Reducing Energy in Instruction Caches by Using Multiple Line Buffers with Prediction

Kashif Ali Mokhtar Aboelaze Suprakash Datta
Department of Computer Science and Engineering
York University
Toronto ON CANADA
email: {kashif, aboelaze, datta}@cs.yorku.ca

***Abstract*— Energy efficiency plays a crucial role in the design of embedded processors especially for portable devices with its limited energy source in the form of batteries. Since memory access (either cache or main memory) consumes a significant portion of the energy of a processor, the design of fast low-energy caches has become a very important aspect of modern processor design. In this paper, we present a novel cache architecture for reduced energy instruction caches. Our proposed cache architecture consists of the L1 cache, multiple line buffers, and a prediction mechanism to predict which line buffer, or L1 cache to access next. We used simulation to evaluate our proposed architecture and compare it with the HotSpot cache, Filter cache, Predictive line buffer cache and Way-Halting cache. Simulation results show that our approach can reduce instruction cache energy consumption, on average, by 75% (compared to the base line architcture) without sacrificing performance**

I. Introduction

On-chip caches can have a huge impact on the processor speed. Caches are faster than the main memory, and consume less power per access than the main memory. A well-designed cache results in a fast and energy efficient processor.

As the size of the chip increases, and the number of transistors on the chip increases, the cache size also increases, for the DEC 21164 processor, 43% of the total energy consumed in the chip is consumed by the cache [3]. Therefore, reducing energy consumption in caches is a priority in the design of embedded processors. In the rest of this section, we briefly review some of the previous attempts to reduce instruction cache energy in embedded processors.

In [8] the authors showed how to use a unified cache to reduce the total area of the cache by 20-30% and maintain the same hit rate as a split cache. Albonesi in [1] proposed the selective way cache. In the selective way cache, preferred ways (a subset of all the ways) are accessed first; in case of a miss, the rest of the ways are accessed. The savings in energy (by not accessing all the ways) is accomplished at the expense of increasing the access time (2 cycles to access the cache in the case of misprediction). Zhang et al [13] proposed a cache where by setting a configuration register they can reconfigure the cache size, the cache associativity, and the cache line size. By fine-tuning the cache parameters to the application, they achieved a power saving of up to 40%.

Way prediction was used in [14] to reduce cache energy. In order not to sacrifice the cache speed, they used a 2-level prediction scheme. First, they decide if they use way prediction or not; if not then all the ways in a set associative cache are accessed. However, if the decision is to use way prediction, the predicted way is accessed first, in case of a miss, the rest of the ways are accessed. A non-uniform cache was introduced in [7]. In this design, the cache has different values for associativity. The optimal value for the number of ways is determined for each application and used for this application. They also proposed some techniques in order to minimize the access to redundant cache way and cache tags to minimize energy consumption.

HotSpot cache was introduced in [11] where a small filter cache was used to store loops that are executed more than a specific threshold. The loops are detected by using the Branch Target Buffer (or BTB) and is promoted to the HotSpot cache when they reach their threshold values. Their design resulted in reducing the energy consumption of the cache. Jouppi in [6] showed how to use a small fully associative cache and prefetching to improve the performance of a direct-mapped cache without paying the price of a fully associative cache. Zhang et al introduced the way-halting cache in [12] where they used some bits from the tag in order to choose which way to access in a multi-way (set associative) cache.

In this paper, we introduce a new cache architecture that has a slightly better average cache access time than many existing architectures and consumes much less energy compared to the existing architectures. We use MediaBench and Mibench benchmark to compare our results with the standard cache without any line buffers, the Filter cache, the HotSpot cache, the Way-halting cache and the single predictive line buffer.

The organization of this paper is as follows. In Section II, we discuss the motivation behind our architecture. In Section III we propose and explain our architecture. Section IV gives details of our prediction and line placement algorithm. Section V presents the simulation setup and compares our architectures with the HotSpot cache, Filter cache and single line buffer cache. Section VI concludes the paper.

II. Motivation

The overall occurrences of branches are application dependent for example, multimedia applications tends to have more structured conditional branches (structured in the sense of relatively small loops that are executed a large number of times), compared with others applications (SPEC2000 applications).

But no matter what type of workload, conditional branches represents a significant fraction of total instructions executed by a program.

A loop block, a sequence of instructions whose iterations is controlled by conditional instruction, may contain a minimum of 2 instructions and can be up to a very large number of instructions or more. In the Mediabench/Mibench suite, Over 75% of such control instructions jump maximum of 16 instructions, while almost 50% of the control instructions jump no more then 6 instructions. For such applications, loop block can be captured within 2-3 line buffer. Fig-1 and Fig-2 shows average branch target distribution for MediaBench and SPEC2000 applications respectively. From the figures we can observe that the MediaBench application concentration is between 5 to 15 instructions. However, by careful analysis of the programs in the MediaBench and SPEC2000 we found the following

- Branches, on average, don't jump very far. (Fig. 2 and 1)
- On average, there is branch instruction after every 7 instruction [5].
- Each program spend most of its time within certain blocks [11]
- More than 75% of the loops in the MediaBench suite include 16 or less instructions.
- Almost 95% of the loops in the MediaBench suite contain 30 or less instructions
- Almost 90% of the loops in SPEC2000 contains 30 or less instructions.

