Using dynamic cache management techniques to reduce energy in general purpose processors

Nikolaos Bellas, Ibrahim Hajj, and Constantine Polychronopoulos

Abstract— **The memory hierarchy of high-performance and embedded processors has been shown to be one of the major energy consumers. For example, the Level-1 (L1) instruction cache (I-Cache) of the StrongARM processor accounts for 27% of the power dissipation of the whole chip [1], whereas the instruction fetch unit (IFU) and the I-Cache of Intel's Pentium Pro processor are the single most important power consuming modules with 14% of the total power dissipation [2]. Extrapolating current trends, this portion is likely to increase in the near future, since the devices devoted to the caches occupy an increasingly larger percentage of the total area of the chip.**

In this paper, we propose a technique that uses an additional mini cache, the *L0-Cache*, located between the 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. We propose, implement, and evaluate five techniques for dynamic analysis of the program instruction access behavior, which is 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. Results for these benchmarks indicate that more than 60% of the dissipated energy in the I-Cache subsystem can be saved.

Keywords— **Low-power-design, Memory, Performance-tradeoffs, System-level**

I. INTRODUCTION

In recent years, power dissipation has become a major design concern for the microprocessor industry. The shrinking device size, and the large number of devices packed in a chip die coupled with the large operating frequencies, have led to unacceptably high levels of power dissipation.

The problem appears to be more acute in portable systems which operate under the energy constraints of a battery. These systems need a small battery for portability, which should provide enough energy to keep the system running for as long as possible. Hence, the energy dissipation of the portable system should be low so that it does not drain the battery quickly. On the other hand, desktop systems operate in an increasingly "hot" environment. The layout compaction and the high operating frequencies entail high power densities and, thus, high thermal stresses on the chip. If this trend is not controlled by low power techniques, the chip will have reliability problems, such as electromigration, which is due to the high current densities on the metal interconnects. Such low-power techniques also help in keeping the cost of packaging low, thus reducing the cost of the final product.

The problem of the wasted power caused by unnecessary activity in various parts of the CPU during code execution has traditionally been ignored in code optimization and architecture design. Processor architects and compiler writers are concerned with system performance/throughput and they do little, if anything at all, to eliminate energy/power dissipation at this level. Researchers in the CAD community have started tackling the problem of power minimization through compiler transformations, yet this process is still in its infancy. On the other hand, power dissipation is rapidly becoming the major bottleneck in today's systems integration and reliability.

Modern microprocessors are large power consumers: Table I shows clearly the power increase for the faster versions of the same processor families. Higher frequencies and larger transistor counts more than offset the lower voltages and smaller devices, and they result in larger power consumption in the newest version in a processor family. This has prompted many manufacturers to design low-power versions of their flagship processors for use in the mobile and multimedia computing industry. Clearly, designing a low-power, high-performance processor is considered an extremely hard problem, which can only be solved if power or energy reduction is a concern from the beginning of the design process and not only an afterthought.

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 on the on-chip I-Cache subsystem.

In order to reduce the effective energy dissipation per instruction access, we propose the addition of an 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. In our case, the L0-Cache is a small, direct-mapped cache which can accommodate 64 to 128 instructions, and has a block size of 8 to 16 bytes. 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 and it will bypass the L0-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 L0-Cache and the I-Cache are always accessed in sequence and never in parallel, thus avoiding redundant accesses and reducing energy dissipation.

This work was supported by Intel Corp., Santa Clara, CA from 1997 to 1999

N. Bellas is with the DigitalDNA Systems Architecture Laboratory, Motorola Inc., Schaumburg, IL

I. Hajj and C. Polychronopoulos are with the Coordinated Sciences Laboratory, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801

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

TABLE I
POWER TRENDS FOR CURRENT MICROPROCESSORS.

	DEC 21164	DEC 21164 Higher freq.	Pentium Pro	Pentium II	Ultra SPARCI	Ultra SPARCII
SPECint95	13.3	18.0	8.20	11.9	7.7	12.1
SPECfp95	18.3	27.0	6.21	8.82	11.4	15.5
Average	15.8	22.5	7.21	10.36	9.55	13.8
Freq. (MHz)	433	600	200	300	200	296
λ (μm)	0.35	0.35	0.6	0.35	0.45	0.35
Voltage (V)	2.4	2.4	3.3	2.0	3.3	3.3
Power (W)	32.5	45	28.1	41.4	30 (max)	58 (max)

Fig. 5 shows the schematic of a CPU with the L0-Cache placed between the pipeline and the L1 I-Cache. Our design objective is to exploit the instruction reuse present in a typical program, by placing the most frequently portions of the code in the L0-Cache. Since the small L0-Cache is the primary cache of the system, dynamic management techniques are employed to reduce the negative effects of excessive miss rates.

The paper is organized as follows: in section II, 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 III we briefly describe the hardware that we use, and, in section IV, we explain the basic idea behind our scheme. Section V 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 V. The conclusions are presented in section VI.

II. 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 [3] and [4]. In [5] the same authors propose a novel way to organize complex data structures in the memory hierarchy so that a cost function is minimized.

A model that views power from the standpoint of the software that executes on a microprocessor and the activity that it causes, rather than from the traditional hardware standpoint has been proposed [6] and tested in different architectures [7]. This methodology attempts to relate the power consumed by a microprocessor to the software that executes on it. In [8], the authors report that the energy-delay product in a wide spectrum of processors is relatively constant, although energy and delay varied by orders of magnitude. For example, the energy-delay product of the low-power, low-performance R-4600 from MIPS was almost the same as the energy-delay product of the powerful 21164 from DEC.

In [9], a mechanism is described which enables the by-pass of the I-Cache by storing the most frequently accessed instructions in an extra buffer. In [10], the authors propose methods to eliminate the tag comparisons in a cache access when the same cache block is accessed. In [11], the authors demonstrate that some popular hardware techniques that have been proposed to improve the hit rates of caches, such as the victim cache, have a beneficial impact on power as well.

