Modified Hotspot Cache Architecture: A Low Energy Fast Cache for Embedded Processors

Kashif Ali Mokhtar Aboelaze Suprakash Datta

Dept. of Computer Science and Engineering
York University
Toronto, ON. CANADA
Email: {kashif,aboelaze,datta}@cs.yorku.ca

Abstract— **The cache memory plays a crucial role in the performance of any processor. The cache memory (SRAM), especially the on chip cache, is 3-4 times faster than the main memory (DRAM). It can vastly improve the processor performance and speed. Also the cache consumes much less energy than the main memory. That leads to a huge power saving which is very important for embedded applications. In today's processors, although the cache memory reduces the energy consumption of the processor, however the energy consumption in the on-chip cache account to almost 40% of the total energy consumption of the processor. In this paper, we propose a cache architecture, for the instruction cache, that is a modification of the hotspot architecture. Our proposed architecture consists of a small filter cache in parallel with the hotspot cache, between the L1 cache and the main memory. The small filter cache is to hold the code that was not captured by the hotspot cache. We also propose a prediction mechanism to steer the memory access to either the hotspot cache, the filter cache, or the L1 cache. Our design has both a faster access time and less energy consumption compared to both the filter cache and the hotspot cache architectures. We use Mibench and Mediabench benchmarks, together with the simplescalar simulator in order to evaluate the performance of our proposed architecture and compares it with the filter cache and the hotspot cache architectures. The simulation results show that our design outperforms both the filter cache and the hotspot cache in both the average memory access time and the energy consumption.**

I. Introduction

The processor speed is, and has been for a long time, advancing at a much higher rate than the main memory access time. That led to a big, and still growing, gap between the processor speed and the main memory speed. Cache, especially on-chip cache, is considered to be the main solution for the processor-memory gap. By placing a small cache memory on the chip between the CPU and the main memory, and having a good cache management scheme, we can reduce the memory access time to 1 or 2 cycles in most processors. On-chip cache is considered to be a standard on almost all different types of processors.

Another reason for the popularity of caches is the growing demand for mobile (battery operated) devices that include one or more processors. Cache is ideal for such applications since it consumes much less energy than the main memory. Although SRAM technology that is used for cache requires more energy per bit access than the DRAM memory used in the main memory, however, the small size of the cache, and the fact that it is on chip results in much less power consumption per access compared with the main memory. Even for more powerful desktop computers, the power consumption is a very important factor in the design process since it affects both the reliability of the system, as well as the price of the system because of the need of sophisticated cooling systems for high power processors.

For modern processor systems, more than half of the chip area, and more than half the transistors count on the chip is dedicated to the cache. That means the power dissipation in the cache is an important part of the overall power consumption in modern processors. It also means that reducing the cache power consumption is very important objective for today's processor's designers.

Energy consumption in the cache can be reduced using three different techniques. The first is at the physical (VLSI) level. In this approach, cache memory is designed with a reduced power consumption in mind. This can be achieved by reducing the voltage levels, reducing the capacitance, or reducing the switching frequency. Second, at the compiler level, some techniques are used in order to fully utilize every data element that was brought to the cache before replacing it (may require some transformations on the source code level). The third approach at the cache management level. This approach includes using different associativity, different levels/types of cache, prediction schemes, and different replacement policies. This approach is usually referred to as *cache architecture*. In this paper we concentrate on the third approach, cache architecture level.

In this paper, we introduce a new instruction cache architecture that results in reducing the average memory response time as well as the power consumption. Detailed simulation using the Simplescalar simulator with Mibench and Mediabench benchmarks, shows that our architecture has a better response time and less power consumption than the hotspot architecture and the filter cache architecture.

The organization of the paper is as follows: Section 2 presents a brief overview of the previous attempts to reduce memory access time and/or the cache power consumption.

In Section 3 we present our architecture and the prediction mechanism. Section 4 presents our simulation results (both energy consumption and average cache access time) for our architecture and compares it with two similar architectures (filter cache and hotspot cache). Section 5 is a conclusion.

II. Low Energy Cache

With increasing the size of the on-chip cache, and having more than one level of caches on chip, that resulted in more power dissipation in the cache. Coupled with the fact that caches are implemented using SRAM instead of the more power thrifty DRAM technology for speed purpose, addressing the power dissipation in caches and trying to minimize it became a very important issue in the design of today's processors. In the rest of this section we present some previous attempts in reducing the cache energy consumption and/or the average cache access time.

Jouppi in [6] introduced victim caches, and stream buffers in order to improve the performance of baseline caches. The victim cache is a small cache (1-5 lines) that holds the victims of recent misses. In case of a miss, the victim cache is checked before the main memory. Victim cache reduces the conflict miss in a low associativity (including direct mapped) cache. Stream buffers are buffers that hold prefetched lines starting at the miss address, which was found to be effective in reducing capacity and compulsory cache misses. Their extensive simulation showed that both victim cache and stream buffers could improve the performance of a baseline cache.

