Published in IET Computers & Digital Techniques
Received on 13th April 2007
Revised on 22nd August 2007
doi: 10.1049/iet-cdt:20070055

ISSN 1751-8601

# Energy efficient i-cache using multiple line buffers with prediction

*K. Ali*[1] *M. Aboelaze*[2] *S. Datta*[2]
[1]*School of Computing, Queens University, Kingston, Ontario, Canada*
[2]*Department of Computer Science and Engineering, York University, Toronto, Ontario, Canada*
*E-mail: aboelaze@cse.yorku.ca*

**Abstract:** Modern microprocessors dedicate a large portion of the chip area to the cache. Decreasing the energy consumption of the microprocessor, which is a very important design goal especially for small, battery powered, devices, depends on decreasing the energy consumption of the memory/cache system in the microprocessor. The authors investigate the energy consumption in caches and present a novel cache architecture for reduced energy instruction caches. Our 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. In the proposed technique, the authors use the multiple line buffers as a continuous small filter cache that can catch most of the cache access but they access only a single line buffer, thus reducing the energy consumption of the cache. They used simulation to evaluate the proposed architecture and to compare it with the HotSpot cache, filter cache and single line buffer cache. Simulation results show that the approach is slightly faster than the above mentioned caches, and it consumes considerably less energy than any of these cache architectures.

## 1 Introduction

In the computer architects' quest for faster, smaller, more powerful and cheaper processors, the size of the processor is increasing and more and more transistors are put on the chip. As the size of the chip increases, the cache size also increases, increasing the total energy in the sytem. Therefore reducing energy consumption in caches is a priority for computer architects.

Reducing the energy consumption in the cache can be achieved in two different ways. At the physical (hardware) level, the physical design of the cache and the voltage requirements can be used to reduce the energy consumption. Reducing leakage power and reducing the voltage swing on the bit lines was proposed in [1]. Whereas in [2] the authors proposed a new register design, Content-charge-aware CRA to reduce the power consumption without added delay. At the architecture level, novel cache architecture, prediction and replacement policies are used to reduce the energy consumption of the cache. In this paper, we concentrate on the architecture level.

There have been many attempts to improve the cache performance (average memory access time) as well as decreasing the energy consumption of the cache. In this section, we briefly discuss some of the recent attempts of reducing energy consumption and the average memory access time at the architecture level. In [3], the authors showed how to reduce the total area of the cache by 20–30% by using a unified cache instead of a split cache, at the same time they did not sacrifice performance by maintaining the same hit rate as a split cache. Albonesi in [4] proposed the selective way cache. In set associative caches, a lot of energy is consumed by accessing all the ways of the set associative cache. Since the requested item, if found, is in only one of the ways, the energy consumed in accessing the other (non-successful) ways is wasted. In selective ways cache, preferred ways (a subset of all the ways) are accessed first; in case of a miss, the rest of the ways are accessed, thus requiring a second access cycle in case of miss in the preferred ways. The saving in energy (by not accessing all the ways) is accomplished at the expense of increasing the access time. Zhang *et al.* [5] proposed fine-tuning the cache architecture to the running application. The tuning is done at run time. They 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-tunning the cache parameters to the application, they achieved a power saving of up to 40%.

The authors in [6] showed how to tune the filter cache to the needs of a particular application in order to save energy. Way prediction was used in [7] to reduce cache energy. In order not to sacrifice the cache speed, they used a two-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 [8]. 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 minimise the access to redundant cache ways and cache tags to minimise energy consumption.

HotSpot cache was introduced in [9]. The main idea of the HotSpot cache is to use a small filter cache (HotSpot cache) to store frequently executed loops. The loops that are executed more than a specific number of times (threshold) are marked as hot blocks and are moved to the HotSpot cache. The loops are detected by modifying the branch target buffer (BTB) by adding a counter to each entry in order to count the number of times that entry is accessed (which is the same as the number of times the loop is executed). These loops are promoted to the HotSpot cache when they reach their threshold values. Their design resulted in reducing the energy consumption of the cache. Jouppi in [10] 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 [11] where they used some bits from the tag in order to choose which way to access in a multiway (set associative) cache.

In [12], the authors proposed a variable sized block cache. Their scheme depends on identifying the basic blocks (block tail is a backward branch, block head is the target of the backward branch) and they mapped them to a variable size cache block. They successfully addressed the problem of the instruction overlap among traces that was present in the trace cache [13]. In [14], the authors investigated the energy dissipation in the bit array and the memory peripheral interface circuits. Taking these parameters into consideration, they optimized the performance and power dissipation in the cache. Different techniques for reducing static energy consumption in multiprocessors caches were compared in [15]. Whereas Benini *et al.* [16] proposed code compression techniques, where the instructions are accessed from the cache in a compressed form.