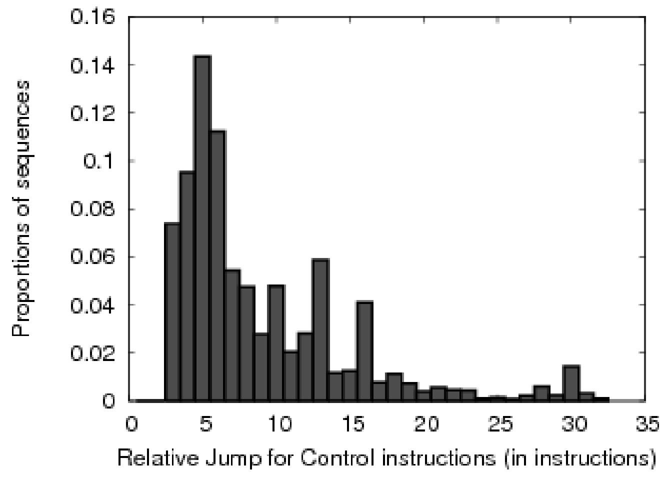


Fig. 1. Distribution of Conditional Instruction Relative Targets (number of instruction between current and target address) for MediaBench/Mibench Benchmark's Applications

In [2] we showed how to use a single line buffer in order to reduce energy consumption in a direct-mapped cache. While 16-instruction loops cannot be be captured using a single line buffer, they could be captured if 4-8 line buffers are used with a good cache organization to guarantee that the instructions in the loops are mapped to the entire set of line buffers instead of replacing each other in a small number of line buffers. Increasing the line size is not the solution since it affects the temporal locality and may reduce the hit ratio.

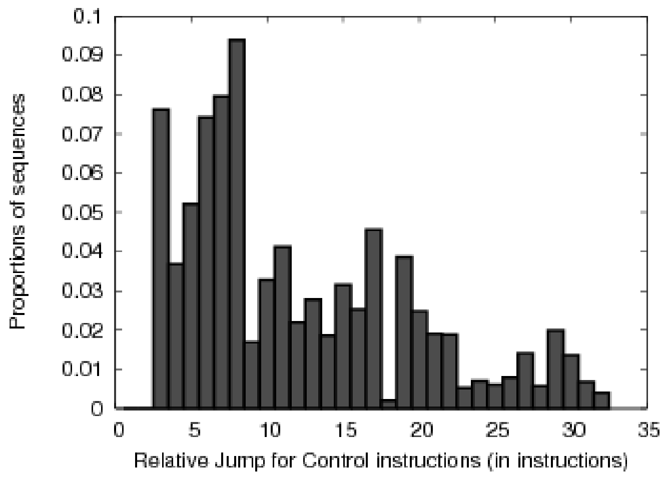


Fig. 2. Distribution of Conditional Instruction Relative Targets (number of instruction between current and target address) for SPEC2000 Benchmark's Applications

III. Proposed Architecture

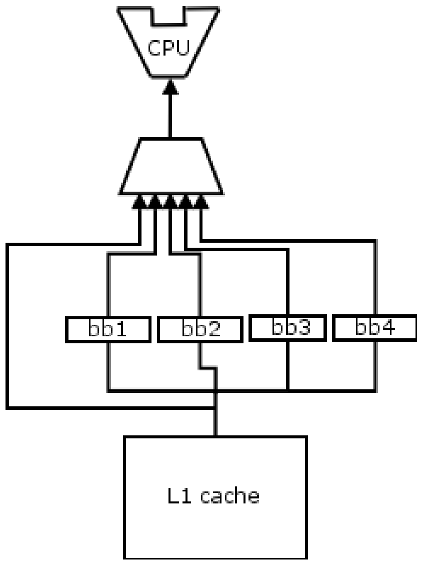


Fig. 3. Cache with Multiple Line Buffers

Our single predictive line buffer (proposed in [2]) doesn't have the ability to capture the whole loop block i.e. cannot take advantage of temporal locality, hence require accessing lower level (level-1) cache more often. To fully utilize the temporal locality in a program, we now extend our single predictive line buffer scheme by adding multiple line buffers between the CPU and L1 cache. We also propose a new prediction scheme to predict which line buffer to access, or the L1 cache if we predict that the line is not tin the line buffers. During the fetch cycle only one of the line buffers is accessed. In case of a miss in the line buffer, the instruction will be fetched from L1 and the line containing the fetched instruction is placed in one of the line buffers. Figure 3 shows a schematic of the proposed architecture. with 4 line buffers, labeled as bb1, bb2, bb3 and bb4, and the L1 cache. The optimal number of line buffers depends on the application. In our simulations, we found that having anywhere between 4 to 8 line buffers achieves the best results for most of the applications in MediaBench and Mibench benchmark. We present detail analysis relating to number of line buffers in Section V-B.

Our scheme dynamically selects either one of the line buffers or the L1 cache for fetching instructions. We assume the existence of BTB (branch target buffer), which is common in many modern embedded processors. The added hardware for the selection mechanism is minimum – it involves some extra registers (called *tag-bit registers*) for storing a part of the tag for the cache lines in the line buffers, and the hardware implementation of the prediction mechanism.

A. Tag-Bit Registers

Our goal is to spread the loop(s) between the different line buffers and to keep the loop(s) that are being currently executed in the line buffers. If we are successful in doing that, the tags of the data in the line buffers are sequential, and they differ only in the low-order bits. This observation can be effectively used to predict the line buffer containing the instruction to be fetched. The main idea of our proposed architecture is to cache a few of the low-order bits of tags in special registers called tag-bit registers. The i low order bits of the tags in each line in line buffer are kept in tag-bit registers.

Our algorithm compares the contents of the tag-bit register with the corresponding bits of the instruction address. If one matches, this is the predicted line buffer and we access it to get the instruction. This requires much less energy than accessing the L1 cache. If our prediction is incorrect, then we have to go to the L1 cache to access the required instruction. Figure 4 shows the organization of the tag-bit register.