The authors in [3] proposed a cache architecture to reduce energy consumption. Their technique depends on three improvements in order to reduce power. They used multiple line buffers. They check the line buffer in parallel with tag checking in cache, if the data is found in one of the line buffers, the ache access is aborted. Second, they divided the data array into sub-banks, thus saving power on the bit line energy. Finally, they used bit line segmentation for a further power saving. They compared their design to a conventional cache with no line buffer, and showed a large energy reduction.

In [14], the authors proposed the use of a small, energy-efficient filter cache. The authors proposed a small filter cache, and they used the spatial hit/miss pattern in order to predict the next access and to minimize the total cache energy, as well as the total cache access delay. Their design resulted in an energy delay saving of 7%.

Way prediction and selective direct mapping was used in [12] in order to reduce the L1 cache dynamic energy without degrading the cache access time. Their objective was to reduce the energy wasted in accessing all the ways in a set associative cache. Since, at most, the data is found in only one way, they predicted the access way and accessed it as a direct mapped. Their technique resulted in achieving the energy-delay of a sequential access while maintaining the the performance level of a parallel access.

In [15], the authors proposed the hotspot cache in order to reduce energy consumption in embedded systems. In their design, they proposed a small filter cache (known as the hotspot cache) between the L1 cache and the CPU. Their goal is to capture loops in the hotspot cache whose access requires much less energy than the much bigger L1 cache. They modified the Branch Target Buffer (BTB) in order to determine which loops will be loaded into the hotspot cache. Their results show up to 52% energy reduction in cache access using mediabench suite without performance degradation.

The authors in [1] proposed a highly associative cache using CAM design. Since CAM requires high-energy consumption, they used a last-used prediction technique in order to reduce energy using a 32-way set associative cache; they showed 30-40% energy reduction using Mibench [4].

In [5] the authors proposed a technique for way prediction. By accessing only the predicted way instead of accessing all the ways in a set associative cache, energy consumption can be reduced. Their results show a reduction in the energy-delay product of 60-70%.

In [11] the authors proposed a replacement policy to reduce energy consumption in data caches. They used a skewed mapping function such that data close to each other in the memory are no longer close to each other in the cache. They compared their replacement policy to LRU policy and show that their design consume less energy (average saving of 35%) compared to the LRU policy.

Location cache was introduced in [10] as a level-2 cache in order to reduce energy consumption. Their technique applies to large set-associative caches and depends on a small cache called location cache that stores the locations of future cache reference. If the location cache can correctly predict the location of the next reference, the cache is accessed as direct mapped cache instead of a set associative cache. Direct mapped cache consumes less energy since there is only one access to the tag.

Victim cache was proposed in [9] to reduce the delay-energy product and the delay-energy-area product. They used a comparison between the tag high/low order bits and the access high/low order bit in order to quickly detect most of the miss in the filter cache and direct the access to the L1 cache. Their proposed scheme resulted in an average saving of 8.6% for energy and 3.8% for execution time.

Cluster miss prediction was used in [2] in combination with prefetch on miss in order to minimize miss rate for ready CPU cores where the designer does not have complete access to cache configuration. Their simulation shows a reduction in the miss rate

III. Proposed Architecture

In this section we start with a brief description of both the filter cache and the hotspot cache, then the motivation of our work showing some of the shortcomings of the hotspot cache, and why do we propose to modify it. Then, we propose and discuss our proposed architecture.

A. Filter Cache

Filter Cache, introduced in [8], adds a small cache (L0 cache, usually 512 bytes) in front of level-1 cache. The main idea is to capture the most recent accessed instruction to avoid accessing level-1 cache. For each memory access, the filter cache is accessed first and L1 cache is accessed only if the filter cache misses. Filter caches usually result in energy saving, but increase the average cache access time. Filter cache is shown in Fig-1.

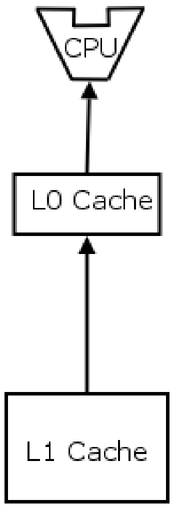

Fig. 1. Filter Cache

B. Hotspot Cache

Hotspot cache [15] avoids such performance degradation by having a steering mechanism to dynamically select between the L0 (hotspot cache) and the L1 cache. The L1 cache is augmented with a block buffer to capture spatial locality as shown in Fig-2. The main idea of the hotspot cache is to capture hot basic blocks (blocks of code that are executed more than a specific number of times) during various phases of the execution.