In [17], the authors proposed a technique for energy saving in caches by using a dynamic zero-sensitivity scheme. They prevented the bitlines from discharging when reading zeros, thus reducing the energy consumption of reads. In [18], the authors dealt with energy consumption in the memory. They considered the operating system level and discussed techniques to reduce the overall energy consumption in the memory. Their technique, power aware buffer cache management, resulted in energy saving up to 63%. However, as we mentioned, they considered the overall memory system not only the cache. Quershi and Patt in [19] introduced a runtime mechanism for partitioning the cache among multiple concurrently executing applications. They showed that their techniques reduces the energy consumption by an average of 11% over LRU-based cache partitioning on a dual-core system using multiprogrammed workload. Both hardware and software-controlled power management techniques were used in [20]. These techniques resulted in both higher system performance and low average power consumption.

In this paper, we introduce a new cache architecture that has a slightly better average cache access time than many existing architectures and consumes considerably less energy compared with the existing architectures. We use applications from Mediabench, Mibench benchmarks, as well as SPEC2000 to compare our results with the standard cache without any line buffers, the filter cache, the HotSpot cache and the single line buffer cache.

The organisation of this paper is as follows. In Section 2, we discuss the motivation behind our architecture. In Section 3, we propose and explain our architecture. Section 4 gives details of our prediction and line placement algorithm. Section 5 presents the simulation setup and compares our proposed architecture with the HotSpot cache, filter cache and single line buffer cache for embedded applications.

## 2 Motivation

In [21], we showed how to use a single line buffer with prediction in order to reduce energy consumption in a direct-mapped cache (for the remaining of this paper, we assume that line buffer is the same size as the cache line). However, by careful analysis of the programs in the Mediabench and Mibench we found the following (Fig. 1 shows the loop length distribution for Mediabench and Mibench suites).

1. More than 75% of the loops in the applications include 16 or less instructions.

2. Almost 95% of the loops in the applications contain 30 or less instructions

3. Almost 90% of the loops in SPEC2000 contains 30 or less instructions.

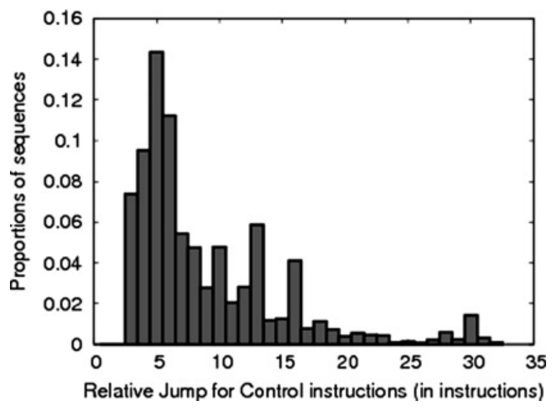

**Figure 1** *Distribution of conditional instruction relative targets for Mibench/Mediabench applications*

Although many loops cannot be captured using a single line buffer, they could be captured if 4–8 line buffers are used with a good cache organisation 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 shows the loop length distribution for SPEC2000 benchmark. We can see that the distribution is more flat for SPEC2000 with higher percentage of loops with >20 bytes. That makes it more difficult to capture loops in SPEC2000 compared with Mediabench and Mibench. As we will see in the experimental result section, our proposed architecture works better for embedded application than for general CPU intensive applications.

In this paper, we propose adding more line buffers in order to capture these loops. We also propose a new prediction and mapping mechanism in order to fully utilise the line buffers. Our simulation results show that we have succeeded in reducing the energy consumption of the cache subsystem, without sacrificing the average cache access time.

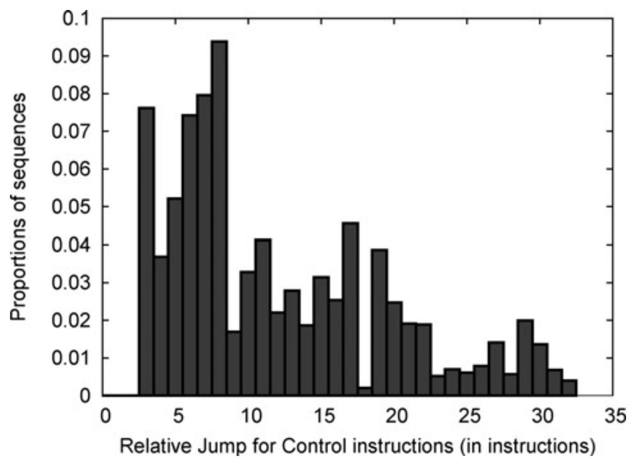

**Figure 2** *Distribution of conditional instruction relative targets for SPEC2000*

## 3 Proposed architecture

In [21], we proposed a single line buffer cache architecture with prediction in order to reduce the energy consumption of the cache. Our proposed architecture consists of the regular L1 cache, with a single line buffer that can hold a cache line. We used prediction in order to steer the cache access to either the L1 cache or the line buffer, thus saving the energy required to access the cache when the required data are in the line buffer. Considering that the line buffer requires less energy compared with the L1 cache, the protocol resulted in considerable energy saving.

