Journal of Systems Architecture 54 (2008) 1155–1163

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Optimizing CAM-based instruction cache designs for low-power embedded systems

Juan L. Aragón [a,*], Alexander V. Veidenbaum [b]

[a] Department Ingenieria y Tecnología de Computadores, Universidad de Murcia, 30100 Murcia, Spain
[b] Department of Computer Science, University of California, Irvine, CA, USA

ARTICLE INFO

Article history:
Received 10 December 2007
Received in revised form 22 February 2008
Accepted 3 June 2008
Available online 11 June 2008

Keywords:
Embedded systems
Energy-efficient architectures
Fetch unit design

ABSTRACT

Energy consumption and power dissipation are important concerns in the design of embedded systems and they will become even more crucial with finer process geometry, higher frequencies, deeper pipelines and wider issue designs. In particular, the instruction cache consumes more energy than any other processor module, especially with commonly used highly associative CAM-based implementations.

Two energy-efficient approaches for highly associative CAM-based instruction cache designs are presented by means of using a *segmented wordline* and a predictor-based instruction fetch mechanism. The latter is based on the fact that not all instructions in a given I-cache fetch are used due to taken branches. The proposed *Fetch Mask Predictor* unit determines which instructions in a cache access will actually be used to avoid fetching any of the other instructions. Both proposed approaches are evaluated for an embedded 4-wide issue processor in 100 nm technology. Experimental results show average I-cache energy savings of 48% and overall processor energy savings of 19%.

1. Introduction

Power dissipation and energy consumption are major design concerns when facing the design of a new microprocessor in the high performance domain (server and desktop) and, more dramatically, in the embedded computing systems domain, especially in the case of battery-operated devices. Embedded processors increasingly use multiple instruction issue, higher frequencies and deeper pipelines to increase performance which leads to higher energy consumption due to the presence of additional resources and higher utilization of such resources. These processors use in-order instruction issue and often do not contain a floating point unit. As a result, I-, D-caches, and TLBs consume an increasing share of overall energy. For instance, I-cache energy consumption was reported to be 27% of the total energy in the StrongArm SA110 [15].

Current embedded processors typically use content-addressable-memory (CAM) tags in highly-associative data and instruction caches (e.g., 32-way set associative for the Intel StrongArm [15] and the XScale [4]; or 16-way set associative for the Transmeta Crusoe [12]). Highly-associative CAM-tag caches consume more energy than SRAM-tag implementations, increasing the energy share of caches [16,22,26]. First, the CAM-tag access consumes a lot of energy (5–10 times more than a traditional, same sized SRAM-tag array [5,25]). Next, the subsequent access into the SRAM data array also consumes a significant amount of energy. However, the higher energy cost is justified by efficient implementations of high associativity. This is a trend that is likely to become commonplace. But the increasing energy consumption is especially important for the I-cache, which is accessed almost every cycle, always “reading out” an entire cache line, even if only some of the instruction words are used. For example, a 32-byte cache line (containing eight 4-byte instructions) may be read out in a given cycle but only two instructions being fetched in that cycle. For these reasons, the energy consumption of both the I-cache and the fetch unit is a major design concern in current and next-generation low-power embedded processors.

Several techniques have been proposed to provide the ability to access just a portion of the entire cache line (e.g., subbanking [6,20] and divided wordlines [23]), which is particularly useful when accessing the D-cache to retrieve a single data word. However, the I-cache fetch is much wider and typically involves fetching an entire line. But, because of the high frequency of branches in applications, in particular taken branches, not all instructions in an I-cache line may actually be used.

The goal of this research is to explore the effect of these trends and propose an energy-efficient instruction cache design for future low-power embedded processors, taking advantage of the *unused* instructions in an I-cache line fetch. The proposed approach attempts to reduce the instruction fetch energy in two ways. First, we propose a modification for the data array organization to use a *segmented wordline* organization. This is similar to proposals for using *segmented bitlines* [6] and it saves energy by allowing the

* Corresponding author. Tel.: +34 968 398788; fax: +34 968 364151.
E-mail address: jlaragon@ditec.um.es (J.L. Aragón).

