A Review of Caching Techniques for Low Power Consumption in Embedded Devices

Aniket Ashwin Samant
Delft University of Technology
Delft, The Netherlands
Email: A.A.Samant@student.tudelft.nl

Apoorva Arora
Delft University of Technology
Delft, The Netherlands
Email: A.Arora-1@student.tudelft.nl

Snehal Jauhri
Delft University of Technology
Delft, The Netherlands
Email: S.Jauhri@student.tudelft.nl

***Abstract*—Embedded systems that operate in the low-power domain need to be as efficient as possible in terms of their components' power consumption, including the on-board processors'. Caches in processors, specifically instruction caches, are known to be power-hungry since instructions are fetched frequently for the processor's operations. This paper reviews and compares some caching techniques specifically intended to lower the power consumption of the instruction cache. The ideas of tiny caches, filter caches and Reduced One-Bit Tag Instruction Cache are discussed and qualitatively compared. These techniques are also compared with traditional instruction caches in terms of performance degradation (if any) and power savings achieved during normal operation.**

Keywords: caching, dynamic power, embedded systems, filter cache, instruction cache, loop cache, low power, short backward branch, tag

I. Introduction

Most of the dynamic power consumed by an embedded processor is through instruction-fetching [1]. The reason is that it occurs in most cycles, involves switching of large numbers of high capacitance wires, and makes accesses to set-associative caches - all of this collectively consumes a lot of power [2]. Thus, it also lays the foundation for using caching techniques in embedded processors to minimize power consumption while fetching instructions.

There are two main research facets of this paper:

- *Fundamental ideas behind the introduced caching techniques and their operation*
- *Factors which influence the power savings for a particular caching technique*

In this paper, we first review multiple tiny cache designs - starting with a direct-mapped filter cache, and then followed by loop caches: dynamic, pre-loaded and hybrid loop caches. All the discussed tiny cache designs are then compared for their power reduction, main instruction-memory access reduction and overall impact on performance. We also discuss and review the energy savings for a cache design that employs a small cache size and a one-bit tag comparison. This kind of cache, the Reduced One-Bit Tag Instruction Cache (ROBTIC), is also aimed at achieving significant power reductions. *This review paper specifically focuses on the findings from [2], [3], [4].*

We see that the techniques explained in this paper indeed reduce the power consumed by the cache subject to certain limitations, as will be explored in the forthcoming sections. We also note that filter caches in general are more suited for applications involving larger instruction block sizes whereas loop caches and ROBTIC are better suited for applications with tiny instruction blocks; however, customization of the caches can be performed as per prior information about the applications.

In the first section, we evaluate the design technique of the filter cache. Loop caches are discussed next (dynamically loaded, pre-loaded, and one- and two-level hybrid loop caches) and compared with traditional I-caches. The ROBTIC is introduced and discussed in the following section, after which some comparisons are drawn between the introduced techniques. The paper ends with some conclusions and ideas for selecting an optimal cache design.

II. Evaluated caching techniques

In this section, we first discuss *filter cache* and explain the concept behind it. We follow it up with a discussion on *loop caching* techniques and the advantages they can offer over filter caches. We cover the dynamically loaded tag-less loop caching technique, pre-loaded loop caches and then introduce hybrid loop caching. We conclude the section with *ROBTIC*, a different approach aimed at reducing tag comparisons without much additional overhead.

A. Filter Cache

A Filter Cache (FC) can offer substantial energy savings in embedded low power applications. The authors of [3] discuss the concept behind using a filter cache and its advantages. Moreover, a method of designing a filter cache whose size is tuned to an application is described. This method uses loop profiling of the application to determine a filter cache size that would maximize savings by reducing the number of accesses to the main Instruction Cache.

1) Concept: Filter Cache (FC) is a small, direct mapped intermediate cache that sits between the processor and the first instruction cache (or L1 cache). It is typically of much smaller size than the L1 cache and its objective is to reduce the number of accesses to the L1 cache. It does this by storing the next instructions for all the small loops of the program. This avoids unnecessary accesses to the L1 cache. In this way, since the

larger L1 cache is not cycled, it leads to substantial energy savings.

2) Points of consideration: Due to its small size and placement close to the execution core, Filter Caches can lead to substantial energy savings, especially if a majority of loops in an application can be serviced using it. Since the loop characteristics of an application can be easily obtained, the Filter Cache can be designed specific to the application. Optimal design of the FC has shown energy reduction numbers above 50% for certain applications [5].