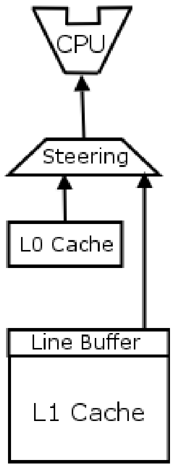

Fig. 2. Hotspot Cache

The block detection mechanism is incorporated into Branch Target Buffer (BTB) and is shown in Fig-3. Each entry in the BTB is augmented with a valid bit, an execution counter, Hot-Block Flag, and Prev-Hot Flag. Execution counter is used to identify hot blocks (the counters in the BTB are used to count the frequency of taken branch or how many times a specific loop was executed). A Hot Block is detected when this execution counter reaches a certain threshold. The system operates in two modes: profiling and monitoring. In the profiling stage, the system counts how many times each branch in the BTB is executed. Once a hot block is detected (using execution counter), that block is transferred to the hotspot cache and monitoring stage starts. In the monitoring stage, the hotspot access is monitored to be sure that it accounts to at least 50% of the cache access. Once the percentage of the hotspot access falls below 50% the system goes back to the profiling stage in order to detect a new hotblock.

An up/down counter called monitor counter (8-bits, initially set to 128) is used to count of the number of times each branch in the BTB is executed. This counter is decremented by one whenever a hot branch is executed and incremented otherwise. When this counter overflows the system goes back to profiling stage, clearing execution counter, and reseting the Hot-Block-Flag for every BTB entries. The system still allow access to L0 cache for hot Block during profiling stage, by keeping the hot block information for previous phase in Prev-Hot-Flag. For each access, if either hot-block or Prev-Hot-Flag is set, it will be directed to L0 cache. The Prev-Hot Flag is cleared once the system goes back into monitoring stage. Mode controller controls whether the instruction will be fetched from L0 or L1 cache during Instruction Fetch (IF) stage. The prediction mechanism to access the hotspot or the L1 cache helps to minimize the number of misses in the hotspot cache.

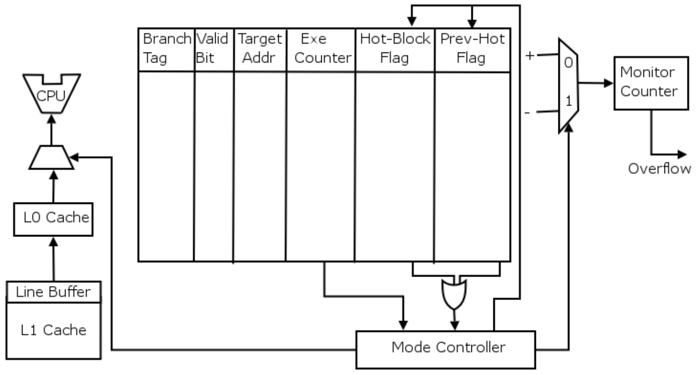

Fig. 3. Block Diagram of Hotspot Cache

C. Motivation

The hotspot cache [15] has good performance in both miss ratio and energy consumption. However detailed simulation using the Mibench suite [4] and the simplescalar simulator [13] show the following.

- Loops whose number of iterations were less than the threshold were not marked as hotspot and never moved into the hotspot cache
- All the hotspot cacheable code below the threshold value was accessed from the L1 cache.

- Multiple hotspot cache code that could not fit simultaneously in the hotspot cache are replacing each other in the hotspot cache.

Table-I shows the percentage of loop iterations that were not caught by the hotspot Cache for various types of applications. Each application ran a maximum of 500 Millions instructions. (See IV-A for more details about the simulation). The Table shows results for the loop iterations that were not caught by the hotspot cache due to the above mentioned reasons. As we can see, for some applications in communication, video and voice hotspot capture loops iteration quite effectively but not so for other types. For example, for applications in Data, Image and Mp3 type, up to 36% of the loop iterations were handled by L1 cache.

TABLE I
AVERAGE LOOP ITERATIONS NOT CAPTURED BY THE HOTSPOT FOR VARIOUS APPLICATION TYPES

Application Type	Loops not captured (%)
Communication	1.05
Data	26.25
Image	10.79
Mp3	36.23
Video	2.48
Voice	2.48

For applications with low L0 (hotspot) cache utilization, such as Mp3 and data applications, low energy consumption can still be achieved by using block buffer. It was reported in [15] that for such applications, if using hot spot cache without block buffer, the energy consumption is even higher then Filter Cache. We can see from Table I that, for such application, up to 36% of iterations were handled by the *relatively* high energy L1 cache.

The use of the block buffer reduces energy consumption since accessing the block buffer requires less energy than the cache. There are three different ways to access the block buffer

- The block buffer is accessed separately in one cycle, if there is a miss, we go to the L1 cache. That approach saves energy by accessing the L1 cache only if there is a miss in the block buffer. The price we have to pay is to waste one cycle in case of a block buffer miss.
- The block buffer is accessed in parallel with the l1 cache. That is fast, but negates the main reason for using block buffer and that is to save energy by not accessing L1 cache.
- The block buffer is accessed first, then if there is a miss, we sequentially access the l1 cache in the same cycle. That saves one cycle, but we have to sequentially access both the block buffer and the cache in the same cycle. usually cache access is on the critical path, which may results in prolonging the cycle time and may result in a slower processor.
- An intermediate solution is to access the block buffer in parallel with accessing the cache tag. If the block buffer hits, we abort the cache access. That means we do not need to extend the cycle time, and the extra energy consumption is reduced, but not eliminated.