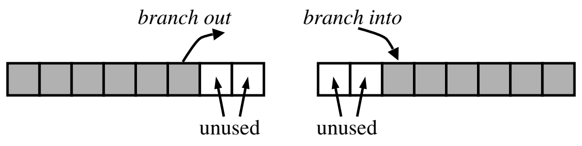

Fig. 1. *Branch out* and *branch into* cases.

fetch of the exact number of instruction words needed in a cycle as determined by the issue width *N* (typically 1, 2, or even 4 for future embedded designs). Second, it further reduces the number of instructions read by fetching only the "useful" instructions among the *N*-word segment.

When a processor fetches *N* instructions per cycle, not all *N* instructions may actually be issued. This happens in two cases which are illustrated in Fig. 1:

(1) One of the *N* instructions is a branch that is taken – a *branch out* case. All instructions in the *N*-word segment after the taken branch will not be used.
(2) An I-cache line contains a branch target, which is not at the beginning of the *N*-word segment – a *branch into* case. The instructions before the target will be unused.

In this work, a *Fetch Mask Predictor* unit is proposed to identify which of the instructions in each *N*-word segment are going to be used. Based on this information, only the useful part of the cache line is fetched in each clock cycle. Determining the unused words requires identifying the two branch cases described above. For the *branch into* case, a standard BTB (*Branch Target Buffer*) is used to obtain the word address of the first useful instruction in a fetch segment. For the *branch out* case, a different table is used to track whether the next line to be fetched contains a branch that *will* be taken, either conditional or unconditional. Finally, both cases can occur in the same cache line and we combine them to identify all instructions to be fetched in a given access. Once the useful instructions have been identified, the I-cache needs the ability to perform a partial access to a line. This may be achieved by using either a subbanked [6] or the proposed *segmented wordline* I-cache organization. Our proposed mechanisms will perform a partial access to the I-cache by supplying a *bit vector* to control either the subbanks to be activated or the pass transistors and drivers in case of using *segmented wordlines*.

The rest of the paper is organized as follows. Section 2 presents some related work. Section 3 motivates and analyzes the effect of the unused fetch instructions and Section 4 describes the proposed *Fetch Mask Predictor* unit. Section 5 details the experimental methodology and presents the energy-efficiency evaluation of the proposed mechanism. Finally, Section 6 summarizes the main conclusions of this research.

2. Background and related work

There have been many hardware and architectural proposals for reducing the energy consumption of TLBs and caches in general. Most of them are only applicable to SRAM-tag cache organizations. However, current and next-generation embedded processors implement highly-associative CAM-tag caches.

A general block diagram of one set of a CAM-tag cache is shown in Fig. 2 as outlined in [3]. In CAM-based caches, the CAM array is used to perform tag comparisons which significantly changes the organization, access mechanism and energy consumption of the cache. There are no longer separate tag and data RAM arrays that can be accessed in parallel (as for SRAM-tag caches). The CAM and data storage are a single unit. A domino-logic comparator in each tag location in the CAM drives a *match line*, which serves a *wordline* for the data array. Thus, there is no decoder for the data store. The bitlines, match lines, and buffers that drive control signals across the tag array are the main consumers of energy in the CAM-tag cache [26].

In order to allow a partial access to a cache line, several approaches have already been applied to D-caches, although they could also be applied to I-caches, supporting the idea of fetching just the desired instructions from the I-cache.

A *subbanked* cache organization divides the cache into subbanks [6,20] and activates only the required subbanks. A subbank consists of a number of consecutive bit columns of the data array. For the case of the I-cache, the subbank will be equal to the width of an individual instruction, typically 32-bit wide. While not directly applicable to CAM-based cache designs, this approach has the same general idea of selectively fetching only the desired instruction words.

