

[5] Teresa Johnson and Wen-mei Hwu, "Run-time adaptive cache hierarchy management via reference analysis," in *Proceedings of the International Symposium of Computer Architecture*, pp. 315–326, 1997.

[6] J. Hennesy and D. Patterson, *Computer Architecture–A Quantitative Approach*. San Francisco, CA: Morgan Kaufmann, 1996.

[7] T. Y. Yeh and Y. N. Patt, "Alternative implementations of a two-level adaptive branch prediction," in *Proceedings of the International Symposium of Computer Architecture*, pp. 124–134, 1992.

[8] S. McFarling, "Combining branch predictors," tech. rep., DEC WRL 93/5, June 1993.

[9] T. Y. Yeh and Y. N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in *Proceedings of the International Symposium of Computer Architecture*, pp. 257–266, 1993.

[10] E. Jacobsen, E. Rotenberg, and J. Smith, "Assigning confidence to conditional branch prediction," in *Proceedings of the International Symposium on Microarchitecture*, pp. 142–152, 1996.

[11] J. E. Veenstra and R. J. Fowler, "MINT: A front end for efficient simulation of shared-memory multiprocessors," in *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 201–207, 1994.

[12] *SpeedShop User's Guide*. Silicon Graphics, Inc., 1996.

[13] S. Wilson and N. Jouppi, "An enhanced access and cycle time model for on-chip caches," tech. rep., DEC WRL 93/5, July 1994.

[14] N. Bellas, I. Hajj, and C. Polychropoulos, "A detailed, transistor-level energy model for SRAM-based caches," in *Proceedings of the International Symposium on Circuits and Systems*, 1999.

[15] D. Grunwald, A. Klauser, S. Manne, and A. Plezskun, "Confidence estimation for speculation control," in *Proceedings of the International Symposium of Computer Architecture*, pp. 122–131, 1998.