In our comparison with the hotspot cache, we simulated the hotspot cache using both the first and third scenarios above.

D. Modified Hotspot Architecture

We now propose our scheme which enjoys a faster memory access time than the hotspot cache, and less energy consumption than the Filter Cache. In our scheme, we did not augment the L1 cache with block buffer for the reasons mentioned above (either longer cycle time, or high energy consumption).

Our proposed architecture is as follows: There are 2 parallel caches between the L1 cache and the CPU, the hotspot cache, and a filter cache. The hotspot cache is used to capture the hot blocks according to [15]. The filter cache is used to capture the loops that were not captured by the hotspot cache. We also used a steering mechanism to forward the memory access to the hotspot cache, the filter cache, or the L1 cache. In case of a misprediction by either the hotspot cache or the filter cache, we go to the L1 cache. If a miss by the L1 cache, then we access the main memory or the L2 cache depending on the architecture. Fig-4 shows the architecture of our proposed cache.

We assumed the same mechanism used in [15] for the hotspot cache. The BTB is augmented with a counter for every branch in the table. the counter counts how many times the branch is taken. If that counter reaches a threshold value, the block is assumed to be a hot block and is moved to the hotspot cache. While the block is in the hotspot cache, the block is monitored to be sure that at least 50% of the references are from the hotspot cache. If the ratio falls below that, the block is considered a *cold* block and the search for another hot block starts.

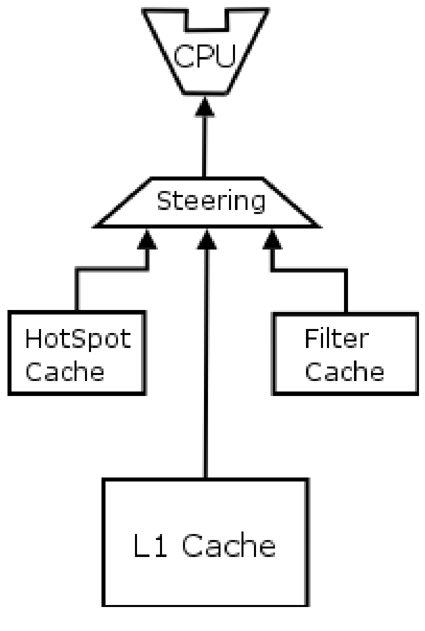

Fig. 4. Modified Hotspot Cache

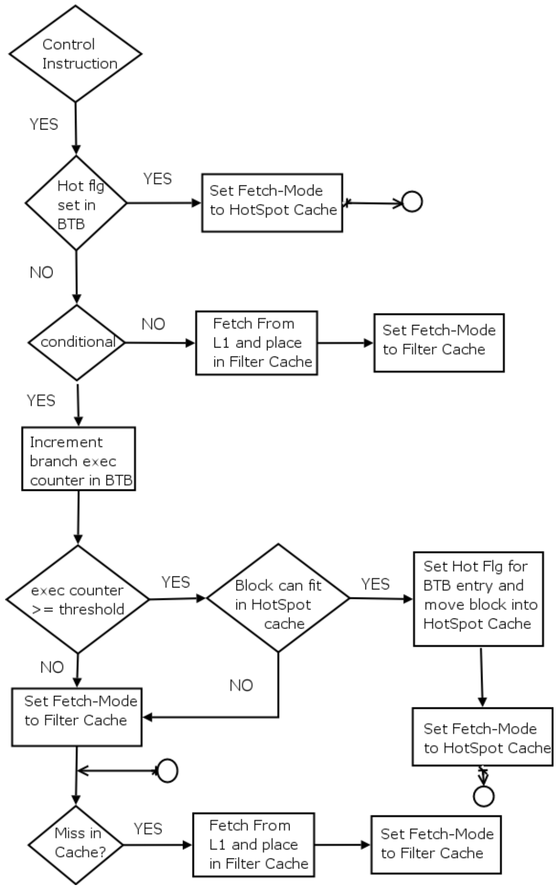
Control Instruction
YES
Hot flg set in BTB
YES
Set Fetch-Mode to HotSpot Cache
NO
conditional
NO
Fetch From L1 and place in Filter Cache
Set Fetch-Mode to Filter Cache
YES
Increment branch exec counter in BTB
exec counter >= threshold
YES
Block can fit in HotSpot cache
YES
Set Hot Flg for BTB entry and move block into HotSpot Cache
NO
NO
Set Fetch-Mode to Filter Cache
Set Fetch-Mode to HotSpot Cache
Miss in Cache?
YES
Fetch From L1 and place in Filter Cache
Set Fetch-Mode to Filter Cache