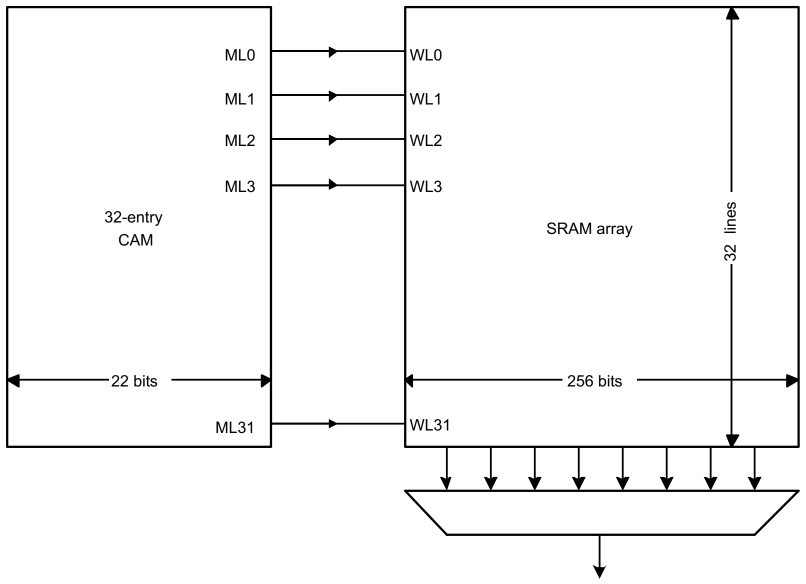

Fig. 2. Organization of a highly-associative CAM-based cache set.

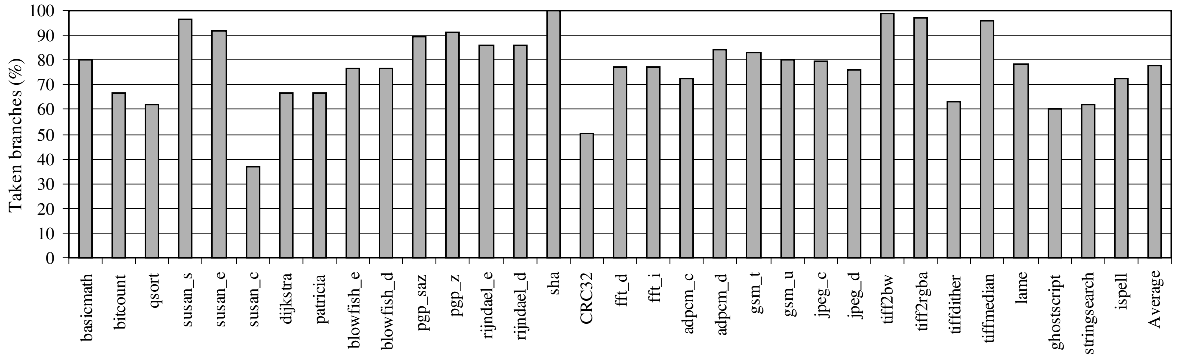

Fig. 3. Fraction of taken branches for the MiBench benchmark suite.

Similarly, the DWL (*Divided Wordline*) approach [23] can be used to reduce the length of a wordline and thus its capacitance. This design has been implemented in actual RAMs. It typically refers to a hierarchical address decoding and wordline driving. However, the need of a hierarchical decoding scheme makes DWL not applicable to CAM-based cache designs.

A related approach is *bitline segmentation* [6], which divides a bitline using pass transistors and allows sensing of only one of the segments. This approach isolates the sense amplifiers from all other bitline segments allowing for a more energy-efficient sensing. This is the design approach assumed for the *segmented wordline* mechanism we use in this paper.

A number of other organization techniques have also been proposed to reduce cache energy consumption although many of them are not applicable to CAM-based designs. Way-prediction predicts a cache way and accesses it as a direct-mapped organization [9,18,21]. A phased cache [8] separates tag and data array accesses into two phases. First, all the tags in a set are examined in parallel but no data access occurs. Next, if there is a hit, the data access is performed for the hit way. This reduces energy consumption but doubles a cache hit time. Way-memorization [13] is an alternative to way-prediction that stores precomputed in-cache links to next fetch locations aimed to bypass the I-cache tag lookup and thus, reducing tag array lookup energy. Other proposals place small buffers in front of caches to filter traffic into the cache. Examples include block buffers [1,20], multiple line buffers [6], the filter cache [11] and the victim cache [14]. In general, these proposals trade performance for power since they usually increase the cache hit time.

Finally, circuit techniques can also be used to reduce the power consumption of CAM structures. Pipelined CAM [17] breaks the match lines into pipelined stages. Since mismatches typically happen in the early stages, the pipelined CAM reduces power through halting additional searching operations in other pipeline stages. Techniques that use both circuit and organization techniques include serially accessed [5] and way-halting [24] caches. Serially accessed CAM caches may prolong the cache access time whereas way-halting caches need a specially designed CAM, which may not be easily available for embedded system designs.