As we mentioned before, 75% of the loops in the Media-Bench suite include 16 or less instructions, while 95% of the loops in the same suite contains 30 or less instructions. Having multiple line buffers increases the probability of including the entire loop (or multiple loops) in the line buffers. Of course that could be achieved with a good organization of the line buffers such that the instructions in the loop are mapped to the entire set of line buffers instead of being mapped to few line buffers replacing each other. The instructions in the loop are sequential, which means they differ in their low order tag bits. If we can map them to different line buffers, the tag-bit register can be effectively used to predict the line buffer containing the instruction to be fetched.

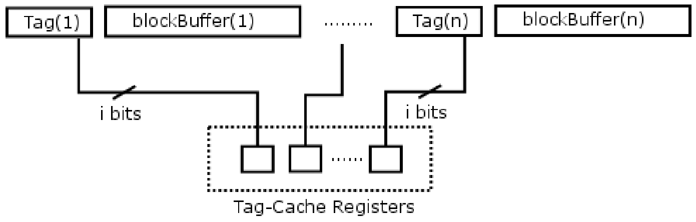


Fig. 4. Tag-Bit Registers

In the next section, we present the prediction mechanism and the placement mechanism used in order to increase the probability of placing the long loops successfully in the line buffers.

IV. Prediction Scheme

In order to predict effectively between line buffers and the L1 cache we need to keep some state variables. These state variables are *fetch_mode* which could be either L1 cache, or line buffer. If *fetch_mode* points to the line buffer, then *curr_bb* is a pointer to the predicted line buffer that holds the instruction to be fetched. Finally, *TAG_bits* is an array which holds the low order *i* bits of every tag in the line buffers. This may be a physical register, or could be just the i bits in the TAG stored in every line buffer. Figure 5 shows a state diagram for the prediction algorithm.

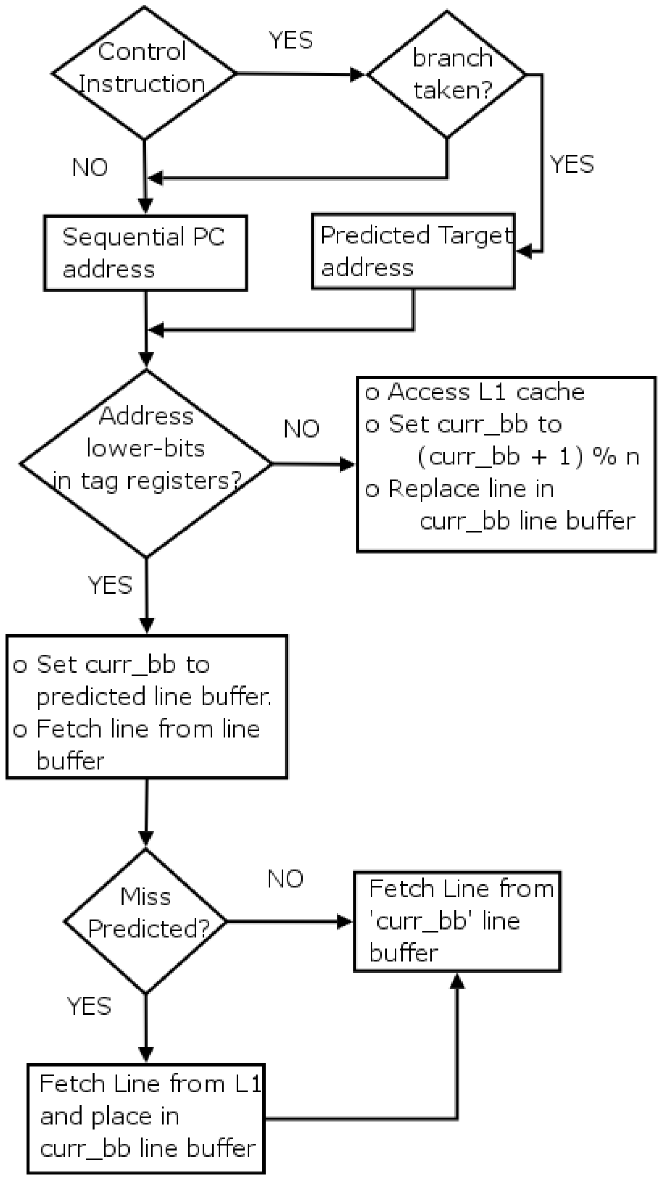

Control Instruction
YES
branch taken?
NO
YES
Sequential PC address
Predicted Target address
Address lower-bits in tag registers?
NO
o Access L1 cache
o Set curr_bb to (curr_bb + 1) % n
o Replace line in curr_bb line buffer
YES
o Set curr_bb to predicted line buffer.
o Fetch line from line buffer
Miss Predicted?
NO
Fetch Line from 'curr_bb' line buffer
YES
Fetch Line from L1 and place in curr_bb line buffer

Fig. 5. Flow Diagram for the Prediction Scheme

The main idea of the prediction algorithm is as follows. Once the instruction is fetched, the program counter (PC) is checked against the BTB to see if that instruction is a branch and predicted taken or not. If the instruction is a branch and is predicted taken, the target address is loaded from the BTB into the PC, otherwise the next sequential address is loaded in the PC (PC is incremented according to the instruction length). The low order bits of the TAG part of the address is checked against the *TAG_bits* array. If there is a match, then the *fetch_mode* is set to that particular line buffer. Otherwise it is set to L1 cache. If there is no match, and the prediction is set to L1 cache, then the instruction will be then fetched from the L1 cache and will be stored in line buffer *curr_bb*+1.