The filter cache [12] 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. Provided that the working set of the program is relatively small, and that the data reuse is large, this "mini" cache can provide the data and instructions of the program and effectively shut down the L1 caches for long periods during program execution. The penalty to be paid is the increased miss rates and, hence, longer average memory access time. Although this might be acceptable for embedded systems for multimedia or mobile applications, it is out of the question for high-performance processors. Our method uses an extra, smaller cache between the the CPU and the L1 I-Cache just as the filter cache does. However, we use a series of dynamic management techniques to regulate the access of the L0-Cache. Contrary to the filter cache, our method does not access the L0-Cache for every instruction, but only when a series of criteria are satisfied. When they are not, the I-Cache is accessed immediately, and the L0-Cache is bypassed.

The concept that an intelligent RAM (IRAM) organization is inherently low power is discussed in [13] and [14]. In [15], 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 Loop-Cache. Profile information and the *control flow graph* of the program are used for this purpose.

The work in [2] 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 authors use the concept of branch prediction and confidence estimation [16] to detect when the CPU fetches and executes instructions from a speculative path that has a small probability to be taken. 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 dynamic techniques. This effort has almost always targeted delay rather than energy reduction. The authors in [17] and [18] 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. Extra hardware is used to keep statistics about the frequency of access of the data used in the program.

The techniques proposed in [17] and [18] 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 reduction 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.

III. BRANCH PREDICTION AND CONFIDENCE ESTIMATION–A BRIEF OVERVIEW

In this section we give an overview of branch prediction and confidence estimation and we focus on the mechanisms that will be useful in the context of our work.

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. To fully exploit the potential of the very powerful CPU cores, programs need to have as few branches as possible. Various compiler techniques, such as loop unrolling, can help towards that direction, yet they cannot fully solve the problem in integer programs, which have few loops and small basic blocks.

Branch prediction is an important technique 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. In such a case, the pipeline only executes from a straight line code and can fetch and issue more than one instruction per clock cycle.

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.

A. Previous work on branch prediction

Successful branch prediction mechanisms take advantage of the non-random nature of branch behavior [19]. Most branches are either mostly taken or mostly not taken in the course of program execution. Moreover, the behavior of a branch usually depends on the behavior of the surrounding branches in the program.

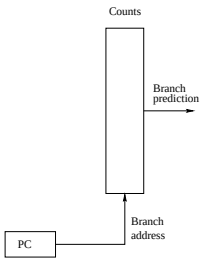

Fig. 1. Bimodal branch predictor. Each entry in the table is a 2-bit saturated counter.

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.

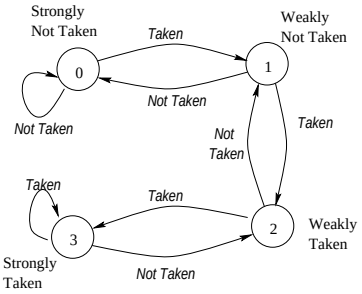

Fig. 2. FSM for the 2-bit saturated counters.

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 causing loss of prediction accuracy. 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. The latter access can happen much later in the pipeline in an out-of-order, superpipelined execution core.

Global branch predictor. In the bimodal branch prediction method, only the past behavior of the current branch is considered. Another scheme is proposed in [20] 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.

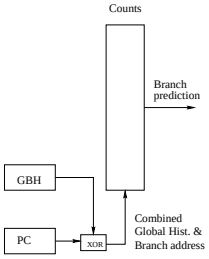

Fig. 3. Global branch predictor with index sharing.

McFarling branch predictor. Finally, McFarling [21] 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 [22] 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

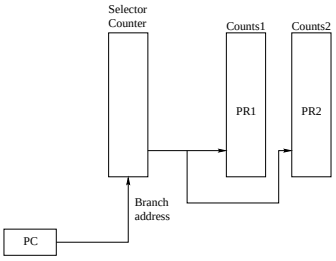

Fig. 4. McFarling branch predictor.

for the prediction and is updated after the direction of the branch has been resolved. The counter is biased towards the predictor that has given most correct predictions in the past for that particular branch.

B. 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 [16].

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.

Various confidence estimation techniques have been proposed in [16] and [23]. Unlike the branch predictors, whose performance can be easily measured using the prediction rate, the confidence estimators are not easy to characterize. Different confidence estimators can be useful for different applications. We will describe the confidence estimation techniques used in our paper later in section V.

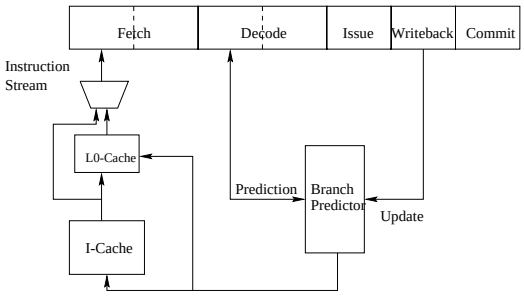

Fig. 5. Pipeline microarchitecture

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.

IV. BASIC IDEA OF THE DYNAMIC MANAGEMENT SCHEME

In this section, we delineate our approach for dynamic management of the L0-Cache. We show how the information that is accumulated by the branch prediction mechanism during program execution can be used as a guiding mechanisms to access or bypass the L0-Cache.

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.

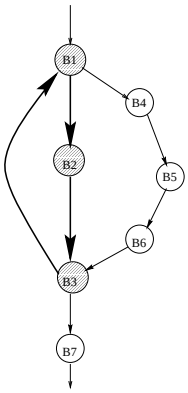

Fig. 6. An "unusual" branch direction leads to a rarely executed portion of the code.

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.