However, one of the problems we faced in the previous work is that the line buffer is not big enough to hold most of the loops we encountered in Mediabench and Mibench. Increasing the line size or using a filter cache that can hold multiple cache lines result in reducing the miss ratio, but requires more energy in accessing the filter cache or a large line buffer compared with the energy required to access a single line buffer with the size of a cache line. Our objective is to increase the size of the line buffer(s) in order to avoid L1 cache access, and at the same time maintain the same energy consumption as a single line buffer.

In order to fully utilise the temporal locality in a program and the line buffers, we now extend our single predictive line buffer scheme [21] by adding multiple line buffers between the CPU and the L1 cache. We also propose a new prediction and placement algorithms. Our goal is to store the loops in the line buffers and to predict the line buffer containing the next memory reference. 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 the L1 cache and the line containing the fetched instruction is placed in one of the line buffers. Fig. 3 shows a schematic of the proposed architecture with four line buffers, labelled as lb0, . . . , lb3 and the L1 cache. The optimal number of line buffers depends on the application.

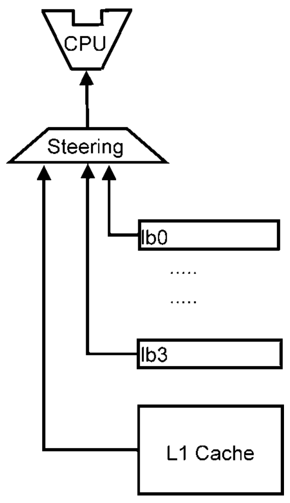

**Figure 3** *Multiple line buffers cache architecture*

In our simulations, we found that having 4–6 line buffers achieves the best results for most of the programs in Mibench, Mediabench suites and SPEC2000.

Our scheme dynamically selects between one of the line buffers and the L1 cache for instructions fetching. We assume the existence of BTB, which is common in many modern processors. The BTB is a small fast buffer that holds the addresses of the branch instructions and the target addresses for them. The BTB is used in order to minimise stall cycles because of control instructions. In this paper, we use the BTB for three things. First, it tells us if the instruction is a control instruction or not, if it is a control instruction, is it taken or not, and if it is taken, what is the target address for that control instruction. Note that our proposed scheme does not add any extra delay cycles for cache access, since predicting instruction $i$ overlaps with the fetching of instruction $i - 1$.

## 4 Prediction scheme for multiple line buffers

In order to predict between line buffers and the L1 cache we need to keep some state information. The state information we have to keep are as follows:

- fetch_mode is an indicator to where to fetch the data from, it could be either L1 cache or line buffer
- curr_lb points to the line buffer number predicted to contain the requested data. Of course this is meaningful only if the fetch_mode points to line buffer.
- pred_way is a flag which is used for steering mechanism between line buffers and L1 cache. pred_way could have one of three different values $-1$, 0 or 1. A value of $-1$ indicates that we predict that the requested instructions will be found in the line buffers, but we are not sure (MAYBE). A value of 0 indicates that the line is not available in any line buffer (NO). Finally, a value of 1 indicates that the prediction we made was correct and for sure the line is in one of the line buffers (YES).

The program counter (PC) is checked against the entries in the BTB at address output time to see if that instruction (if it is a conditional branch) predicted is taken or not. If it is a backward-branch and predicted taken, then we follow the flow chart in Fig. 4*a*. The predicted target's address tag value from the BTB along with the current instruction's address tag value (from PC) is used to find if the predicted target cache line can be found in any of the line buffers. This is simply done by using the difference of the line number (the instruction address less the offset bits, indicated by target_tag) of the target and current instructions.

$$\text{pred\_lb} = \text{curr\_lb} - (\text{addr\_tag} - \text{target\_tag}).$$

Note that, the term (addr_tag − target_tag) means how many line buffer we have to jump backwards in order to go to the target address. If the target cache line can be found, (pred_lb $\geq$ 0); then curr_lb is set to that particular line buffer along with pred_way to $-1$ (MAYBE) and fetch_mode to line buffer. Otherwise, if the target cache line cannot be found, fetch_pred is set to L1 with curr_lb and pred_way both set to 0. The idea here is to start each loop from the first line buffer to capture all block's instructions in the line buffers sequentially. This ease the task of finding the block because either the whole block exists in sequence in line buffers or it doesn't. The first access after pred_way is set to MAYBE, we have to decide if it will be changed to YES or NO. Fig. 4*b* shows the flow chart to do that.