However, a point to consider when adding a FC is that when a cache access 'miss' takes place in the FC (when the required next instruction is not present in the Filter cache), then the L1 cache has to be accessed. Since we now need an extra step to fetch from the L1 cache (going through the Filter Cache), this leads to an overhead and reduces the performance.

To avoid this performance reduction, a parallel prediction method is described in [3]. This method uses an extra parallel path to access the L1 (main) cache. Moreover, a predictor is used which can predict whether the next instruction can be fetched from the FC or not. If the next instruction cannot be fetched from the FC, the alternate parallel path is used to directly access the L1 cache. This avoids the extra step through the FC in case of a miss and leads to a performance improvement (See Figure 1). Modern predictors can achieve close to 90% accuracy in prediction and a design for them is mentioned in [6].

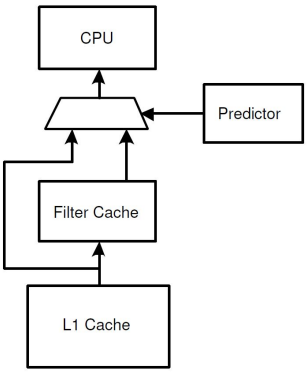

Fig. 1. A Predictive Filter Cache [3]

3) Choosing FC size using Loop Profiling: Designing of the FC for an application implies choosing a FC size that can accommodate a majority of the loops in the application to achieve maximum hits and hence maximize power saving. Note that the FC size cannot be too big otherwise it ends up consuming too much power by itself (due to accesses and static power consumption of such a large cache). A design methodology is described in [3] that uses loop profiling to analyze the application and accordingly chooses a FC size. To avoid unnecessarily long computations, FC size is chosen from a fixed set of: 128, 256, 512 bytes etc.

Given a binary file of the application, Loop profiling is done using algorithms that generate a Control Flow Graph of the program and analyze the back edges of the graph to identify loops. Note that within a loop, the part of the code within it which is executed in at least 20% of the loop's iterations is chosen to be cached in the FC. Once the loops are identified, the energy savings for these loops are calculated analytically and the size with the best relative energy saving is chosen.

- **Energy savings**: Usage of a 256 byte filter cache results in a power reduction of 58% with a 21% performance reduction [5]. Moreover, a comparison was made using eighteen applications from MediaBench between choosing an optimal size for the FC for an application as compared to using a 256 byte FC for all the applications. Tuning the FC using the mentioned design strategy and finding optimal size was found to, on average, further lead to 11% energy savings as compared to general usage of a FC of 256 bytes for all the applications [3].

B. Loop Caching Techniques

As discussed above, similar to the mechanism in a traditional cache, a direct-mapped filter cache also stores part of the address of each instruction, and compares the tag with the desired address to determine a cache hit or miss. Whenever a miss occurs, a microprocessor stall occurs. Tag accesses and comparisons require some energy overhead, while stalls due to misses cause performance and energy overhead [2]. To eliminate these overheads, loop caching techniques were introduced as discussed below:

1) Dynamically Loaded Loop Cache:

- A dynamic loop cache is a tiny cache which is free from tags and misses unlike a filter cache. Whenever a loop is detected in the dynamic instruction flow through a short backwards branch (sbb) instruction, the cache controller is triggered to fill the tiny cache in the next iteration of this small loop. This fill is non-intrusive and the processor continues to fetch and execute from main instruction memory. When the sbb is detected again, the instruction fetch is switched to the loop cache. The fetch continues from this loop cache until the loop is exited [2].
- The advantage of using a dynamic loop cache over a filter cache is that, firstly, no tag comparisons are required in the loop cache, thus reducing overhead and power per access. Secondly, performance does not degrade since filling is done non-intrusively and a hit is always guaranteed which avoids any CPU stalls [2].
- **Energy Savings**: Running Powerstone and MediaBench benchmarks on MIPS and Superscalar respectively, the energy savings results using dynamic loop cache were obtained and examined. It was observed that on average energy savings for all benchmarks was 22% [2]. The experiments with dynamic loop caching show that minimizing the power within the loop cache itself is not a high priority, which makes sense as this power is very small compared to L1 fetching. To really improve a loop cache, the number of supported loops needs to be increased. This is the motivation behind a Pre-loaded Loop cache design.

2) Pre-loaded Loop Cache:

- In a pre-loaded loop cache, the the entire loop is stored in the loop cache during the the microprocessor boot sequence, before the program begins executing. First, the program is profiled to detect critical loops. These are the most frequently executed loops which are stored in the loop cache. Unlike the dynamic loop cache, a pre-loaded loop cache can support change of flow instructions and subroutines and stores multiple loops and subroutines simultaneously [2].