3. Problem overview: unused fetched instructions analysis

In this section, we study the number of instructions that are *unused* in an I-cache line fetch due to taken branches in order to provide some insight on how these extra accesses may impact the energy consumption of a highly-associative I-cache organization as those typically implemented in embedded microprocessors such as the XScale.

The XScale microprocessor implements three pipelines as outlined in [4]. The main integer pipeline is seven stages long, memory operations follow an eight-stage pipeline, and finally, when operating in thumb mode an extra pipe stage is inserted after the last fetch stage in order to convert thumb instructions into ARM instructions. Both the I- and D-caches are 32 Kb, 32-way set associative caches with a 32-byte line size (i.e., *eight* 4-byte instructions per cache line). The replacement policy is a round-robin algorithm and caches also support the ability to lock code in at a line granularity. The I-cache has a fill buffer of two entries whereas the D-cache has a fill buffer of four entries. The D-cache supports *hit-under-miss* operation and there is an eight-entry coalescing writeback buffer. In addition, 32-entry fully associative TLBs that support multiple page sizes are provided for both caches. There is also a 128-entry, direct-mapped branch target buffer (BTB) for improving branch performance. The BTB stores the history of branch outcomes along with their targets.

In any case, for the analysis of the number of *unused* instructions in an I-cache line fetch due to taken branches, we must focus on the XScale's fetch policy operation [4]. Assuming a general *N*-issue width pipeline and a cache line of eight instructions, every *eight*/*N* cycles a whole 32-byte line is read out and placed in a *fetch buffer*. The purpose of the fetch buffer is to decouple the I-cache from the decode unit and the rest of the pipeline, as well as to provide a smooth flow of instructions to the decoders even at the presence of I-cache misses. Instructions from the *fetch buffer* are introduced into the pipeline at the issue width rate of *N*. The described XScale fetch policy works well from a performance-oriented point of view but it is not energy-efficient due to the high frequency of branches in applications, in particular taken branches, that interrupt the smooth flow of instructions through the pipeline.

Fig. 3 shows the fraction of taken branches for the evaluated MiBench benchmark suite [7], a publicly available suite designed to be representative for several embedded system domains. The baseline configuration is an *in-order* 4-wide embedded processor with a 4K-entry bimodal branch predictor and, as the XScale processor, it uses a 32 KB, 32-way I-cache with a 32-byte line size[1]. It can be observed that the average fraction of taken branches is about 78%, with some applications close to 99%. In addition, the average distance between *taken branches* has been measured to be 13 instructions. Note that this is a well-known application behavior also followed by other benchmark suites such as Spec2000 that is commonly used in the superscalar processor domain. As a reference point, the SpecINT2000 and SpecFP2000 have an average fraction of taken branches of 65% and 71%, respectively. Note however, that the fraction of taken branches is exacerbated in some embedded applications since they are very small, simple and loop-based programs.

This huge fraction of taken branches leads to a great amount of *unused* instructions in an I-cache line fetch. Therefore, for highly-associative CAM-based I-caches with long cache lines, it would

[1] See Section 5.1 for further details about processor configuration.

be more energy-efficient to read instructions from the I-cache at a smaller granularity instead of reading a whole line (typically eight instructions), given the high probability of some of the instructions never being used. This negative effect is illustrated in Fig. 4. If the second instruction of the cache line is a taken branch out of the line, the 4-instruction cache line results in 50% of unused instructions (2 out of 4) whereas the 8-instruction cache line results in 75% of unused instructions.

The next analysis quantifies the effect of fetching unnecessary instructions from the I-cache. Figs. 5 and 6 show the fraction of total unused instructions for the evaluated *in-order* 4-wide issue embedded processor respect to the ideal case where we were able to fetch the exact number of instructions needed. It can be observed that, on average, about 61% of all retrieved instructions from the I-cache are not introduced into the pipeline due to taken branches. This trend is followed by most benchmarks, going up to 67% of unused instructions for some of them such as CRC32 and gsm_t.