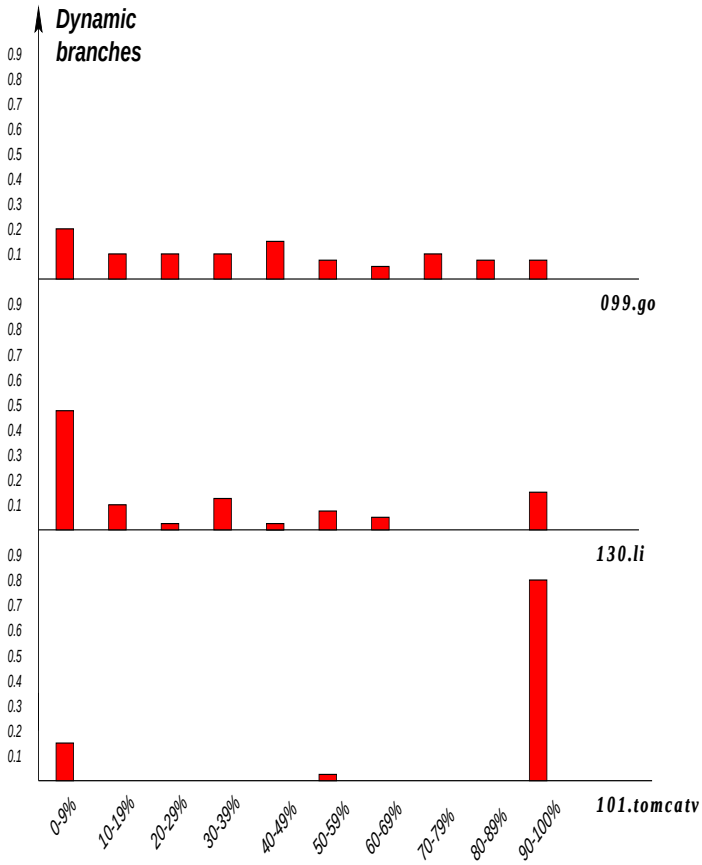

Fig. 7. Branch direction percentages for three SPEC95 benchmarks.

Not all branches can be characterized as "high confidence." In Fig. 7, the dynamic branches are classified according to how many times they are "taken." For example, the height of the column 30-39% gives the percentage of the dynamic branches that are "taken" from 30 to 39% percent of the time. For the 130.*li* benchmark, almost 11% of all the dynamic branches were "taken" between 30% and 39% of the times they are executed. 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.

This experiment shows that different benchmarks demonstrate different branch behavior. For most integer, and many FP benchmarks there are many branches that cannot be classified as "mostly taken" or as "mostly not taken." These benchmarks can be "mostly taken" or "mostly not taken" in different stages of the execution, or they may follow a more random pattern.

Branch predictors have a much easier task in programs like *101.tomcatv* in which the behavior of branches is predictable, than in *099.go* where it is more erratic. For a lot of integer programs, branches do not demonstrate a bimodal behavior. In Fig. 8, the branch misprediction rates for most of the SPEC95 benchmarks are shown.

Programs whose branches are evenly distributed in all the columns of Fig. 7 suffer from a large misprediction rate even with the McFarling predictor. On the other hand, programs whose dynamic branches are concentrated near the edges do not have such problems. In general, branch predictors are quite successful with numerical or computation-intensive code (like the *101.tomcatv*).

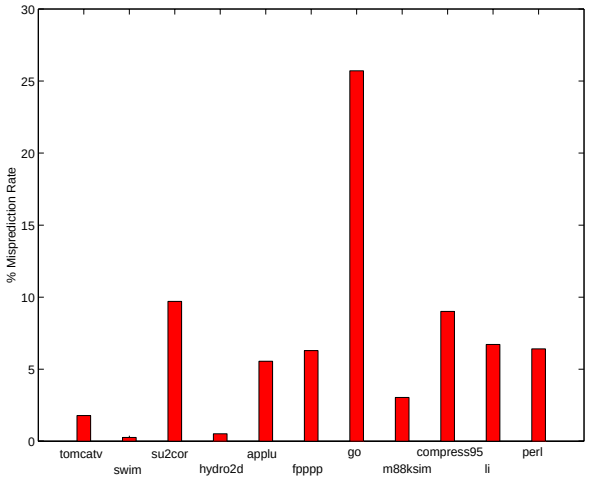

Fig. 8. Misprediction rate for the SPEC95 benchmarks.

V. DYNAMIC TECHNIQUES FOR SELECTING BASIC BLOCKS FOR THE L0-CACHE

In this section we describe the particular schemes that implement the basic idea as this was outlined in section IV. First, we describe the experimental setup, and then, we continue with the five different methods that implement the dynamic L0-Cache management concept. The last part of this section is devoted to the comparison between the five methods, and the energy/delay trade-offs. The energy models used for the experimental evaluation can be found in the Appendix.

A. 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* [24] and the *SpeedShop* [25] 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. This is a single-issue, in-order, pipelined machine with eight stages. The simulator accounts for all the stall because of structural and data hazards in the machine, as well as for stalls because of cache misses and branch mispredictions. Structural hazards are mainly a problem for the floating point unit of the R-4400.

Table II shows the latencies (in clock cycles) of the functional units of our simulator based on [26], and Table III 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 [27]. We use the tool *cacti*, described in [27], to estimate

the access time of the on-chip caches, as well as the optimal banking that minimizes the access time.

TABLE II
FUNCTIONAL UNITS LATENCY.

Resource	Latency
Integer ALU	1
Integer MULT	12
Integer DIV	76
FP ADD/SUB	4
FP MULT (single)	7
FP MULT (double)	8
FP DIV (single)	23
FP DIV (double)	36
FP SQRT (single)	54
FP SQRT (double)	112
FP CONVERT	2–6

TABLE III
MEMORY SUBSYSTEM CONFIGURATION IN THE BASE MACHINE.

Parameter	size/Block size/A/Cycle time/L2 cache/band.
L1 I-Cache	32KB/32/1/1/4/8
L1 D-Cache	32KB/32/2/1/4/8

In our experiments the L0-Cache is implemented as a direct-mapped cache, with size ranging from 256 to 512 bytes, and block size from 8 to 16 bytes. The dynamic management scheme regulates the access to the L0-Cache as we will describe shortly.

B. Energy models for the cache subsystem

We have developed our cache energy model based on the work by Wilson and Jouppi [27] in which they propose a timing analysis model for SRAM-based caches [28]. 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.). These models are used for the estimation of energy in both the I-Cache and the L0-Cache.

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 [27]. Details for the cache energy modeling are presented in the Appendix.