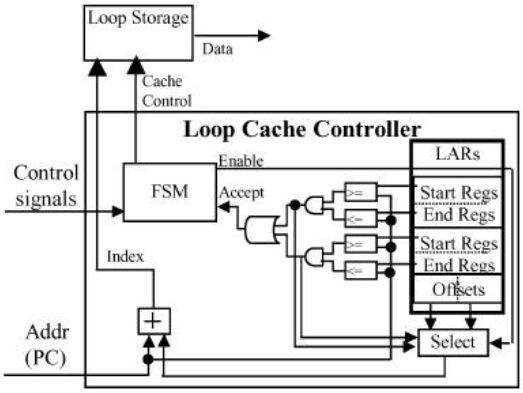

Fig. 2. Architecture of a pre-loaded loop cache [2].

- As shown in Figure 2, a pre-loaded loop cache consists of loop storage, a loop cache controller, and loop address registers (LARs) [2]. The loop cache controller is responsible for filling the loop cache, switching instruction fetches from main instruction memory to the loop cache and vice-versa.
- **Energy Savings**: Improvement in the loop cache hit rates for both MediaBench on SimpleScalar and Powerstone on MIPS was reported. This justifies the observed improvement in the average instruction fetch energy savings of 66% [2].

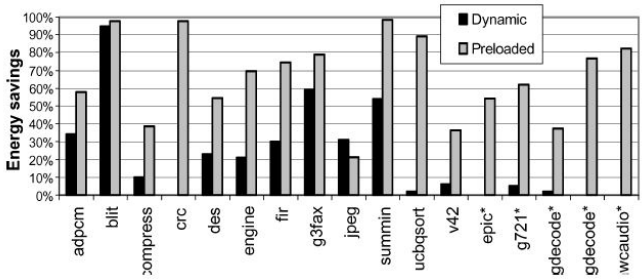

Fig. 3. Percentage of instruction fetch energy saved by dynamic and pre-loaded caches using a 128-instruction loop cache [2].

3) Hybrid Pre-loaded/Dynamic Loop Cache : It was reported that for some benchmarks dynamic loop cache performed better while for some pre-loaded performed better [2]. This became the motivation for designing a hybrid cache which behaved like both dynamic and pre-loaded loop cache.

One Level Hybrid

In the one-level hybrid loop cache, part of the cache storage holds pre-loaded loops, while the rest of the storage is filled dynamically. Thus, the pre-loaded part does not store loops that are handled by dynamic loop caching.

Two Level Hybrid

In a two level hybrid cache, a second level of the storage holds pre-loaded loops while the first level of storage acts like a dynamic loop cache. Moreover, when a short backwards branch is detected, the first (dynamic) level could be filled in both from regular memory or from the second level (pre-loaded) storage. If a change of flow occurs, the target address is checked against the loop address registers. If a match occurs, then the first level loop storage is filled with the appropriate loop from the second level.

- **Energy savings**: Simulation results showed that the one-level hybrid loop cache reduces main instruction memory fetching by 50% for both benchmark suites, even with a very small cache size of 32 instructions with a dynamic portion of 16 instructions [2]. On the other hand, the two level scheme does not seem to perform as well as the other loop caching methods. [2]

C. The ROBTIC

The previously described caching techniques involve the addition of a tiny cache block next to the L1 cache. However, it is also possible to optimize the L1 cache block itself, without the addition of a separate cache block; that is the fundamental idea behind this caching technique. In a traditional instruction cache, the tag accounts for a considerable proportion of the chip area and also consumes a lot of power (for example, a 20-bit tag mapped to a 32-bit PC address leads to the tag occupying about 37.7% of the cache area, in addition to a 20-bit comparator circuitry [4]). To reduce the area occupied by the tag and consequently the tag-comparator size, J.Gu et al [4] introduce a new cache design, namely the ROBTIC, or the Reduced One-Bit Tag Instruction Cache, which employs a technique involving a comparison of just one bit (the LSB) of the tag [4]. There is some additional circuitry involved, but the overall power consumption is lower compared with a traditional cache's case.

ROBTIC's scheme takes advantage of the spatial and temporal locality of programs for which its usage is intended, so as to achieve low power consumption without compromising on the performance of the cache. The main point to note is that in ROBTIC, there is no extra cache block involved as is the case with loop caches, filter caches etc; cache tag comparison reduction is the main idea.