Note also that, in general, Figs. 3 and 5 point out a correlation between the fraction of taken branches and the fraction of unused instructions. However, we can find some benchmarks with a high fraction of unused instructions, such as CRC32 (67%), that do not have a high fraction of taken branches (just 50%). Contrarily, fft_d presents 56% of unused instructions but 77% of taken branches. The reason is the following. As illustrated in Figs. 1 and 4, a taken branch can be either at the beginning or at the end of the cache line, therefore, its negative effect highly depends on its position within the cache line (*branch out* case) as well as on the position of its target address (*branch into* case). In our case, the experimental results have shown that CRC32 presents most of taken branches at the opposite cache line extremes (*branch out* case at the beginning but *branch into* case at the end), contrarily fft_d presents its *branches out* at the end of the line but its *branches into* at the beginning of the line.

Summarizing, these results show that for CAM-based I-caches with long lines, the presence of taken branches interrupting the sequential flow of instructions is very significant and, consequently, there is a significant impact on the energy consumption of the I-cache. These results show the potential of the proposed *Fetch Mask Prediction* unit to reduce the energy consumption of the I-cache due to unused instructions for embedded processors.

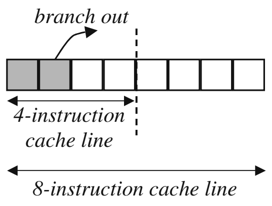

Fig. 4. Unused instructions for a 4-wide issue processor with either 4 or 8 instructions per cache line.

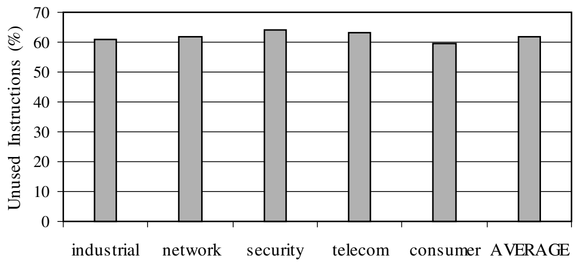

Fig. 6. Average unused instructions for each benchmark category.

4. Fetch Mask Predictor unit

The FMP (*Fetch Mask Predictor*) unit generates a control bit vector, whose length is equal to the number of instructions in a cache line, used to decide which instruction words within the line will be fetched in the *next* cycle. The bit vector controls either the subbanks to be activated or the pass transistors and drivers in case of using a *segmented wordline* I-cache organization. This allows for a partial line access to only the useful instructions and, therefore, it can save I-cache energy. In order to determine the bit mask for each fetch cycle, let us consider each of the two cases described in Section 1: *branching into* and *branching out* of a cache line.

For *branching into* the next line, it is only necessary to determine whether the current fetching line contains a branch instruction that is going to be taken. This information is provided by both the BTB (*Branch Target Buffer*) and the conditional branch predictor, as depicted in Fig. 7. Once a branch is predicted to be taken and the target address is known, its position in the next cache line is easily determined. For this case, only the instructions from the target position until the end of that cache line should be fetched. This information is stored into a *target_mask*.

For *branching out* of the next line, it is necessary to determine if the next I-cache line contains a branch instruction that is going to be taken. In that case, instructions from the branch position to the end of the line do not need to be fetched in the next cycle. To accomplish this, a *Mask Table* (MT) is used to identify those I-cache lines that contain a branch that will be predicted as taken for its next execution. The number of entries in the MT equals the number of lines in the I-cache. Each entry of the MT stores a binary-encoded mask, so each entry has $\log_2(issue\_width)$ bits. Every cycle, the MT is accessed to determine whether the next I-cache line con-