In the following subsections, we will present the five methods for the dynamic management of the L0-Cache. We will examine how the energy reduction can be traded for less performance degradation, and how the extra information about the branch as this is given by the confidence estimation mechanisms can improve both delay and energy. The five methods are: the *simple method* in which no confidence estimation mechanism is used, the *static method*, the *dynamic confidence estimation method*, the *restrictive dynamic confidence estimation method*, and the *dynamic distance estimation method*.

C. Simple method (without confidence estimator)

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 IV
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 V
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 been decoded.

Tables IV and V 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 L0-Cache 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.

In addition, we account for a branch misprediction stall, which is equal to two clock cycles. The delay increase is due to the relatively high miss ratio on the L0-Cache. The energy

results in this and the next sections refer only to the instruction memory hierarchy subsystem and not the whole processor. The delay is the execution time of the program. Therefore, the complete picture of the effects of this method on the execution profile is captured.

Numeric code has more energy reduction 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 reduction for non-numeric code is also very significant. Table VI shows the percentage of dynamic instructions that cause the L0-Cache to be accessed irrespective if the access was a hit or a miss, as well as the hit rate of the L0-Cache. In this experiment as well as the next ones, the L0-Cache is 512 bytes with a block size of 16 bytes.

TABLE VI
DYNAMIC INSTRUCTIONS THAT CAUSE THE L0-CACHE TO BE ACCESSED AND THE L0-CACHE HIT RATIO IN THE SIMPLE METHOD.

Benchmark	% dyn. instructions	Hit ratio in L0-Cache
tomcatv	99.47%	98.66%
swim	99.95%	98.50%
su2cor	95.01%	97.41%
hydro2d	99.58%	92.37%
applu	95.94%	99.98%
fpppp	98.00%	88.50%
go	73.09%	96.69%
m88ksim	97.56%	83.33%
gcc	84.29%	93.11%
compress95	91.69%	99.98%
li	93.11%	99.97%
perl	93.57%	79.63%

D. Static method

The next technique we used to select basic blocks for the L0-Cache is not dynamic. We used profiling, and simulated the branch predictor. We captured the behavior of the branches of the program and we classified them as "high confidence" if they were predicted correctly most of the time, and "low confidence" if not. A threshold was used to determine confidence, so that the branches that were predicted correctly at least 90% of the time were tagged as "high confidence," whereas all other branches were tagged as "low confidence." The static method is not implementable, since it it is not easy to communicate to the machine whether a particular branch is "low confidence" or "high confidence" without changing the instruction opcode. We include this approach to indicate its potential.

After profiling, we ran the benchmarks again and we selected the basic blocks as follows: If a "high confidence" branch was predicted incorrectly, the I-Cache is accessed for the subsequent basic blocks. Moreover, if more than N "low confidence" branches have been decoded in a row, the I-Cache is accessed. Therefore, the L0-Cache will be bypassed when either of these two conditions is satisfied. 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 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. A series of "low confidence" branches form a path which might not have been taken very often in the past.

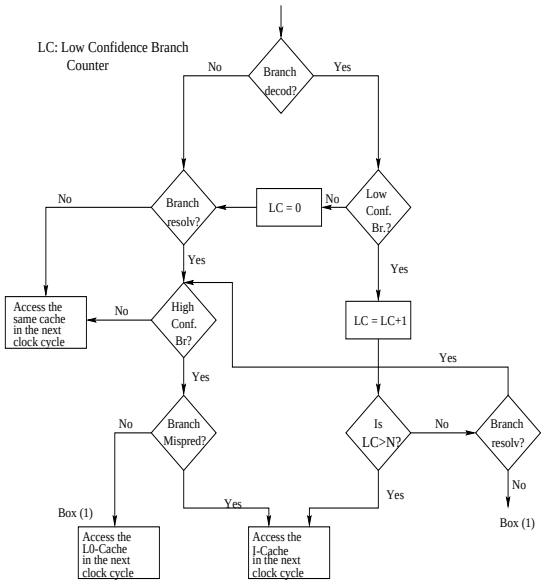

Fig. 9. Decision diagram for the cache access in the static method

There are two controlling parameters in the static method; the threshold used to classify a branch as "high" or "low confidence" and the number of successive "low confidence" branches which need to be decoded before the machine turns to the I-Cache. A larger threshold of successive "low confidence" branches results in more basic blocks accessed from the L0-Cache. In our experiments, an empirical value of $N = 2$ was used.

Fig. 9 shows the decision diagram that the mechanism follows implicitly in each clock cycle to decide whether to access the L0-Cache or the I-Cache in the next clock cycle.

Tables VII and VIII show the normalized energy and delay results for the SPEC95 benchmarks. The same experiments and the same experimental framework was used here. The results we present are from self-profiled executions where the same input was used to profile and evaluate our approach.

The problem with the static method is that it does not exploit the temporalities of the branches, but it only assigns a confidence to them statically. It has similar performance to the simple method, except for the *099.go* benchmark for which it has much lower delay and much higher energy dissipation. This is because this particular benchmark has a large number of "low confidence" branches. Table IX presents the percentage of dynamic instructions which cause the CPU to access the L0-Cache.

E. Dynamic confidence estimation method

A dynamic version of the static method is presented in this subsection. As we showed in Fig. 8, 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 intuition about the behavior of the branch.

We are using a similar methodology as in the static method,

TABLE VII
ENERGY RESULTS FOR THE STATIC METHOD.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	0.185	0.163	0.100	0.120
swim	0.123	0.134	0.099	0.118
su2cor	0.201	0.194	0.130	0.154
hydro2d	0.124	0.137	0.091	0.114
applu	0.309	0.231	0.269	0.208
fpppp	0.572	0.361	0.562	0.355
go	0.757	0.702	0.736	0.690
m88ksim	0.431	0.309	0.378	0.283
gcc	0.596	0.493	0.557	0.470
compress95	0.410	0.315	0.294	0.246
li	0.431	0.337	0.382	0.296
perl	0.520	0.393	0.531	0.450

TABLE VIII
DELAY RESULTS FOR THE STATIC METHOD.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	1.049	1.025	1.011	1.006
swim	1.029	1.017	1.015	1.008
su2cor	1.061	1.033	1.027	1.013
hydro2d	1.021	1.013	1.007	1.004
applu	1.109	1.057	1.089	1.045
fpppp	1.239	1.121	1.234	1.118
go	1.090	1.052	1.078	1.045
m88ksim	1.187	1.104	1.158	1.089
gcc	1.167	1.100	1.146	1.087
compress95	1.205	1.121	1.126	1.074
li	1.198	1.124	1.169	1.099
perl	1.226	1.138	1.158	1.095

but no profiling information is used. Instead, the confidence of each branch is determined dynamically using the *saturating counters* approach. In that approach, 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. An added advantage of this method is that it uses a minimal amount of extra hardware.