If there is a match, then the instruction is fetched from the predicted line buffer. The fetched instruction may or may not be the required instruction (we checked only the low order i bits). If our prediction is correct, then the instruction is sent to the CPU. If we mispredicted, then the instruction is accessed from the cache, the line contains that word is sent to the *curr_bb*.

V. Experimental Results

In this section, our proposed scheme, multiple predictive line buffers will be compared against various other scheme for both performance and energy. We'll first show our experimental results for the effect of the number of line buffers for various applications in the Mediabench/Mibench benchmarks. Then, we show the results of the effect of the number of bits in the tag-bit registers on the miss ratio. Then, we evaluate the effectiveness of multiple predictive line buffer scheme by comparing it with Filter cache, HotSpot cache, single predictive line buffer and Way-halting cache. We also present our results on the off-chip memory access for our proposed cache architecture and compare it with the other architectures. Finally, we present a small note on the hardware cost of our proposed architecture.

A. Experimental Setup

We use SimpleScalar toolset [9] and CACTI 3.2 [10] to conduct our experiments. We have modified SimpleScalar to simulate Filter Caches, HotSpot caches, Predictive Line buffer and Way-Halting cache. Our baseline architecture uses a 16KB direct-mapped cache or 16KB 4-way set-associative cache. Our Line buffer is 32 bytes. We have used a 512 bytes, direct-mapped L0 cache for Filter cache and HotSpot cache. The BTB is 4-way set-associative with 512 sets. We have used a 2-level branch predictor in our simulation. We evaluated energy consumption using $0.35\mu m$ process technology. For HotSpot cache, we used a value of 16 as candidate threshold as was suggested in [11]. As our proposed scheme is targeted toward embedded microprocessors, we have used multimedia benchmarks, MediaBench and Mibench but our scheme will yield similar results for other types of workload. Each applications was limited to 500 million instructions using the data set included with the benchmark suites. We choose sets of encoder/decoder from different media types such as data, voice, speech, image, video and communication. (See Table-I). Energy per cache access, as is obtained from CACTI is shown in Table-II

B. Optimal number of line buffers

The ideal number of Line buffer depends on each applications behavior. Our experiments shows that anywhere between 4 and 8 lines buffers are optimal for most applications. Fig-6 shows the averages energy reduction when using multiple line buffer (2 to 8) for the MediaBench and Mibench benchmark applications using width of 4 bits for tag-bit registers.

TABLE I
Benchmark Applications Summary

Application	Type	Benchmark
crc32/fft	Communication	Mibench
epic	Data	MediaBench
adpcm/g721/gsm	Voice/Speech	MediaBench
jpeg	Image	MediaBench
lame	Mp3	Mibench
mpeg2	Video	MediaBench

TABLE II
Energy per Access for Various Cache Configurations

Cache	Energy
512 L0 cache	0.69nJ
line buffer	0.12nJ
16KB direct-map	1.63nJ
16KB 4-way set-assoc	2.49nJ

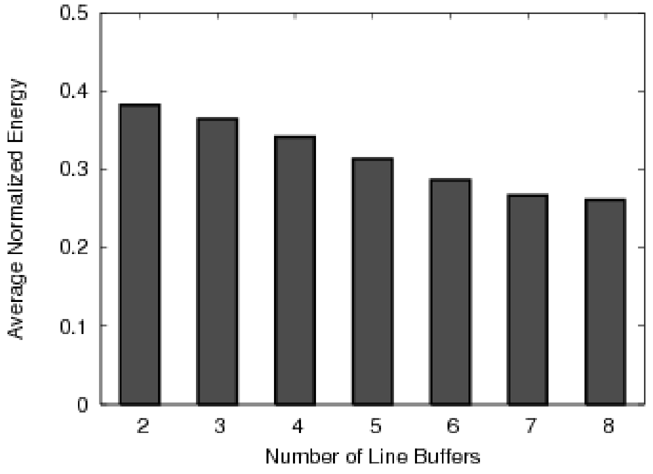
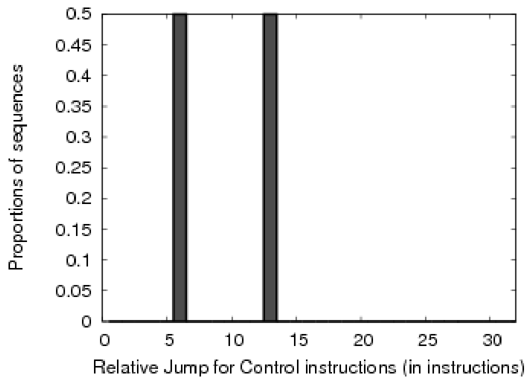
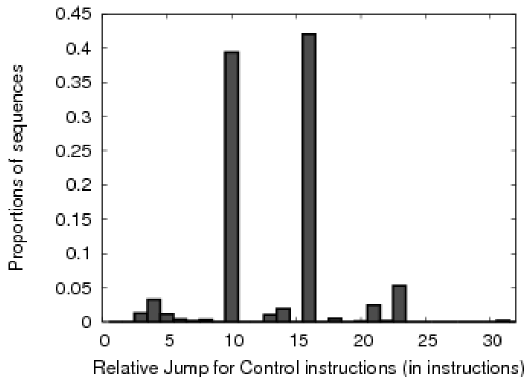