1) High-level architecture: A ROBTIC cache is similar to a traditional cache in terms of its structure - for every cache entry there is a valid bit, a tag field, and a *cache line (for data)* - however, the tag field is only one bit wide. The valid bit together with the tag bit is used to determine a cache hit or miss. This modification does require some additional circuitry though, and it is present in the form of a cache operational control unit, which will be explained in a more detailed fashion

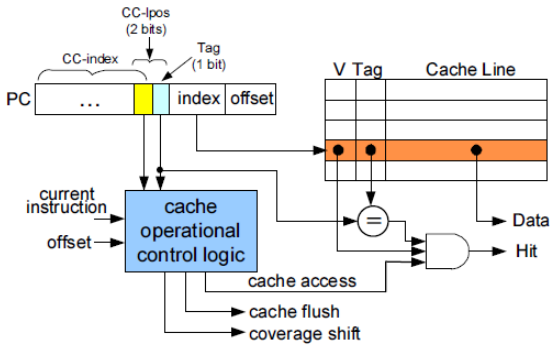

Fig. 4. ROBTIC Architecture [4]

soon. But as Figure 4 suggests, the overall architecture is quite similar to a traditional cache's.

2) Assumptions made: There are some assumptions made regarding the ROBTIC cache's operation, viz.:

- The applications that run on the processor mainly involve *loop basic blocks*, that is, instruction blocks that are repeatedly fetched and hence are spatially and temporally local.
- A cache line holds multiple instructions, all instructions being of equal size
- A standard cache size of 32 lines (2^5) is defined for ROBTIC, based on surveys suggesting that more than 90% of loops contain fewer than 30 instructions [7] [8]
- The instruction cache is directly mapped to avoid the cache replacement policy issue (that is, if there are sets present in the cache, an algorithm would be needed to decide which exact cache items must be replaced).

3) ROBTIC Operation: The authors of [4] define cache coverage (CC) as the "cache-mappable address space" of the main memory. In a full-tag cache, the tag size is chosen so as to map the entire memory address space using the cache tag; that is, to have the entire memory space in the CC. However, in ROBTIC, since the tag size is one bit, the tag by itself cannot be used for covering the entire address space.

Thus, for CC in ROBTIC, the first few MSBs from the program counter (PC) are used directly along with the 1-bit tag to uniquely determine a cache block; the first few MSBs, collectively constituting the "MSB set", form the *CC-index* as shown in Figure 4. *We clearly see that we are left with only a 1-bit comparison and a series of small operations for determining the CC-index, instead of a full-tag comparison, which is a much more power-hungry operation. This is how the ROBTIC achieves its power consumption reduction.*
The CC-index changes when the PC changes its MSB set based on the instruction block to be executed. The existing cache is flushed and the next set of instructions gets cached (corresponding to the next CC-index).

4) A point to consider: In an ideal case, all instructions in a block would be aligned so as to be local to a CC-indexed block. However, generally, a block of instructions could also be spread over two CC-index values (that is, the instruction block could have parts in different CC-index values) and cause an undesired cycle of cache flushing brought about by the two parts of the instruction block evicting each other for every execution cycle (referred to as the "Ping-pong effect" in [4]). To tackle this issue, an "overlapped dynamic cache coverage control" scheme is proposed.

5) Dynamic cache coverage scheme: The main focus is on not flushing the cache immediately when the CC-index changes, and letting the currently cached instructions finish execution instead. The authors discuss three control flows that could arise in the ROBTIC's operation states, viz.:

- Cache coverage shift (CCS): When the CC-index changes due to the PC's movement, the cache may need to be retained or flushed depending on how the loop block is positioned in memory.
- Cache flush: Invalidates all the instruction entries in the cache. It is assumed that all jump instructions and branch instructions with an offset larger than the cache size should be followed by a cache flush operation (hops are assumed to be long distance in such cases.)
- Normal Cache Access: As is the case with a traditional cache, if a PC address is within the current CC region and the instruction to be executed gets a hit in the cache, it should be fetched and executed normally.