The management of the cache subsystem is identical to the static method. We access the I-Cache if a "high confidence" branch is mispredicted, or more than N successive "low confidence" branches are encountered. The schematic of the modified pipeline is shown in Fig. 10. The "Low Confidence Branch Counter" will trigger a subsequent I-Cache access if its value exceeds a predefined threshold N ($N = 2$ in our experiments). The I-Cache will also be accessed in case of a misprediction of a "high confidence" branch.

Again, Fig. 9 describes the decision process for the access of the L0 or L1 caches. All the information necessary for the hardware to decide which cache to access in the next clock cycle is available from the branch prediction mechanism and the decoder of the CPU. The extra hardware needed is a number of extra gates to compute the confidence "on-the-fly", a multiplexer

TABLE IX
DYNAMIC INSTRUCTIONS THAT CAUSE THE L0-CACHE TO BE ACCESSED AND THE HIT RATIO OF THE L0-CACHE IN THE STATIC METHOD.

Benchmark	% dyn. instructions	Hit ratio of the L0-Cache
tomcatv	99.55%	98.60%
swim	99.95%	98.5%
su2cor	97.45%	97.13%
hydro2d	99.69%	99.10%
applu	98.57%	90.71%
fpppp	99.02%	75.44%
go	43.52%	81.23%
m88ksim	98.59%	83.17%
gcc	77.30%	78.89%
compress95	96.42%	88.30%
li	97.08%	83.01%
perl	79.49%	79.90%

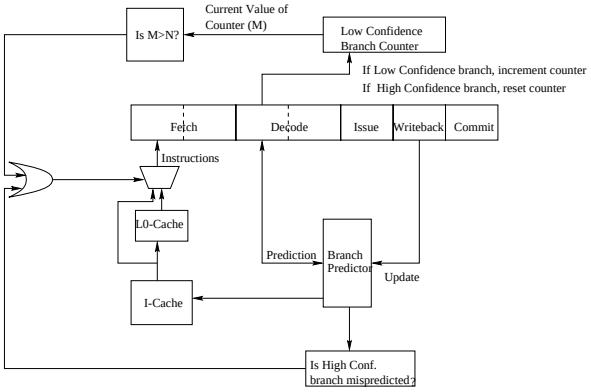

Fig. 10. Microarchitectural modifications for the confidence estimation method.

to redirect the instruction stream from the I-Cache or L0-Cache to the pipeline, and the "Low Confidence Branch Counter".

Tables X and XI show the normalized energy and delay results for the SPEC95 benchmarks. Table XII presents the percentage of dynamic instructions which cause the CPU to access the L0-Cache.

This method is slightly better in terms of energy reduction than the simple or the static method. The delay is lower than the previous two methods in some benchmarks and higher in some others. Since the confidence estimator can adapt dynamically in the temporalities of the code, it is more accurate in characterizing a branch and, then, regulating the access of the L0-Cache.

F. Restrictive dynamic confidence estimation method

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. 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

TABLE X
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 XI
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

TABLE XII
DYNAMIC INSTRUCTIONS THAT CAUSE THE L0-CACHE TO BE ACCESSED AND HIT RATIO IN THE L0-CACHE IN THE CONFIDENCE ESTIMATION METHOD.

Benchmark	% dyn. instructions	Hit ratio of the L0-Cache
tomcatv	99.77%	98.62%
swim	99.95%	98.49%
su2cor	98.26%	96.93%
hydro2d	99.77%	99.10%
applu	91.00%	90.00%
fpppp	98.91%	75.47%
go	67.50%	80.00%
m88ksim	98.26%	83.26%
gcc	86.54%	79.95%
compress95	93.78%	88.43%
li	95.91%	83.23%
perl	95.78%	79.55%

TABLE XIII
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

especially true in FP benchmarks.

Again, Tables XIII and XIV present the normalized energy and delay results. As before, the delay results consider all the possible stalls in the R-4400 processor. Table XV shows the percentage of the dynamic instructions which causes the CPU to access the L0-Cache.

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 reduction. It is probably preferable in a system where performance is more important than energy.

G. Dynamic distance estimation method

Another confidence estimator which was proposed in [23] exploits the clustering of mispredicted branches. As was shown experimentally in that paper, a mispredicted branch triggers a series of succesive mispredicted branches. The degree of clustering depends on the particular program, and the predictor used. This correlation fades as more branches are executed, and is stronger immediately after the mispredicted branch. The observation that branches that follow a mispredicted branch are more probable to be mispredicted can be exploited in our scheme.

We use a counter to measure the distance of a branch from the previous, mispredicted branch. This is similar to the *miss distance counter (MDC)* proposed in [23]. The method works as follows: all N branches after a mispredicted branch are tagged as "low confidence," otherwise as "high confidence." The basic blocks after a "low confidence" branch are fetched from the I-Cache, whereas the basic blocks after a "high confidence" branch are fetched from the L0-Cache. The net effect is that a branch misprediction causes a series of fetches from the I-Cache.

The parameter N was set equal to four in our experiments. It can be used to bias the mechanism towards larger energy reduction(smaller N) or higher performance (larger N). Fig. 11 shows the new pipeline that can support this scheme.

Again, Tables XVI and XVII present the normalized energy and delay results. Table XVIII shows the percentage of the dynamic instructions which causes the CPU to access the L0-Cache.