Fig. 6. Normalized Average Energy Reduction for Mediabench/Mibench Applications

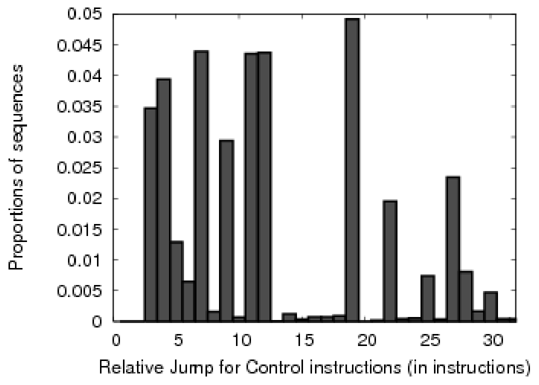
By analyzing average normalized energy for various applications of benchmarks (See Figures 7 for some of the applications from benchmark), we observed that number of line buffer to use depends upon the conditional branches relatives target address. Knowing, on average, how far these instructions jumps to (i.e. size of loop block), we can relate them to how many line buffers is required for that particular application. For instance, for communication application crc32, using 6 line buffer is optimal and adding 7 or 8 line buffer does not improve the energy consumption (See Fig. 7(a)). This is because for such application, almost none of loop block are of greater then 14 instruction and can be easily captured using maximum of 6 line buffers. Similar observation can be made for other applications in MediaBench and MiBench applications.



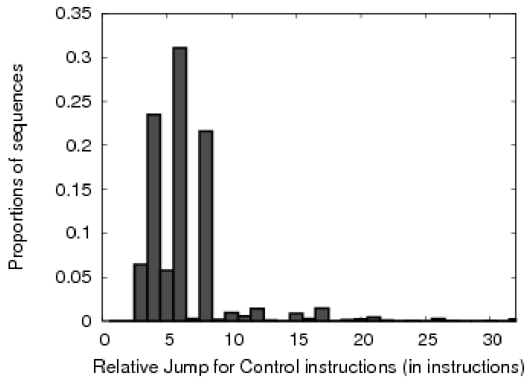
(a) crc32



(b) epic



(c) jpeg2-encode



(d) mpeg2-decode

Fig. 7. Applications Branch Target Address distribution

C. Tag-bit Registers and Misprediction

The key component for effectiveness of multiple predictive line buffer is tag-bit registers. We experimented with numerous width of low-order tag bits. A near-ideal tag bit size is that which can find the right line buffer, hence less miss-prediction. By storing more bits in registers will give more accurate prediction but will also increase overhead as registers data width will increase.

Table III and IV show the result of using between 2 and 6 bits with 4 line buffer and 4-8 bits with 8 line buffer for direct map cache. From the average of these two tables we can see when using 2 tag bit with 4 line buffer, the miss prediction can be as high as 22.43%. As we stated in Section II, that almost 90% of loops contains 30 or less instructions, using 2 bits we can only accurately distinguish between maximum of 4 instructions in any loop block. Therefore using more bits will help us predicts more efficiently. From the tables we can see that using 4 or 5 bits can gives satisfactory results for most of applications when using 4 or 8 line buffers. For all our experiment that follows, we have used 5 bits as data width for our tag-bit registers

TABLE III
MISPREDICTION RATIO (4 LINE BUFFER, 2-6 BITS FOR TAG-BIT REGISTER) USING DIRECT-MAP L1 CACHE

BenchMark	2-tag	3-tag	4-tag	5-tag	6-tag
apdcm-decode	26.59	0.18	0.14	0.00	0.00
apdcm-encode	25.88	3.86	0.06	0.05	0.01
crc32	20.02	0.02	0.01	0.00	0.00
epic	18.24	1.05	0.01	0.00	0.00
fft	27.70	8.63	3.37	1.47	1.20
fft-inv	27.70	8.63	3.37	1.47	1.20
g721-decode	19.18	3.97	1.88	0.90	0.47
g721-encode	18.47	2.66	1.15	0.40	0.17
gsm-decode	30.88	3.27	1.46	0.04	0.02
gsm-encode	26.56	0.57	0.17	0.10	0.00
jpeg2-decode	26.11	1.91	0.54	0.04	0.01
jpeg2-encode	19.64	3.90	0.45	0.08	0.03
lame	19.22	3.07	1.20	0.15	0.06
mpeg2-decode	15.25	4.32	2.34	2.24	2.20
mpeg2-encode	22.05	4.23	1.32	0.56	0.01
unepic	15.72	2.53	0.38	0.02	0.01
Average	22.45	3.30	1.12	0.47	0.34

D. Energy

In this section we compare the energy consumption of multiple predictive line buffer with conventional filter cache, HotSpot Cache, way-halting cache, and single Predictive line buffer. Fig. 8 and Fig. 9 show the normalized energy consumption of the above mentioned 5 different cache architectures. These results are normalized to a baseline cache. We used two different baseline caches, a direct mapped cache Fig. 8 and a 4-way set associative cache Fig. 9. Note that by definition the way-halting cache requires a set associative cache and could not be compared with the direct mapped L1 cache.