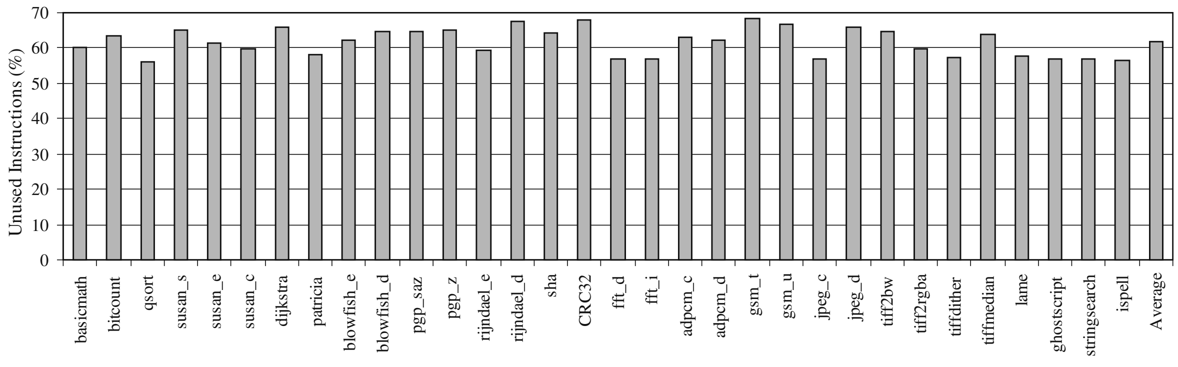

Fig. 5. Unused instructions for a 4-wide issue processor.

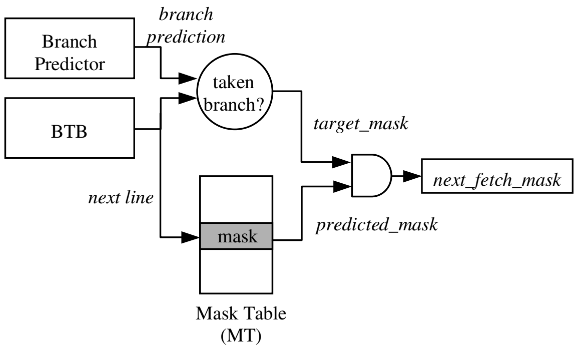

Fig. 7. The *Fetch Mask Predictor* unit.

tains a taken branch. This mask is called *predicted_mask*. When a branch is committed and prediction tables are updated, we can easily determine what the next prediction for that branch will be just by looking at the saturating counter. This information is used to update the MT in order to reflect if the branch will be predicted as taken the next time. Therefore, there are no extra accesses to the branch prediction tables, and thus, no additional power dissipation.

It is important to note that the proposed *FMP* unit is not degrading nor improving performance since it just anticipates the behavior of the underlying branch predictor to detect future taken branches, either correctly predicted or mispredicted. But the *FMP* unit does not change the behavior of the branch predictor and, therefore, the execution time remains the same. When a branch is executed, the corresponding MT entry will provide the correct mask for a *branch out* case, always in agreement with the prediction for that branch. In this way, the proposed *FMP* unit is not performing any additional predictions and, therefore, the *FMP* unit cannot miss. *FMP* just uses either the next prediction for a particular branch (n cycles before the next dynamic instance of the branch) to identify a *branch out* case or the information from the BTB to identify a *branch into* case. In other words, the proposed *FMP* is a deterministic unit that follows the branch predictor guesses, and there is no prediction accuracy associated to it.

In addition, neither the I-cache hit rate nor the accuracy of the branch predictor affects the energy savings provided by our proposal, only the amount of branches predicted as taken (as shown in previous Section). In case of branch misprediction, all necessary recovery actions will be done as usual and the corresponding MT entry will be reset to a mask of all 1's as explained below. Finally, it is possible for both *branching into* and *branching out* cases to occur within the same cache line. In this case, the *target_mask* and the *predicted_mask* need to be combined (ANDed).

To better understand the proposed design, let us consider the following example for two different I-cache lines:

*line*1: I_1, *branch*$_1$*_to_target*$_A$, I_2, I_3.
*line*2: I_4, *target*$_A$, *branch*$_2$, I_5.

where I_j ($j = 1,\ldots,5$) and *target*$_A$ are non-branching instructions. Assume that *line*1 is the line currently being fetched and the branch in *line*1 is predicted to be taken to *target*$_A$ in *line*2. For this *branch into* case, *target_mask* = 0111 for the second cache line. If *branch*$_2$ from *line*2 is also going to be predicted as taken[2], then only the first three instructions from *line*2 must be fetched. For this *branch out* case, the corresponding MT entry will provide a *predicted_mask* = 1110. When both masks are combined by a logical AND operation, the resulting mask is *next_fetch_mask* = 0110 which are the control bits that will be used for fetching just the required instructions from *line*2.