This scheme is also very selective in storing instructions in the L0-Cache, even more than the previous method. Provided that the L0-Cache is not too small to contain the working set of the program, this approach will be able to manage the L0-Cache such that only the basic blocks with the larger degree of reuse will be stored there.

H. Comparison of dynamic techniques

The normalized energy and delay results of the five different schemes we proposed and the filter cache proposed in [12] are

TABLE XIV
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

TABLE XV
DYNAMIC INSTRUCTIONS THAT CAUSE THE L0-CACHE TO BE ACCESSED AND HIT RATIO OF THE L0-CACHE IN THE MODIFIED CONFIDENCE ESTIMATION METHOD.

Benchmark	% dyn. instructions	Hit ratio of the L0-Cache
tomcatv	97.03%	98.81%
swim	99.30%	98.51%
su2cor	89.11%	98.00%
hydro2d	97.84%	99.21%
applu	65.70%	89.10%
fpppp	92.06%	75.32%
go	35.60%	80.48%
m88ksim	90.98%	83.51%
gcc	56.49%	84.54%
compress95	70.07%	87.79%
li	69.75%	82.57%
perl	77.37%	78.64%

shown graphically in Figs. 12 and 13, 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 these methods.

The energy reduction and the delay increase is a function of the algorithm used for the regulation of the L0-Cache access, the size of the L0-Cache, its block size, and its associativity. The effect that these parameters have on the energy and the performance profile of the new scheme is not always clear. For example, a larger block size causes a larger hit ratio in the L0-Cache. This results into smaller performance overhead, and bigger energy efficiency since the I-Cache does not need to be accessed so often. On the other hand, if the block size increase does not have a large impact on the hit ratio, the energy dissipation may go up, since a cache with a larger block size is less energy efficient than a cache with the same size but smaller block size. Therefore, the cumulative effect of all these parameters on the energy and delay characteristics of the processor is dependent on the program and its degree of instruction reuse.

Note that the second and third methods make the implicit assumption that the less frequently executed basic blocks usually follow less predictable branches or more predictable branches that are mispredicted. On the other hand, the first and fourth methods address the problem from another angle: they assume

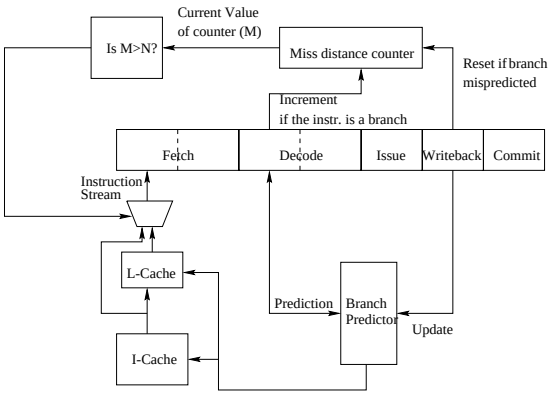

Fig. 11. Microarchitectural modifications for the distance estimator method.

TABLE XVI
ENERGY RESULTS FOR THE DISTANCE CONFIDENCE ESTIMATION METHOD.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	0.197	0.192	0.115	0.137
swim	0.126	0.137	0.103	0.122
su2cor	0.268	0.263	0.222	0.236
hydro2d	0.133	0.147	0.101	0.126
applu	0.430	0.356	0.399	0.338
fpppp	0.587	0.384	0.580	0.380
go	0.820	0.781	0.806	0.773
m88ksim	0.465	0.355	0.412	0.328
compress95	0.554	0.489	0.466	0.436
li	0.577	0.502	0.540	0.472
perl	0.569	0.481	0.531	0.450

that most frequently executed basic blocks usually follow highly predictable branches.

The last "dynamic" method is the most successful in reducing the performance overhead, but the least successful in energy reduction. The method that uses the restrictive dynamic confidence estimator poses stricter requirements for a basic block to be selected for the L0-Cache than the original dynamic confidence method.

The numeric benchmarks show the largest potential for energy reduction without a severe performance penalty. The FP benchmarks have a smaller number of basic blocks that are frequently executed compared with the non-numeric code. These blocks have a larger degree of reuse, and once they are stored in the L0-Cache, they will probably remain there for a long period of time. However, the basic blocks in FP benchmarks tend to be larger. If they are larger than the size of the L0-Cache (as in fpppp), the performance and energy results deteriorate rapidly.

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.

The filter cache shows the largest energy reduction, at the expense of a bigger performance overhead. This was expected since the filter cache is always accessed before the I-Cache is

TABLE XVII
DELAY RESULTS FOR THE DISTANCE CONFIDENCE ESTIMATION METHOD.

Benchmark	256 B		512 B	
	8 B	16 B	8 B	16 B
tomcatv	1.045	1.029	1.008	1.005
swim	1.028	1.017	1.015	1.008
su2cor	1.038	1.022	1.015	1.009
hydro2d	1.018	1.012	1.005	1.003
applu	1.100	1.051	1.085	1.043
fpppp	1.229	1.116	1.225	1.114
go	1.064	1.037	1.056	1.033
m88ksim	1.169	1.094	1.139	1.079
compress95	1.141	1.083	1.081	1.047
li	1.154	1.095	1.132	1.077
perl	1.188	1.114	1.158	1.095

TABLE XVIII
DYNAMIC INSTRUCTIONS THAT CAUSE THE L0-CACHE TO BE ACCESSED AND HIT RATIO OF THE L0-CACHE IN THE DISTANCE ESTIMATOR METHOD.

Benchmark	% dyn. instructions	Hit ratio of the L0-Cache
tomcatv	97.37%	98.89%
swim	99.54%	98.50%
su2cor	87.24%	97.85%
hydro2d	98.21%	99.23%
applu	83.54%	89.57%
fpppp	95.24%	75.42%
go	31.53%	81.42%
m88ksim	91.31%	83.91%
compress95	70.31%	89.96%
li	73.39%	82.55%
perl	79.49%	79.90%

accessed. It suffers from a large degree of cache pollution, but, on the other hand, it has the largest potential for energy savings if the working set of the program is small.