From these two figures, we can see that using multiple line buffer does have a huge effect on the energy consumption. For some applications such as crc32, epic, and jpeg-encodes

TABLE IV

MISPREDICTION RATIO (8 LINE BUFFER, 4-8 BITS FOR TAG-BIT REGISTER), USING DIRECT-MAP L1 CACHE

BenchMark	4-tag	5-tag	6-tag	7-tag	8-tag
apdcm-decode	0.16	0.01	0.00	0.00	0.00
apdcm-encode	0.07	0.06	0.00	0.00	0.00
crc32	0.01	0.00	0.00	0.00	0.00
epic	0.02	0.01	0.00	0.00	0.00
fft	7.20	3.85	2.70	1.31	0.71
fft-inv	7.20	3.85	2.70	1.31	0.71
g721-decode	4.19	1.35	0.76	0.25	0.24
g721-encode	2.71	1.05	0.55	0.08	0.07
gsm-decode	2.38	0.61	0.58	0.55	0.00
gsm-encode	1.72	1.15	0.05	0.04	0.00
jpeg2-decode	3.00	0.17	0.08	0.02	0.00
jpeg2-encode	2.28	1.40	0.34	0.28	0.28
lame	2.49	0.67	0.31	0.17	0.02
mpeg2-decode	2.84	2.57	2.36	0.04	0.03
mpeg2-encode	3.76	2.03	0.04	0.00	0.00
unepic	1.23	0.08	0.05	0.01	0.00
Average	2.58	1.18	0.66	0.25	0.13

using 8 line buffer significantly reduce energy consumption compared to are used, over 4 line buffer.

The reason is because for these applications branch target distribution. Fig. 7(a), 7(b), 7(c) and 7(d) show that most of the loops could not be included in 4 line buffers. Therefore using 8 line buffer significantly reduces energy consumption for these application compared to others in benchmark. Using 8-multiple line buffer reduces normalized energy consumption by up to 74%, compared to the baseline cache, and 47% comapred to with HotSpot Cache.

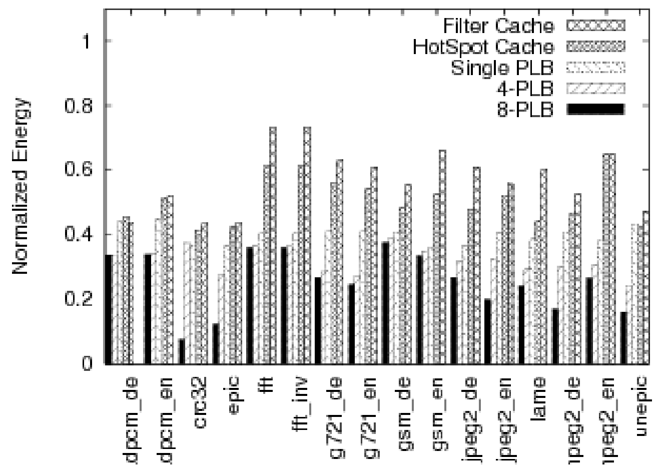


Fig. 8. Normalized energy reduction using direct-map L1 cache

Table V shows the average normalized energy consumption of various scheme (the average is taken over all the programs in Mediabench and Mibench suites). From the table, we can observe that using 8 multiple line buffer, on the average, significantly lower energy consumption compared with others.

E. Delay

In this section we show that multiple predictive line buffer scheme doesn't sacrifice performance for the sake of reducing

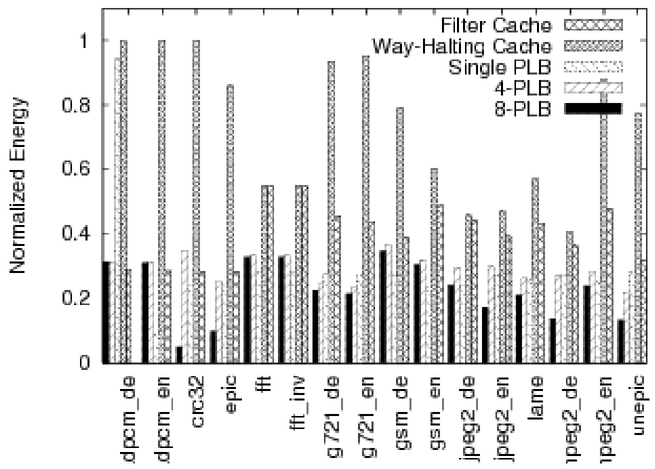


Fig. 9. Normalized energy using 4-way set-associative L1 cache

TABLE V

VARIOUS SCHEMES AVERAGE NORMALIZED ENERGY USING DIRECT-MAP AND 4-WAY SET-ASSOCIATIVE CACHE

Scheme	Direct Mapped	Set-associative
4 PLB	0.32	0.29
8 PLB	0.26	0.22
Filter Cache	0.57	0.46
HotSpot Cache	0.53	0.41
Single PLB	0.40	0.36
Way-Halting Cache	N/A	0.61

energy consumption. We'll compare normalized delay for multiple predictive line buffer (using 4 and 8 line buffer), with HotSpot Cache and single predictive line buffer, filter cache, and way-halting cache with both direct-map L1 cache and 4-way set associative cache.

The normalized delay using conventional direct-map L1 cache is shown in Fig. 10, and using 4-way set associative cache in Fig. **??**. Using 4 predictive line buffer performs as good as single predictive line buffer. Using 8 multiple line buffer have higher delay compared to single predictive line buffer but still is significantly better then HotSpot Cache. As we mentioned in Section V-C, if using 6 tag width, the delay for 8 line buffer can be improved. Using 5 bit width for tag-bit registers, the performance overhead is still very minimal. Table VI shows the average normalized delay for HotSpot Cache, single predictive line buffer, Filter cache, way-halting cache, and 4 and 8 line buffers, when using direct-mapped and 4-way set associative as the L1 cache. The results clearly show that our scheme, on the average, achieves near-ideal delay for various applications (The average is taken over all the programs in Mediabench/Mibench suites)..