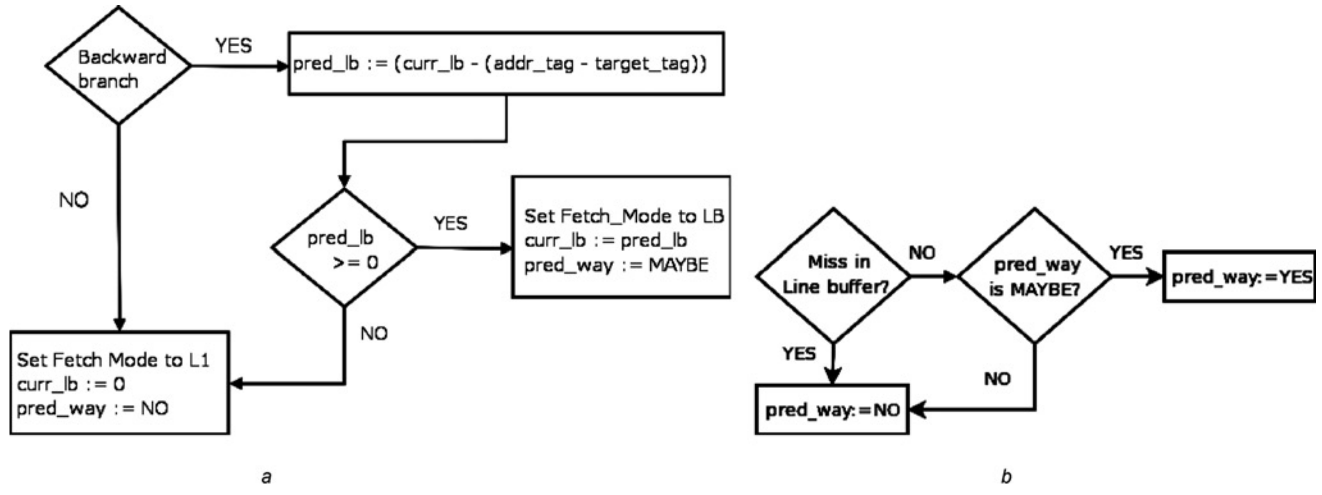

**Figure 4** *Flow chart for the prediction and correction mechanism*

*a* Prediction
*b* Correction

If the last accessed instruction is the last instruction in a cache line, then the next sequential instruction is definitely not in the same line buffer. Fig. 5 shows the flow chart to account for the last word access. If we are in the middle of a block, once we reached the end of a line buffer, then we increment curr_lb in order not to miss on the next access. If the block is being accessed for the first time, then the next line buffer does not contain the next word and we set the access mode to L1.

The steering mechanism comes at a price. We have to implement the prediction (Fig. 4*a*), correction (Fig. 4*b*) and cache access mechanism (Fig. 5) in hardware. For the prediction mechanism, we need a subtracter (adder) in order to calculate the pred_lb. That is a delay that is usually on the critical path of memory access. Moreover, we have to wait for the result of the BTB access to get the target_tag before we start subtraction. The rest of the prediction requires comparison to 0 and setting flags which require minimum delay. The correction mechanism does not require any time since it could be done in parallel with the memory access.

The cache access Fig. 5 mechanism requires only incrementing the line buffer number (in our design it is 2–3 bits long, fast addition could easily be implemented) and some testing logic and setting flags. That definitely will require increasing the cycle, however as we mentioned fast addition and subtraction could be easily implemented. The added hardware is minimal.

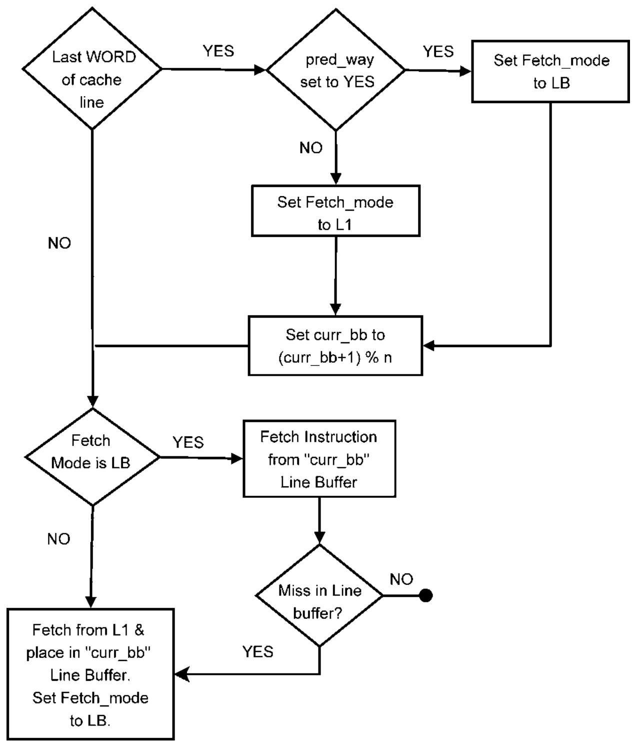
Last WORD of cache line
YES
pred_way set to YES
YES
Set Fetch_mode to LB
NO
Set Fetch_mode to L1
NO
Set curr_bb to (curr_bb+1) % n
Fetch Mode is LB
YES
Fetch Instruction from "curr_bb" Line Buffer
NO
Miss in Line buffer?
NO
YES
Fetch from L1 & place in "curr_bb" Line Buffer. Set Fetch_mode to LB.