Larger block size and associativity will have a beneficial effect on both energy and performance. The hit rate of a small cache is more sensitive to the variation of the block size (as is shown in the results) and the associativity.

VI. CONCLUSION

In this paper, we presented methods for "dynamic" selection of basic blocks for placement in the L0-Cache. First, we presented an extensive overview of previous research in improving the memory hierarchy subsystem performance using dynamic techniques. Usually, extra hardware is used to keep statistics about the program execution and, accordingly, to allow or disallow the storage of specific instructions or data in the L1 caches.

Then, we proceeded by explaining the functionality of the branch prediction and the confidence estimation mechanisms in high-performance, speculative processors. After that, we explained how those mechanism can provide information to the CPU about the frequency of execution of parts of the code, and, finally, we presented five different "dynamic" techniques for the selection of the basic blocks. These techniques try to capture the execution profile of the basic blocks by using the branch statistics that are gathered by the branch predictor.

The experimental evaluation demonstrates the applicability of the dynamic techniques for the management of the L0-Cache.

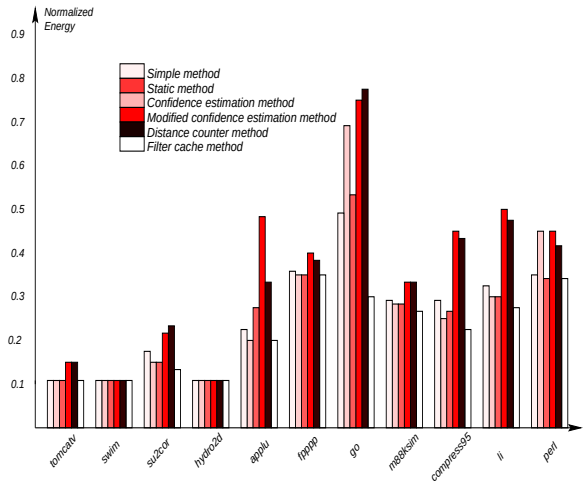

Fig. 12. Normalized energy dissipation for the five dynamic methods. Those are the same numbers that appeared in the tables of the previous sections.

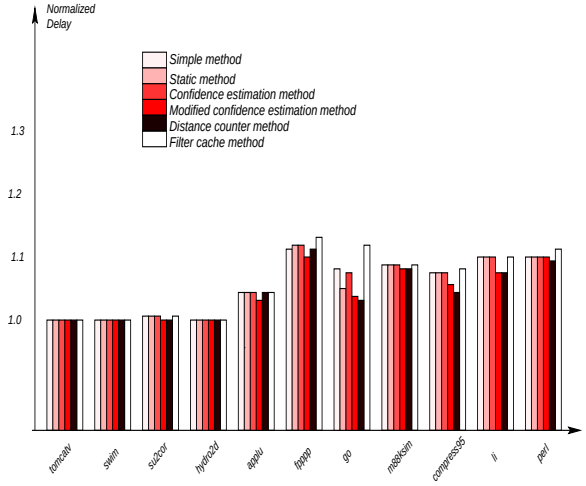

Fig. 13. Normalized delay for the five dynamic methods. Those are the same numbers that appeared in the tables of the previous sections.

Different techniques can trade off energy with delay by regulating the way that the L0-Cache is accessed. By varying the parameters of the method, the designer can choose from a wide range of caching policies.

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. In addition, techniques similar to those described in [17] and [18] are more accurate in locating frequently executed portions of the code, and they should also be assessed along the lines of reduced energy consumption.

APPENDIX

I. CACHE ENERGY MODELS

Fig. 14 shows the assumed internal cache organization. This is a very general model of an A-way set associative cache, with size of C bytes and a block size of B bytes. The operation of the cache is now briefly described.

The cache is organized as a collection of $S = \frac{C}{B \times A}$ sets, so

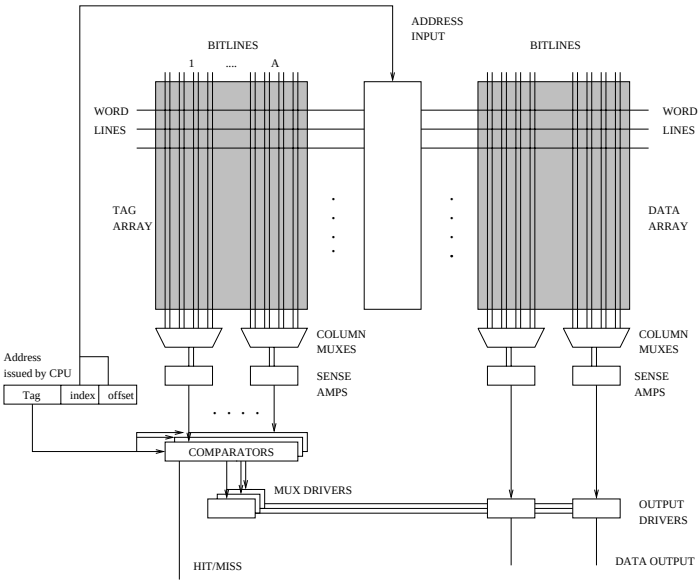

Fig. 14. Cache Model (taken from [27])

that one set contains A blocks, or $B \times A$ bytes. The CPU issues an address to the cache consisting of three parts: the tag, the index and the offset. The index part has length $\log_2(S)$ bits, and is used to index the set from which the data will be retrieved. The offset part has length $\log_2(B)$ bits, and is used to select the appropriate word within a block to return to the CPU. Finally, the tag part is used to check whether there is a hit or a miss in the cache.

The cache consists of two arrays used to store the tag and the actual data. Each one of them is organized as a series of rows and columns so that there is one CMOS Static RAM cell at the intersection of a row and two columns (the bitline and its complement). In Fig. [27] we assume that one row in the data array stores a single set. The decoder first selects a row from the tag and data array using the index and offset bits of the CPU address. Each bitline is first precharged high. When the decoder makes the selection, each memory cell in that row pulls down one of its two bitlines, depending on the value of the cell.

A set of sense amplifiers monitors small changes in the bitlines and transforms them into legitimate voltage values. Usually, a sense amplifier is shared among several pairs of bitlines. Extra column multiplexers are used in both arrays to implement this sharing.