F. $Energy * Delay$

The product of the energy and the delay is considered to be a good measure of performance since it takes into consideration both the delay and the energy consumption of the cache. If a scheme can improve the delay, on the expense

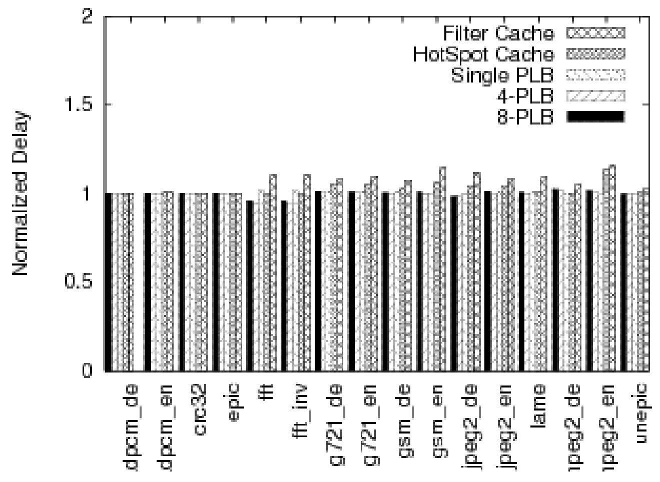


Fig. 10. Normalized Delay using direct-map L1 cache

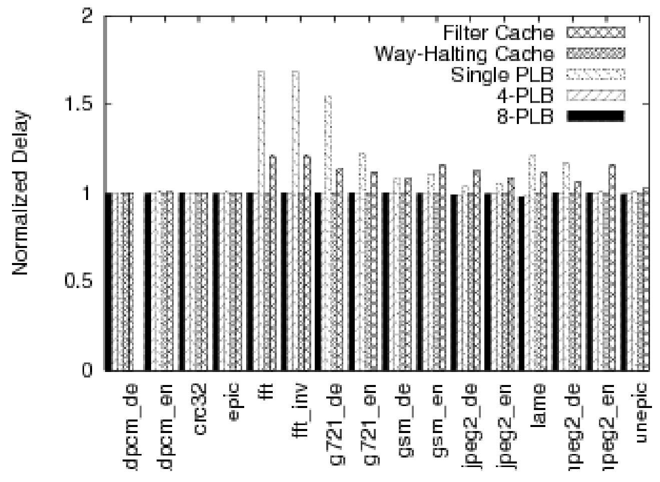


Fig. 11. Normalized Delay using 4-way set-associative cache

of energy consumption, or vise versa, that will show in the $Energy * Delay$. Fig. 12 and Fig. 13 show $Energy * Delay$ product for HotSpot cache, filter cache, way-halting cache, and single and multiple line buffer cache scheme, when using Direct-mapped and 4-way Set-associative L1 cache respectively. Our proposed scheme outperforms all the other schemes.

G. Off-chip memory access

Accessing off-chip memory is expensive, both in terms of energy consumption and delay. Accessing 512KB 4-way set-associative off-chip cache is almost 6 times more expensive than accessing the same cache but on-chip. Direct-mapped cache although has a fast access time, but can suffers from thrashing problem. Thrashing occurs when two memory lines maps to same line in the cache. Thrashing can cause performance issue as most of time is spend in moving data between memory and caches. Thrashing can be avoided if the loop-block can be captured in upper level cache hence avoiding conflicts. Our proposed scheme did not increase off-chip access. For most of the applications in Mediabench/Mibench suites, our proposed architecture slightly better than the other architectures, and for some applications it did significantly better than the other architectures.

TABLE VI
VARIOUS SCHEMES AVERAGE NORMALIZED DELAY USING DIRECT-MAP AND 4-WAY SET-ASSOCIATIVE CACHE

Scheme	Direct Mapped	Set-associative
4 PLB	0.993	0.997
8 PLB	0.999	0.997
Filter Cache	1.070	1.091
HotSpot Cache	1.027	N/A
Single PLB	1.004	1.007
Way-Halting Cache	N/A	1.000

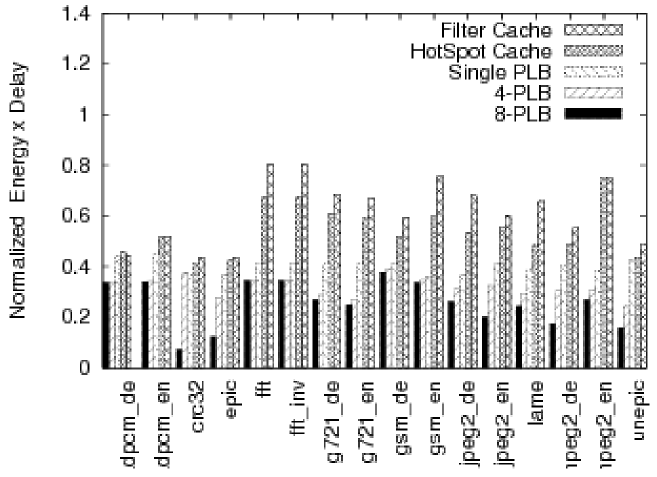


Fig. 12. Normalized $Energy * Delay$ using direct-map L1 cache

H. Hardware Cost

A detailed hardware cost of our proposed scheme is beyond the scope of this paper. However, we present here a very rough estimation of the hardware required to implement our architecture.

Our architecture does not use filter or L0 cache, however it adds 4-8 line buffers. the Filter of L0 cache is usually more than 4-8 lines, thus on the cache level it requires less hardware compared to filter of HotSpot cache. Compare to the Way-Halting cache, it requires 4-8 more line buffers. We also need hardware to implement the prediction mechanism. We need pointers to *curr_bb*, and the prediction flag. Also few multiplexers and flip-flops can be used for the control of the prediction mechanism. In our opinion, compared with the energy saving, that is a very good price to pay.