**Figure 5** *Flow diagram for the cache access*

## 5 Experimental results

In this section, we will compare our proposed scheme with filter cache, HotSpot cache and a single line buffer cache. Our baseline cache architecture is a direct-mapped cache. We compare both the average memory access time and energy consumed of our proposed architecture with the above mentioned architectures. First, we starts with a set of experiments to decide the best number of line buffers. We simulated the programs in the Mediabench, Mibench and SPEC's CPU2000 benchmarks using up to eight line buffers. We found that the optimal number of line buffer to use depends on the application. From our simulations we concluded that 4–6 line buffer are the best choice over a large number of applications spanning SPEC2000, Mibench and Mediabench benchmarks. Throughout the rest of our simulation we use both four and six line buffers.

The optimal number of line buffers depends on the application. For Mediabench and Mibench, four line buffers capture most of the loops in most of the programs. Some programs such as lame, g721_en, and especially mpeg_de benefits from six line buffers. Adding seven- or eight-line buffers did not result in any improvement. For SPEC2000 benchmark, there is less benefit for six-line buffers, most of the programs work as well with four-line buffers as with six (just few programs report an improvement by increasing the number of line buffers from four to six).

### 5.1 Experimental setup

We used SimpleScalar toolset [22] and CACTI 3.2 [23] to conduct our experiments. We have modified SimpleScalar to simulate filter cache, HotSpot cache and single line buffer cache architecture. Our base architecture is using 16 KB direct-mapped level-1 cache with 32 bytes line size. The line buffer used in the HotSpot cache is 32 bytes. We also assumed 512 bytes, direct-mapped L0 cache for filter cache and HotSpot cache. The BTB is four-way set-associative using two-level branch predictor. We evaluated energy consumption using 0.35 $\mu$m process technology. For HotSpot cache, we used value of 16 as candidate threshold as was suggested in [9] and used HotSpot with a line buffer. We used SPEC2000, Mediabench and Mibench benchmarks and datasets, to evaluate the different architectures. Although we ran the simulation for all the programs in these suites, we show the results for some representative applications in these two benchmarks. However, the average is taken over all the programs in the corresponding suite.

Energy per cache access, as is obtained from CACTI, the energy for the line buffer is obtained from [9]. The energy consumption of the different components is shown in Table 1.

### 5.2 Embedded/media applications

We start by presenting our results for applications from Mediabench and Mibench benchmarks. These benchmarks

**Table 1** Energy consumed per access

| Cache | Energy, nJ |
|---|---|
| 256 L0 cache | 0.62 |
| 512 L0 cache | 0.69 |
| line buffer | 0.12 |
| 16 KB direct-map | 1.63 |

represent mainly multimedia applications that are common in portable devices. Our results show that the proposed architecture does save energy without sacrificing the average cache access time.

*5.2.1 Energy:* We now compare energy consumption of multiple predictive line buffers with HotSpot cache, filter cache and single predictive line buffer. Fig. 6 shows the energy consumption normalised to the baseline architecture, which is the direct-mapped cache. The multiple line buffer graphs are generated using four- and six-line buffers. We can see that our proposed architecture consumes less energy than the filter cache, HotSpot cache and single line buffer cache for most of the applications.

For some applications, such as mpeg2 benchmark, the energy consumption is 20% that of the standard direct mapped cache and 40% that of the HotSpot cache. That is because of the match between the loop size and the buffer size. Most of the references of mpeg2 are from the line buffer which require much less energy than the L1 cache. For some applications, such as epic and unepic, there is hardly any improvement by using six-line buffer over four-line buffer. This is because for most loops block in such application, can easily be contained within four-line buffers.