Fig. 5. Flow Diagram for Steering Mechanism

Our basic Idea is to avoid accessing the L-1 cache and capture branches whose execution threshold did not promote them into hot spot cache into the parallel filter cache. Our steering mechanism chooses between hotspot cache, filter cache and the L1 cache. A flow diagram for our steering mechanism is shown in Fig. 5. The flow diagram shows how the prediction mechanism works. *fetch_mode* indicates the memory we access for the next instruction. For all the non-control transfer instructions, we use the last set *fetch-mode*. For the control transfer instructions, if the branch is marked as a *hot block* in the BTB, we set the mode to *fetch_from_HS*. If the instruction is not a hot block instruction, then if it is a non-conditional control transfer (probably jumping to a function or a subroutine), we fetch from L1 and place the block in the filter cache, the fetching mode is set to *fetch_from_FC*. Otherwise, we fetch from the filter cache and set the fetching mode to *fetch_from_FC*. In case of a miss, we go to L1 cache. Now we present our simulation results and compare it with the hotspot cache and the filter cache.

IV. Simulation Results

A. Experimental Methodology

We used Simplescalar toolset [13] and CACTI 3.2 [7] to conduct our experiments. We have modified Simplescalar to simulate Filter Cache, Hotspot cache, and our proposed modified hotspot cache. Our base architecture is using 16KB 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 the modified hotspot cache. The BTB is 4-way set-associative with 512 sets and 2-level branch predictor. We evaluated energy consumption using $0.35\mu m$ process technology. For the hotspot cache, we used value of 16 as candidate threshold as was suggested in [15]. We simulated hotspot cache twice. Once assuming that the block buffer is accessed sequentially in the same cycle as the L1 cache, and another time assuming it will be accessed in a separate cycle. Since we are using the number of cycles as a measure of memory delay, that underestimate our proposed architecture by not taking into account the increase in the cycle required to access the block buffer and the L1 cache in the same cycle in the hotspot cache architecture. We ran our simulation for a up to 500 millions instructions per program, or to program completion if less than 500 millions instructions. As we are interested in multimedia applications, we use Mediabench and Mibench benchmarks to evaluate the various schemes. We simulated our architecture using the entire Mediabench and Mibench benchmarks. However, we choose to report the results of some representative programs because of space limitations. We reported on sets of encoder/decoder for different media types data, voice, image, video and communication. The results for the rest of the programs are almost identical to the ones reported here. Table-II shows the programs we reported on in this paper. Table-III shows energy per cache access and is obtained from CACTI.

TABLE II
Benchmark Applications Summary

Application	Type	Benchmark
crc32/fft	Communication	Mibench
epic	Data	Mediabench
G721	Voice	Mediabench
Jpeg	Image	Mediabench
Lame	Mp3	Mibench
mpeg2	Video	Mediabench

TABLE III
Energy consumed per access

Cache	Energy
256 L0 Cache	0.62nJ
512 L0 Cache	0.69nJ
Line Buffer	0.12nJ
16KB Direct-Map	1.63nJ
512KB Direct-Map	12.04nJ

B. Energy

We now compare the energy consumption of our proposed architecture with the energy consumption of the filter cache and hotspot Cache. Figure-6 shows the energy consumption of some representative programs in Mibench and Mediabench benchmarks normalized to the baseline architecture (assuming that the baseline architecture energy consumption is 1), which is the direct-mapped cache. For our scheme we used 256 and 512 bytes filter cache along with 512 bytes hotspot cache. Table IV shows the average (over all the simulated programs in Mibench and Mediabench) energy consumption of our proposed architecture and compares it with the other two architectures. Note that in this table, hotspot cache* means the hotspot architecture where the line buffer will be accessed in a separate cycle in order to not increase the cycle time. We can see that our proposed architecture consumes less energy than the Filter cache, and almost the same energy consumption as the hotspot Cache. As we will see the delay and the off-chip memory access of our proposed architecture is better than the hotspot and filter cache. For communication and data application, such as crc32 and epic, using 256 Filter cache along with 512 hotspot cache perform slightly better than the other two architectures. The main reason is because for such applications, over 90% of branch targets are at fixed PC-relative distance and doesn't require bigger cache.

TABLE IV
AVERAGE ENERGY AND DELAY FOR VARIOUS SCHEMES NORMALIZED TO BASE ARCHITECTURE

Scheme	Energy	Delay	Energy*Delay
Filter Cache	0.58	1.096	0.60
Hotspot Cache	0.51	1.056	0.53
Hotspot Cache*	0.51	1.172	0.60
Hotspot with 256 FC	0.52	1.069	0.54
Hotspot with 512 FC	0.51	1.026	0.50

TABLE V
AVERAGE ENERGY AND DELAY FOR VARIOUS SCHEMES (DELAY IN CYCLES PER MEMORY ACCESS

Scheme	Energy	Delay	Energy*Delay
Filter Cache	0.58	1.311	0.78
Hotspot Cache	0.51	1.248	0.65
Hotspot Cache*	0.51	1.420	0.74
Hotspot with 256 FC	0.52	1.255	0.67
Hotspot with 512 FC	0.51	1.155	0.59