6) Algorithm for detecting CCS: A crude and straightforward method of detecting "*overlapped cache coverage*" is to compare the top and bottom tags of the CC region with the PC address's full tag to know if a CCS is involved, and take actions accordingly. However, with the full-tag comparison involved, the purpose of having a 1-bit tag is defeated. Hence, the authors of [4] make use of a simpler approach in which only three consecutive CC regions need to be differentiated to determine if a CCS occurs, and this differentiation is performed on the basis of the two LSBs of the full tag from the PC (referred to as the "*CC-lpos*", as shown in Figure 4). The CC-lpos bits are converted to their corresponding Gray code representation and it is used to determine if a CCS takes place. Two values need to be stored - the "common value" (**CV**) of the Gray code representation (i.e. the bit value between consecutive 2-bit Gray-code numbers that doesn't change), and its corresponding bit position (**CVP**). The Gray code representation of the CC-lpos for each new PC address is computed, and the common value is checked with the CC region's common value. If they're the same, a normal cache access is performed, and if different, it implies the CC region needs to be shifted. The new CC region's CV and CVP values are calculated accordingly. This algorithm clearly requires very little additional circuitry and hence does not significantly increase the chip area or the power consumption.

- **Energy Savings:** The ROBTIC cache design was tested by the authors of [4] on applications from the Motorola Powerstone and MiBench benchmark suites and compared with the results obtained by running the same applications on a traditional cache design. For a general

ROBTIC cache (with 32 cache lines) power savings and performance metrics were compared, and in most cases the ROBTIC cache provided a significant amount of power savings (in the range of 25 to 30%) at almost the same performance; the authors noted a few cases, however, in which there was a significant performance degradation due to non-locality of programs. Furthermore, for specific applications in which the loop block size is considerably different, varying the size of the ROBTIC cache between 8 and 128 lines added to the overall power savings due to the increased cache hit rate.

III. Comparative Study: Caching Techniques

This paper reviews various caching techniques for low power consumption as explained in the above sections in detail. In this section we highlight some of the advantages and disadvantages of each of discussed techniques to aid a comparative study.

Filter vs. Loop Caches

On reviewing the papers on Filter and Loop caching techniques, we see that for a small loop cache of size 32 instructions, a one-level hybrid loop cache gives maximum energy savings [2]. On the other hand for larger sizes of the caches, a filter cache performs the best (Filter cache achieves more than 50% energy savings for Powerstone benchmarks [3]).

Figure 5 highlights the difference in memory hierarchy of the filter and loop cache. A key advantage of loop caches is that they are tag-less and thus hits are guaranteed, resulting in no performance overhead. When this relative performance overhead is large enough, it also leads to less energy consumption as compared to the filter cache.

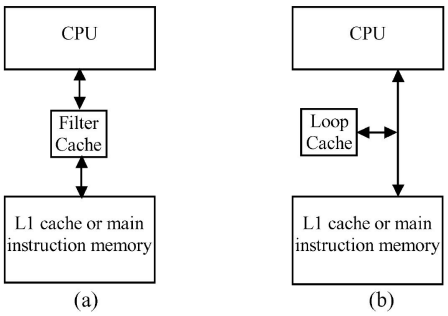

Fig. 5. Memory hierarchy roles for (a) a filter cache (b) a loop cache [2].

A comparison of the different loop caching techniques was done in [2] and is summarized in Table I. We can conclude from this comparison that among the loop caching methods, the one-level hybrid loop cache seems the better choice. It achieves more energy savings than a pure dynamic loop cache. Though the one-level hybrid performs slightly worse than a pre-loaded loop cache, it also has the advantage of operating in a dynamic-only mode and avoid pre-loading altogether and hence is a better choice.

TABLE I
A comparison of Loop Caching techniques

Type of Loop Cache	**Average Energy Savings**	**L1 cache access reduction**
Dynamic	30%	30% L1 fetch reduction no matter how large the loop cache size
Preloaded Loop cache	66%	40% for a loop cache of size 32 instructions with six preloaded regions of code.
One Level Hybrid	60%	50% for a loop cache of size 32 instructions and dynamic portion of 16 instructions
Two-Level Hybrid	56%	30% for first level size of 32 instructions

(Energy savings are for a 128-Instruction Dynamic Loop Cache for Powerstone benchmarks [2])

Comparing ROBTIC with Filter and Loop Caches

Filter and loop caches are supplementary caches added to an existing L1 cache to reduce main memory accesses and hence have to be small enough so as to not offset the benefit of reduced access times with the demerit of increased area. The ROBTIC, on the other hand, is a scheme for reduced tag comparisons in the instruction cache, since the tag comparison circuitry otherwise accounts for significant power consumption. Unlike the filter cache and the loop cache, ROBTIC is a kind of I-cache with a different scheme for fetching cached data, rather than a supplementary cache to an existing I-cache. Given that the complexity of the logic involved is constant, this scheme is scalable to larger sizes of the cache and memory, so as to get reduced area and energy consumption with little or no performance loss.