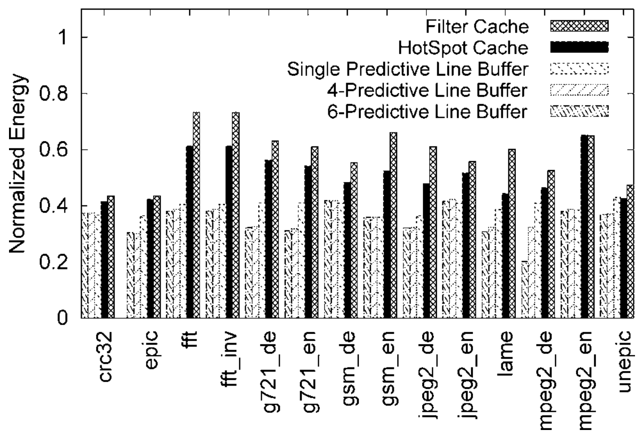

**Figure 6** *Normalised consumption for Mediabench/Mibench*

Table 2 shows average normalised energy and delay of the different architectures over all the programs in Mediabench and Mibench benchmarks. It is obvious that our proposed architecture has a better performance than the other four architectures.

*5.2.2 Delay:* Our proposed architecture does have a lower delay than the other four architectures. And it does not incur any performance overhead compared with the base architecture. Fig. 7 shows the normalised delay for the different architectures. When using six-line buffers, application such as FFT and unepic performs slightly better than the base architecture. The reason is because for such applications, our scheme effectively captures most of loop blocks in the line buffers. That results in avoiding an extra delay of accessing and eliminate thrashing in level-1 cache, as we'll see in Section 5.2.3. We can also see in Fig. 7 that for application such as epic, the four-line buffers perform worse than six- or single-line buffer. That could be explained if a part of the loop is in the line buffers but the entire loop require $>4$. That means there is a miss in every iteration of the loop. We also calculated the normalised energy–delay product for the different architectures. We found that using either four or six predictive line buffer significantly reduces energy–delay product. On the average, for all applications of Mediabench and Mibench, using six-line buffers decreases energy–delay product by 66%, compared with 37% and 47% for filter cache and HotSpot cache, respectively. We did not report on the energy $\times$ delay because of space limitation, the average results are shown in Table 2.

*5.2.3 Off-chip memory access:* Accessing off-chip memory is very expensive in terms of energy consumption and delay. Our proposed architecture does not increase the number of off-chip accesses. We ran simulations to

**Table 2** Various schemes average normalised energy, delay and normalised delay using direct-mapped cache

| Scheme | Normalised energy | Normalised delay | Delay | Normalised energy $\times$ delay |
|---|---|---|---|---|
| Four-predictive line buffer | 0.3581 | 1.0086 | 1.2011 | 0.3612 |
| Six-Predictive line buffer | 0.3458 | 0.9962 | 1.1899 | 0.3445 |
| filter cache | 0.5858 | 1.0800 | 1.2969 | 0.6327 |
| HotSpot cache | 0.5104 | 1.0308 | 1.2330 | 0.5261 |
| predictive line buffer | 0.3932 | 1.0034 | 1.2036 | 0.3945 |

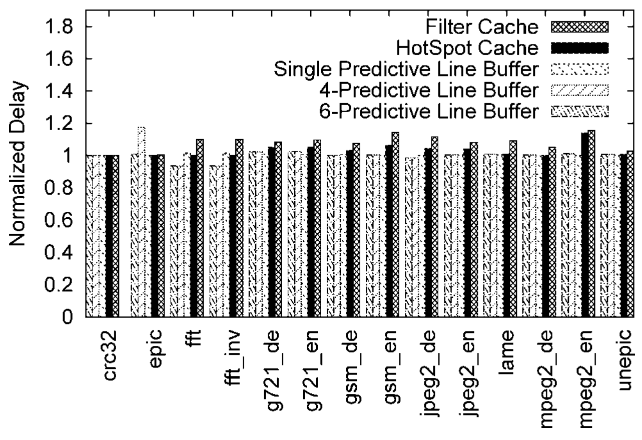

**Figure 7** *Normalised delay for Mediabench/Mibench*

measure the off-chip memory access of our proposed architecture and compared it with HotSpot cache and filter cache. Fig. 8 shows the normalised memory access for the five different architectures. On the average, over all the programs in Mediabench and Mibench, our proposed architecture has 8% less off-chip memory access compared with HotSpot cache and filter cache. This shows that our proposed scheme is effective not only in reducing cache energy consumption but also the overall cache and memory energy.

## 5.3 SPEC2000 applications

Now, we test our new architecture for general purpose CPU intensive applications using SPEC CPU2000 [24]. SPEC2000 is a benchmark suite for testing different aspects of computer performance. SPEC2000 can be used to test and compare the performance of many systems such as general CPU performance, graphics applications, web servers, mail servers etc. In this section, we use CPU2000 and calculate the average memory access time and the power consumption using our proposed cache architecture.