The information read from the tag array is compared to the tag bits of the address issued by the CPU. There are A such comparators for an A-associative cache. The result of the comparison is used to drive the output bus with the data that have been read, in the meantime, from the data array.

In most of today's caches, the tag and data arrays are broken rowwise and columnwise so that the time to access the data is reduced. Three new parameters are defined for that purpose for each of the two arrays. The parameter N_{dwl} shows how many times the data array is split vertically resulting into more and shorter wordlines. The parameter N_{dbl} shows how many times the data array is split horizontally resulting into more and shorter bitlines. Finally the parameter N_{spd} indicates how many sets are mapped into a single row. The tag array can be broken independently according to the parameters N_{twl}, N_{tbl}, and N_{tspd}.

Using these organizational parameters, each data subarray has $\frac{8 \times B \times A \times N_{spd}}{N_{dwl}}$ columns and $\frac{C}{B \times A \times N_{dbl} \times N_{spd}}$ rows. The total number of data subarrays is $N_{dbl} \times N_{dwl}$.

We will show the equation for energy dissipation in the wordlines. Detailed modeling of the other components of Fig. 14 is given in [28]. In every clock cycle, a wordline will be charged and another one will be discharged (Fig. 15). The energy dissipation in the word lines is given by:

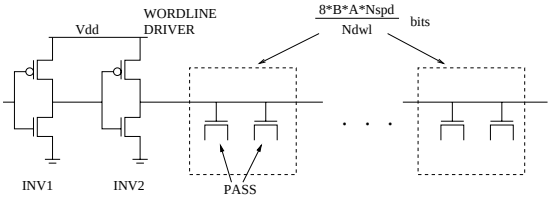

Fig. 15. Word line architecture

$$E_{wordline} = V_{dd}^2 \times N_{acc} \times N_{dwl} \times (C_{j,INV1} + C_{g,INV2}) + V_{dd}^2 \times N_{acc} \times (8 \times B \times A \times N_{spd} \times 2 \times C_{g,PASS} + C_{j,INV2} + 8 \times B \times A \times N_{spd} \times C_{wordmetal}) + V_{dd}^2 \times N_{acc} \times N_{twl} \times (C_{j,INV1} + C_{g,INV2}) + V_{dd}^2 \times N_{acc} \times [(T + St) \times N_{tspd} \times 2 \times C_{g,PASS} + C_{j,INV2} + (T + St) \times N_{tspd} \times C_{wordmetal}]$$

The gate capacitance of a device x is denoted as $C_{g,x}$ and its junction capacitance as $C_{j,x}$. The length of the tag is T bits, and there are also St bits for the status in each block. For example, the *valid* and the *dirty* bit are status bits. Each data subarray has $2 \times 8 \times B \times A \times N_{spd}$ pass transistors, and each tag subarray has $2 \times (T + St)$ pass transistors. Finally, N_{acc} is the number of accesses in the cache.

Our experiments in [28] show that the largest percentage of the energy is consumed for the precharge and discharge of the bitlines as well as for the transfer of data from the output drivers to the CPU or to the next level of the memory hierarchy. Low swing bitlines are extensively used for any low power implementation of SRAM-based caches. Also, shorter bitlines and wordlines are very beneficial in reducing power. That is why a smaller cache is also more energy efficient than a larger one.

References

[1] D. Dobberpuhl, "The design of a high-performance low-power microprocessor," in *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 11–16, 1996.

[2] 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.

[3] 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.

[4] S.Wuytack, F.Catthoor, L. Nachtergaele, and H. De Man, "Power exploration for data dominated video applications," in *Proceedings of the International Symposium of Low Power Electronics and Design*, 1996.

[5] S. Wuytack, F. Catthoor, and H. DeMan, "Transforming set data types to power optimal data structures," *IEEE Transcactions on Computer-Aided Design*, vol. 15, pp. 619–629, June 1996.

[6] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step towards software power minimization," *IEEE Transactions on VLSI Systems*, vol. 2, pp. 437–445, Dec. 1994.

[7] V. Tiwari, S. Malik, A. Wolfe, and T.C. Lee, "Instruction level power analysis and optimization of software," *Journal of VLSI Signal Processing*, vol. 13, Aug. 1996.

[8] R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose processors," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 1277–1284, Sept. 1996.

[9] R. Bajwa, M. Hiraki, H. Kojima, D. Gorny, K. Nitta, A. Shridhar, K. Seki, and K. Sasaki, "Instruction buffering to reduce power in processors for signal processing," *IEEE Transactions on VLSI Systems*, vol. 5, pp. 417–424, Dec. 1997.

[10] R. Panwar and D. Rennels, "Reducing the frequency of tag compares for low power I-Cache design," in *Proceedings of the International Symposium of Low Power Electronics and Design*, Aug. 1995.

[11] I. Bahar, G. Albera, and S. Manne, "Power and performance trafeoffs using various caching strategies," in *Proceedings of the International Symposium of Low Power Electronics and Design*, pp. 64–69, 1998.

[12] 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.

[13] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McGaughy, D. Patterson, T. Anderson, and K. Yelick, "The energy efficiency of IRAM architectures," in *Proceedings of the International Symposium of Computer Architecture*, pp. 327–337, 1997.

[14] J. Zawodny, E. Johnson, J. Brockman, and P. Kogge, "Cache-in-memory: A lower power alternative?," in *Proceedings of the Power-Driven Microarchitecture Workshop, ISCA*, pp. 67–72, 1998.

[15] 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.

[16] 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.

[17] 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.

[18] Teresa Johnson, Matthew Merten, and Wen-mei Hwu, "Run-time spatial locality detection and estimation," in *Proceedings of the International Symposium on Microarchitecture*, 1997.

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

[20] S. T. Pan, K. So, and J.T. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 76–84, 1992.

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

[22] 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.

[23] 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.

[24] 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.

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

[26] G. Kane and J. Heinrich, *MIPS RISC Architecture*. Englewood-Cliffs, NJ: Prentice Hall, 1992.

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

[28] 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.