C. Delay

The filter Cache and the hotspot Cache significantly reduce the energy consumption by avoiding energy expensive level-1 cache access. Both of them have performance overhead. Table-IV shows normalized delay for the filter cache and the hotspot cache. On the average, for simulated applications, up to 10% and 6% performance degradation was observed for Filter Cache and HotSpot Cache respectively compared to the baseline architecture. Using our proposed scheme we reduced the performance overhead to only 2%. As our scheme doesn't use line buffer between level-1 cache and L0 cache, we avoid increasing cache access time. Note also that, in terms of cycles

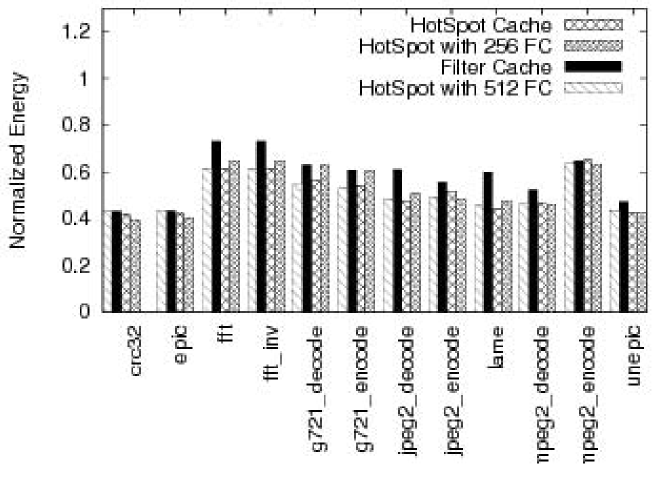

Fig. 6. Normalized Energy Consumption for Filter Cache, Hotspot Cache, Hotspot with 256 bytes Filter Cache and Hotspot with 512 bytes Filter Cache

per memory access, our proposed architecture outperfoms the hotspot cache even when we consider the hotspot cache access the line buffer and the L1 cache in the same cycle.

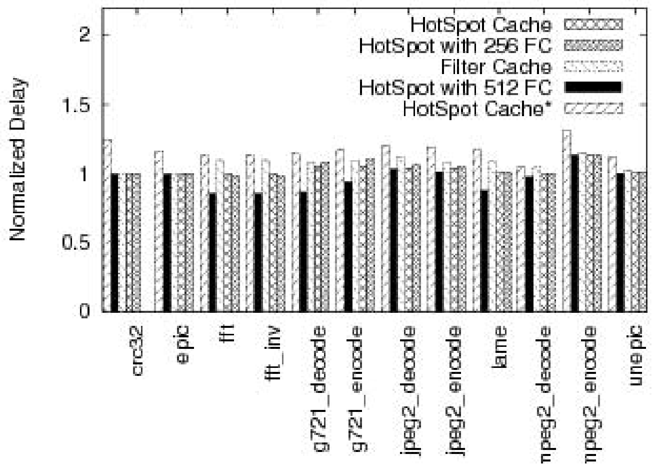

Fig. 7. Normalized Delay for Filter Cache, Hotspot Cache, Hotspot with 256 bytes Filter Cache and Hotspot with 512 bytes Filter Cache

Fig-7 shows the delay normalized to the baseline architecture for filter cache, hotspot cache, modified hotspot with 256 bytes filter cache and modified hotspot with 512 bytes filter cache. As with energy, for both data and communication application the delay is lower when using the modified hotspot with 256 bytes filter cache. Whereas for all other applications the modified hotspot cache with 256 bytes filter cache is actually worse than the original hotspot cache. For the modified hotspot cache with 512 Filter cache, the delay for all programs for various types of applications are better than all other schemes. Fig-8 shows the average memory access time in cycles per memory access for the different architectures.

Fig 9 shows the energy-delay product for all the schemes using representative programs form Mibench and Mediabench,

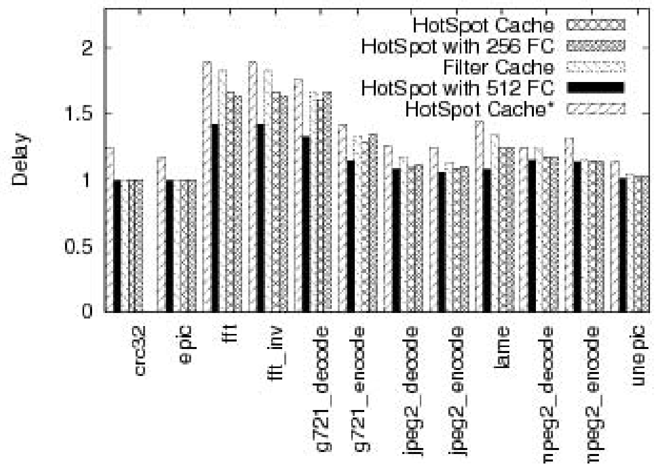

Fig. 8. Delay in cycles per memory access for Filter Cache, Hotspot Cache, Hotspot with 256 bytes Filter Cache and Hotspot with 512 bytes Filter Cache.