VI. CONCLUSION

In this paper, we extended our single predictive-line buffer scheme (proposed in [2]) in order to capture long loops in the line buffers. We presented a cache architecture that utilizes 4-8 line buffers, the BTB and a simple prediction mechanism to reduce the energy consumption in the instruction cache. The prediction mechanism we proposed let us access only one

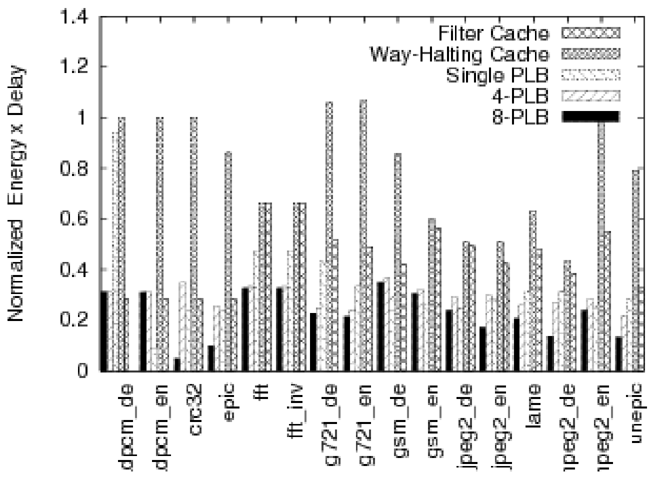


Fig. 13. Normalized $Energy * Delay$ using 4-way set-associative L1 cache

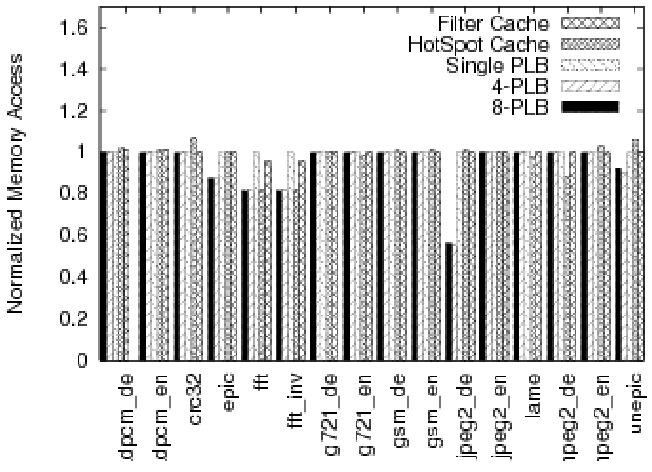


Fig. 14. Normalized Off-Chip Memory Access

single line buffer, or the L1 cache in any cache access. Our Simulation results show that on average, our scheme reduces instruction cache energy up to 75% compared with a baseline cache, without sacrificing performance.

References

[1] D. Albonesi, "Selective cache ways: on-demand cache resource allocation" *Proc. of the* 32^{nd} *ACM/IEEE International Symposium on Microarchitecture*, pp 248-259, Nov. 1999.

[2] K. Ali, M. Aboelaze and S. Datta, "Predictive line buffer: A fast energy efficient cache architecture" Submitted to IEEE Southeast-Con 2006 in January 2006.

[3] J.F. Edmondson, "Internal organization of the Alpha 21164, a 300-MHz 64 bit quad-issue CMOS RISC microprocessor," *Digital Technology J.*, Vol. 7, No. 1, pp 119-135, 1995.

[4] A. Hasegawa, I. Kawasaki, S. Yoshioka, S. Kawasaki, and P. Biswas, "SH3: High code density, low power", *IEEE Micro*, Vol. 15, No. 6, pp 11-19, Dec. 1995.

[5] J. L. Hennessy, and D. A. Patterson, "Computer Arcjitecture: A Quantitative Approach" Morgan Kaufmann Publishing, 2003.

[6] N.P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers" *the* 17^{th} *Annual International Symposium on Computer Architecture ISCA*, pp 364-373, May 1990.

[7] T. Ishihara and F. Fallah, "A non-uniform cache architecture for low power system design", *Proceedings of the 2005 International Symposium on Low Power Electronics and Design, ISLPED '05*, pp 363-368, Aug. 2005.

[8] H. Mizuno and K. Ishibashi, "A separated bit-line unified cache: Conciliating small on-chip cache die-area and low miss ratio", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Volume 7, No. 1, pp 139-144, March 1999.

[9] The Simplescalar simulator *www.simplescalar.com* May 2006

[10] Shivakumar, P.; Jouppi, N.; "CACTI 3.0: An integrated cache timing, power, and area model" *Technical Report 2001.2 Compaq Research Lab* 2001

[11] C.-L. Yang and C.-H. Lee, "HotSpot cache: joint temporal and spatial locality exploitation for I-cache energy reduction", *Proc. of the 2004 International Symposium on Low Power Electronics and Design ISPLD'04*, pp 114-119, Aug. 2004.

[12] C. Zhang, F. Vahid, J. Yang and W. Najjar, "A way-halting cache for low-power high-performance systems", *Proc. of the 2004 International Symposium on Low Power Electronics and Design ISPLD'04*. Aug. 2004.

[13] C. Zhang, F. Vahid and W. Najjar, "A Highly Configurable Cache for Low Energy Embedded Systems", *ACM Transactions on Embedded Computing Systems (TECS)*, Vol. 4, No. 2, pp 363-387, May 2005.

[14] Z. Zhu and X. Zhang, "Access mode prediction for low-power cache design", *IEEE Micro*, Vol. 22, No. 2, pp 58-71, March-April 2002.