There are some limitations to the ROBTIC cache though: jump instructions needn't always lead to a cache flush, and could be handled based on whether the jump is long or within the CC area. Moreover, ROBTIC is only a direct-mapped cache without any associativity (as stated in the initial assumptions).

IV. Conclusions

Each of the evaluated caching techniques relies heavily on its target applications being *local* (that is, having successive instruction addresses close to each other), Since this is the case in most embedded applications [4]; they are all viable solutions for low power consumption in embedded applications with lots of loop-basic blocks. Moreover, given the fact that most embedded processors run the same application throughout their lifetime of operation, the caching technique and the cache size can be customized as per the application. Thus, we can choose between the evaluated caching solutions based on the application's loop profile and the area available for a supplementary cache (For filter and loop caches).

We can conclude that:

TABLE II
RELATIVE COMPARISON OF THE PRESENTED CACHING TECHNIQUES

Parameters	**Filter cache**	**Loop cache**	**ROBTIC**
Area increase	Yes	Yes	Minimal
Target program assumptions	Lots of loop basic blocks present	Tiny loop basic blocks present	Lots of loop basic blocks present
Relative Power savings	Large	Large for small cache sizes	Limited
Cache type	Direct-mapped	Tag-less	Direct-mapped
Performance degradation	Large if no parallel prediction	Minimal	Minimal
L1 cache access reduction	Large	Large if program has mostly tiny loops	N/A (ROBTIC is the L1 cache)

- A Filter Cache offers size-able power savings provided it can service most of the loops in an application while still remaining relatively small in size as compared to the L1 cache. A majority of the performance degradation due to the filter cache can be mitigated by using an accurate parallel predictive path which directly accesses the L1 cache when needed [6].
- In the case where the application constitutes of tiny loops or the area available for a supplementary cache is small, loop caches become more favourable (can be made as small as just 32 instructions[2]). This is because there are no tag comparisons and a hit is guaranteed in a loop cache. Hence, tiny loops can be directly serviced without CPU stalls or performance degradation, unlike in a filter cache.
- Between different loop caching methods, a one-level hybrid solution seems most favourable due to relatively larger power savings from the tests in [2]; as well as the fact that it can both be pre-loaded with loop instructions and dynamically service tiny loops.
- In the case where adding a supplementary cache is undesired, the ROBTIC design approach can be considered. For an application with many loop basic blocks, it offers virtually no performance degradation but significant power savings. Moreover, the area added is small since it is limited to the control blocks needed to implement this solution. However, a major drawback to this approach is the fact that it is only for direct-mapped caches. This significantly reduces the scope for this solution.

(A relative comparison of caching techniques is shown in Table II.)

REFERENCES

[1] Segars, S.*Low power design techniques for microprocessors*, IEEE International Solid-State Circuits Conference, 2001.

[2] Ann Gordon-Ross, Susan Cotterell and Frank Vahid, *Tiny Instruction Caches For Low Power Embedded Systems*, ACM Transactions on Embedded Computing Systems, Vol. 2, No. 4, November 2003, Pages 449–481.

[3] K. Vivekanandarajah and T. Srikanthan, *Custom instruction filter cache synthesis for low-power embedded systems*, 16th IEEE International Workshop on Rapid System Prototyping (RSP'05), Montreal, Que., 2005, pp. 151-157.

[4] J. Gu and H. Guo and P. Li, *ROBTIC: An On-chip Instruction Cache Design for Low Power Embedded Systems*, 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications.

[5] Kin J. Gupta M. and Mangione-Smith W.H. *Filtering memory references to increase energy efficiency*, Computers IEEE Transactions on Volume: 49 Issue: 1 pp. 1-15 Jan 2000.

[6] K. Vivekanandarajah T. Srikanthan and S. Bhattacharyya *Energy-delay efficient filter cache hierarchy using pattern prediction scheme*, IEE Proceedings - Computers and Digital Techniques Vol. 151.Issue 2 March 2004.

[7] C.-L. Yang and C.-H. Lee, *Hotspot cache: Joint temporal and spatial locality exploitation for icache energy reduction*, Proceedings of ISLPED'04, 2004, pp. 114–119.

[8] K. Ali, M. Aboelaze, and S. Datta, *Reducing energy in instruction caches by using multiple line buffers with prediction*, Lecture Notes in Computer Science, v 4759 LNCS., pp. 508–521, 2008.