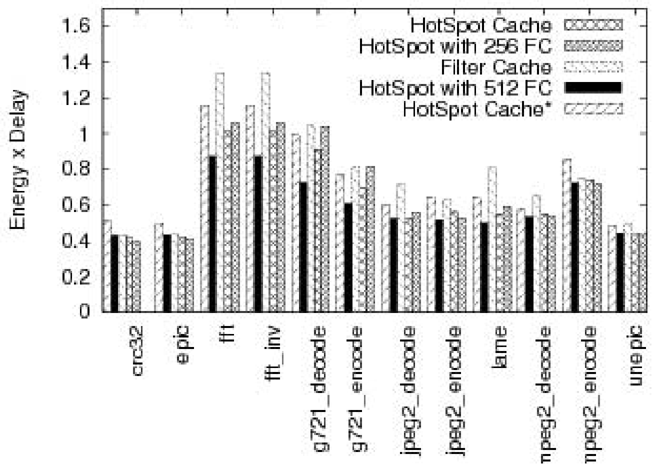

Fig. 10. Energy*Delay for Filter Cache, Hotspot Cache, Hotspot with 256 bytes Filter Cache and Hotspot with 512 bytes Filter Cache

with the average values shown in Table-IV, table-V shows the same results with delay in cycles per memory access instead of normalized delay to the base architecture. Our schemes shows, on average, up to 50% improvement in energy-delay product.

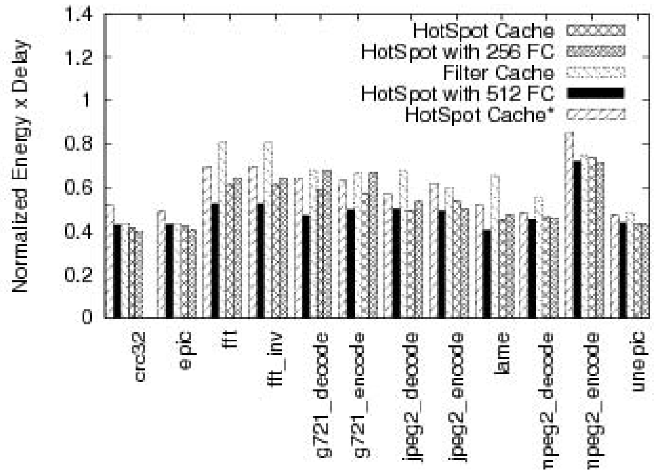

Fig. 9. Normalized Energy*Delay for Filter Cache, Hotspot Cache, Hotspot with 256 bytes Filter Cache and Hotspot with 512 bytes Filter Cache

D. *Off-chip memory access*

Accessing the off-chip memory is expensive, both in terms of energy consumption and delay. Off-chip memory could be the main memory, or an off-chip second level cache (L2 cache). Direct-mapped cache, although comparatively have fast access, can suffers from thrashing problem. Thrashing occurs when two memory lines map to same line in the cache. Thrashing can cause performance degradation as most of the time is being spent 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. Normalized off-chip memory access for various schemes are shown in Fig-11. For some applications such as 'lame' and 'jpeg decode', our proposed architecture reduces off-chips memory access by up to 75%.

TABLE VI
VARIOUS SCHEMES AVERAGE NORMALIZED OFF-CHIP MEMORY ACCESS

Scheme	Normalized Memory Access
Filter Cache	0.992
Hotspot Cache	0.970
Hotspot with 256 FC	0.884
Hotspot with 512 FC	0.539

Table-VI shows the average normalized memory access for hotspot, filter cache and our scheme. As shown, when using our proposed scheme with 512 bytes filter cache reduces off-chips memory access by up to 48%. This in turns will further reduces overall memory (cache and RAM) energy consumption. Fig-12 shows the normalized total (on-chip and off-chip access) energy consumption of the three architectures normalized ot the direct mapped cache.

V. CONCLUSION

In this paper, we proposed a new cache architecture for the instruction cache to minimize both the average cache access time and the energy consumption. Our proposed architecture combines the low miss rate of the hotspot cache architecture and the low energy of the filter cache architecture. Our simulation, using Simplescalar, Mediabench and Mibench, shows a reduction in both the average memory access time and the cache energy consumption compared to both the hotspot architecture and the filter cache architecture.