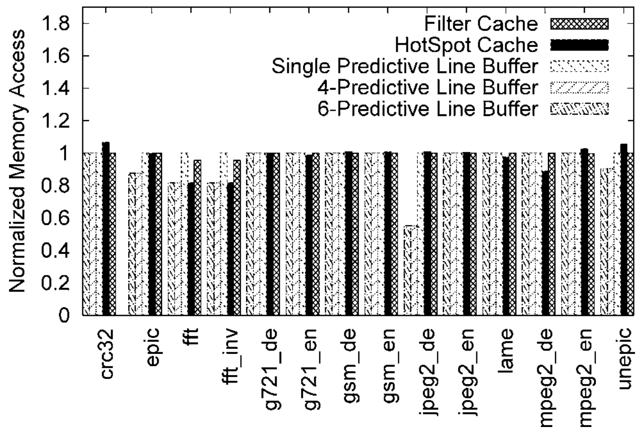

**Figure 8** *Normalised main memory access for Mediabench/Mibench*

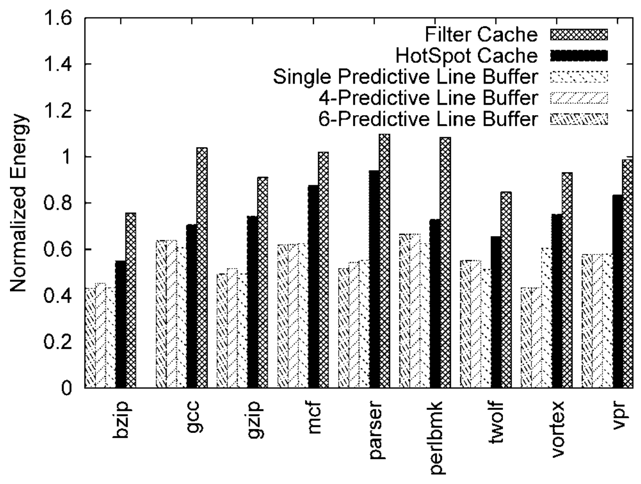

**Figure 9** *Normalised energy for SPEC2000*

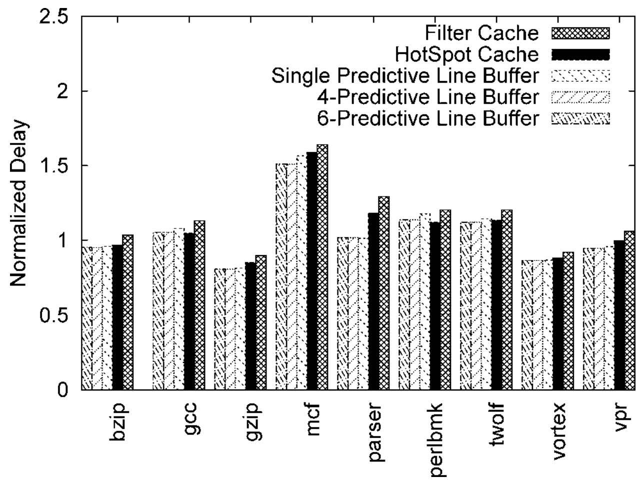

**Figure 10** *Normalised delay for SPEC2000*

*5.3.1 Energy:* Fig. 9 shows the average normalised energy consumption for some representative programs in CPU2000. From that figure, we can see that our proposed scheme with four- or six-line buffers consume considerably less energy than filter cache or HotSpot cache. However, in some applications a single line slightly buffer outperform multiple line buffers (gcc and gzip). Whereas four–six line buffers slightly outperform a single line buffers for other applications (parser, and vortex).

*5.3.2 Delay:* Fig. 10 shows the average memory access time for some representative programs in CPU2000 applications. Again, in that figure we can see that our proposed architecture has a slightly better delay than filter cache or HotSpot cache. Also, for completeness, we must mention here that for our proposed architecture, there was no change in the main memory access pattern for SPEC's CPU2000.

## 6 Conclusion

In this paper, we extended our single predictive-line buffer scheme in order to capture long loops in the line buffers. We presented a cache architecture that utilises 4–6 line buffers, the BTB and a simple prediction mechanism to reduce the energy consumption in the instruction cache. Our simulation results, using Mediabench, Mibench and SPEC's CPU2000 benchmarks show that on the average, our scheme greatly reduces instruction cache's energy compared with a baseline cache, the filter cache and the HotSpot cache without sacrificing performance.