REFERENCES

[1] Efthymiou, A.; Garside, J.D. "A CAM with mixed serial-parallel comparison for use in low energy caches". IEEE Transactions on Very Large Scale Integration (VLSI) Systems. Volume 12, Issue 3, March 2004 Page(s): 325 - 329

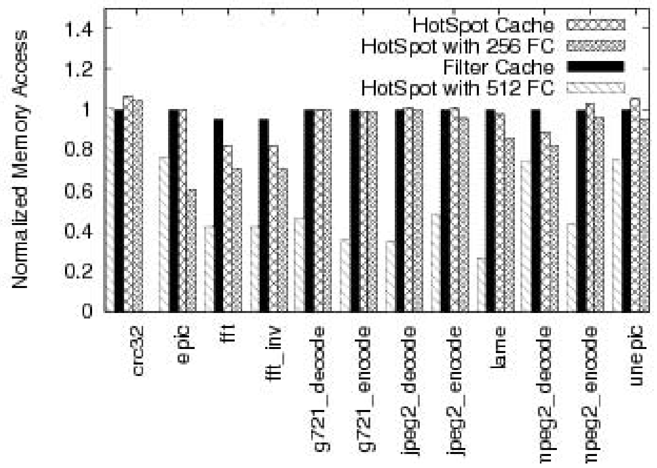

Fig. 11. Normalized Off-Chip Memory Access

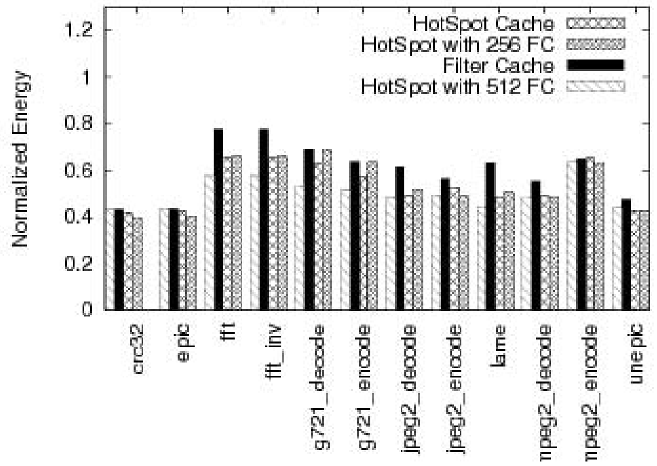

Fig. 12. Normalized Energy Consumption (including off-chip memory access energy consumption) for Filter Cache, Hotspot Cache, Hotspot with 256 bytes Filter Cache and Hotspot with 512 bytes Filter Cache

[2] Batcher, K.; and Walker, W. "Cluster miss prediction with prefetch on miss for embedded CPU instruction cache". Proc. of CASE'04 pp 24-34 Washington D.C. Sept. 22-25 2004.

[3] Ghose, K; and Kamble, M. B. "Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation". Proc. of ISLPED99, San Diego, CA pp 70-75

[4] Guthaus M.; Marsman, E.; McCorquodale, M.; Gebara, F.; Kraver. K.; Zolotov V. "Mibench: A free, commercially representative embedded benchmark suit". *IEEE 4th Annual Workshop on Workload Characterization*. Austin, TX, Dec. 2001

[5] Inoue, K.; Ishihara, T.; Murakami, K "Way-predicting set-associative cache for high performance and low energy consumption". *Proc. of International Symposium on Low Power Electronics and Design ISLPED99* pp: 273 - 275. 1999.

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

[7] Jouppi, N. "CACTI" can be found at www.hpl.hp.com/personal/Norman_Jouppi/cacti4.html

[8] Kin, J.; Gupta, M.; Mangione-Smith, W. "The filter cache: An energy efficient memory structure". *Proc. of the 30th Annual International Symposium on Microarchitecture* December 1997

[9] Memik G.; Reinman G.; and Mangione-Smith, W. "Reducing energy and delay using efficient victim caches". Proc. of the International Symposium on Low Power Electronics and Design ISLPED'03. August 25-27 Seoul Korea pp 262-265 2003.

[10] Min, R,; Jone, W.-B.; and Hu, Y. "Location cache: A low-power L2 cache system". Proc. of of the 20004 International Symposium on Low Power Electronics and Design ISLPED'04 August 9-11 Newport Beach, CA. pp 120-125.

[11] Musalappa, S.; Sundaram, S.; and Chu, Y. "A replacement policy to save energy for data cache". Proc. of the 19th International Symposium on High Performance Computing Systems and Applications HPCS'05. pp641-642. April 2005.

[12] Powell M; Agarwal, A. Vijaykumar, T. N; Falsafi, B; and Roy, K. "reducing set-associative cache energy via way-prediction and selective direct mapping". Proc. of the International Symposium on Microarchitecture, 2001

[13] www.simplescalar.com The simplescalar LLC Oct. 2005

[14] Vivekanandarajah, K; Srikanthan, T.; and Bhattacharyya S. "Energy-delay efficient filter cache hierarchy using pattern prediction scheme". IEE Proc. on Computers and Digital Techniques. Vol. 151, Issue 2, pp 141-146 March 2004

[15] Yang, C.-L; and Lee C.-H "Hotspot cache: joint temporal and spatial locality exploitation for I-cache energy reduction". Proc. of the 20004 International Symposium on Low Power Electronics and Design ISLPED04 pp 114-119 Aug. 2004