## 7 References

[1] ALY R.E., BAYOUMI M.A.: 'Precharged SRAM cell for ultra low-power on-chip cache'. Proc. IEEE Int. SOC Conf., November 2005, pp. 95–98

[2] CHANG Y.-J.: 'A new register design for low power TLB and cache'. Proc. NORCHIP Conf., November 2005, pp. 301–304

[3] MIZUNO H., ISHIBASHI K.: 'A separated bit-line unified cache: Conciliating small on-chip cache die-area and low miss ratio', *IEEE Trans. Very Large Scale Integr. Syst.*, 1999, **7**, (1), pp. 139–144

[4] ALBONESI D.: 'Selective cache ways: on-demand cache resource allocation'. Proc. 32nd ACM/IEEE Int. Symp. Microarchitecture, 1999, pp. 248–259

[5] ZHANG C., VAHID F., NAJJAR W.: 'A highly configurable cache for low energy embedded systems', *ACM Trans. Embed. Comput. Syst.*, 2005, **4**, (2), pp. 363–387

[6] VIVEKANANDARAJAH K., SRIKANTHAN T.: 'Custom instruction filter cache synthesis for low-power embedded systems'. 16th IEEE Int. Workshop Rapid System Prototyping, (RSP 2005), June 2005, pp. 151–157

[7] ZHU Z., ZHANG X.: 'Access mode prediction for low-power cache design', *IEEE Micro*, 2002, **22**, (2), pp. 58–71

[8] ISHIHARA T., FALLAH F.: 'A non-uniform cache architecture for low power system design'. Proc. 2005 Int. Symp. Low Power Electronics and Design, ISLPED '05, August 2005, pp. 363–368

[9] YANG C.-L., LEE C.-H.: 'HotSpot cache: joint temporal and spatial locality exploitation for I-cache energy reduction'. Proc. 2004 Int. Symp. Low Power Electronics and Design ISPLD'04, August 2004, pp. 114–119

[10] JOUPPI N.P.: 'Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers'. 17th Annual Int. Symp. Computer Architecture ISCA, May 1990, pp. 364–373

[11] ZHANG C., VAHID F., YANG J., NAJJAR W.: 'A way-halting cache for low-power high-performance systems'. Proc. 2004 Int. Symp. Low Power Electronics and Design ISPLD'04, August 2004

[12] BEG A., YUL C.: 'Improved instruction fetching with a new block-based cache scheme'. Proc. Int. Symp. Signals, Circuits and systems ISSCS2005, 14–15 July 2005, vol. 2, pp. 765–768

[13] ROTENBERG E., *ET AL.*: 'A trace cache microarchitecture and evaluation', *IEEE Trans. Comput.*, 1999, **48**, (2), pp. 111–120

[14] KO U., BALSARA P.T., NANDA A.K.: 'Energy optimization of multilevel cache architecture for RISC and CISC processors', *IEEE Trans. Very Large Scale Integr.*, 1998, **6**, (2), pp. 299–308

[15] HANSON H., HRISHIKESH M.S., AGARWAL V., KECKLER S.W., BURGER D.: 'Static energy reduction techniques for multiprocessor caches', *IEEE Trans. Very Large Scale Integr. Syst.*, 2003, **11**, (3), pp. 303–313

[16] BENINI L., MACII A., NANNARELLI A.: 'Code compression architecture for cache energy minimization in embedded systems', *IEE Proc. Comput. Digit. Tech.*, 2002, **149**, (4), pp. 157–163

[17] CHANG Y.-J., LAI F.: 'Dynamic zero-sensitivity scheme for low-power cache memories', *IEEE Micro*, 2005, **25**, (4), pp. 20–32

[18] LEE M., SEO E., LEE J., KIM J.-S.: 'PABC: power-aware buffer cahe management for low power consumption', *IEEE Trans. Comput.*, 2007, **56**, (4), pp. 488–501

[19] QUERSHI M., PATT Y.: 'Utility-based cache partitioning: a low-overhead, high-performance runtime mechanism to partition shared caches'. Proc. Annual IEEE/ACM Int. Symp. Microarchitecture, July 2006, pp. 423–432

[20] HASEGAWA A., KAWASAKI I., YOSHIOKA S., KAWASAKI S., BISWAS P.: 'SH3: high code density, low power', *IEEE Micro*, 1995, **15**, (6), pp. 11–19

[21] ALI K., ABOELAZE M., DATTA S.: 'Predictive line buffer: a fast energy efficient cache architecture'. Proc. IEEE Southeast Conf. 2006, January 2006

[22] The Simplescalar simulator, available at: www.simplescalar.com, May 2006

[23] SHIVAKUMAR P., JOUPPI N.: 'CACTI 3.0: An integrated cache timing, power, and area model', Technical Report, 2001.2 Compaq Research Lab'2001

[24] SPEC CPU2000, available at: www.spec.org