4.1. FMP implementation details

The *Fetch Mask Predictor* unit operates as follows:

(1) All the entries in the MT table are initialized to all 1's which means that all instructions in the line are going to be fetched.
(2) When an I-cache miss occurs and a line is replaced, the associated mask in the MT is reset to all 1's.
(3) In the fetch stage:
```
1. if (taken branch in current line) then
2.     use branch predictor/BTB to compute the
   target_mask;
3. else target_mask=all 1's;
4. predicted_mask=MT[next_line];
5. next_fetch_mask=target_mask AND predicted_
   mask;
6. if (next_fetch_mask==0) then
7.     next_fetch_mask=target_mask;
```
The last test above (line 6) is necessary for the following case:

*line*1: I_1, *branch*$_1$*_to_target*$_A$, I_2, I_3
*line*2: *branch*$_2$, *target*$_A$, I_4, I_5
If the first line is fetched and *branch*$_1$ is predicted as not taken, the program will continue with *line*2. If *branch*$_2$ is taken, then the MT will contain for *line*2 a *predicted_mask* = 1000. The next time *line*1 is fetched, if *branch*$_1$ is taken then *target_mask* = 0111 for *line*2. The combination of both masks results in zero, which is incorrect. According to steps 6 and 7 above, the *next_fetch_mask* used for fetching *line*2 will be equal to *target_mask*, which is the correct mask for this example.

(4) When updating the branch predictor at commit, also update the MT entry that contains the branch. If the branch that is being updated will be predicted as taken for the next execution, then disable all the bits from the position of the branch to the end of the line. Otherwise, set all the bits to 1.
(5) In case of a branch misprediction, reset the corresponding MT entry to all 1's. There is no other effect for our proposal in case of misprediction.

As for the effect on cycle time, note that determining the *next_fetch_mask* for cycle $i+1$ is a two-step process (see Fig. 7). In cycle i the BTB is used to create a *target_mask* for the next line. Then, the next line PC is used to access the MT to determine the *predicted_mask* and, finally, both masks are ANDed. Since the BTB and MT access are sequential, the *next_fetch_mask* may not be ready before the end of cycle i. If this is the case, despite the very small MT size (2 Kbits – see details at the end of Section 5.2), during the first half of cycle $i+1$, the CAM-tag comparison could be overlapped[3] with the access to the MT started in cycle i. By doing this, by the time the selected wordline is driven into the SRAM data

[2] We are assuming that its last execution changed the 2-bit saturated counter to the *predict-as-taken* state, updating the MT entry with *mask* = 1110.

[3] According to CACTI simulations of a 32 Kb, 32-way I-cache, approximately 50% of the whole cache access time is spent on the CAM-tag comparisons that select the wordline, whereas the other 50% is spent on driving the wordline into the SRAM data array, pulling down the corresponding bitlines, sensing them, and driving the output drivers for reading the line out of the cache.

array in cycle $i+1$, the *next_fetch_mask* will be ready for disabling the requested SRAM data subbanks.

4.2. Wordline segmentation

Wordline segmentation is aimed at reducing the I-cache energy by allowing the fetch of the exact number of instructions needed in a cycle as determined by the instruction issue-width N (instead of reading out the 32-byte cache line every four cycles, in case of a 2-issue width). As shown in Section 3, recall that due to the high frequency of taken branches it is more energy-efficient to fetch instructions at a smaller granularity instead of reading out a whole line and placing the eight instructions in a fetch buffer, given the high probability of some of the instructions never being used.

The implementation of *wordline segmentation* is similar to that of *bitline segmentation* [6], by using pass transistors in order to isolate the different N-word segments. This approach allows for a more energy-efficient sensing of only one of the segments since the effective load capacitance of the cache line is reduced (implementation details can be found in [6]).

5. Experimental results

5.1. Simulation methodology

To evaluate the *Fetch Mask Predictor* unit design, the Wattch v1.02 power simulator [2] was augmented with a model for the *FMP* unit (as will be shown in Section 5.2). In addition, since the original Wattch power model was based on CACTI version 1, the dynamic power model has been changed to the one from CACTI version 3.2 [19] in order to increase its accuracy.

Next-generation embedded processors modeled in this paper use multiple instruction issue and deep pipelines. In particular, the pipeline has been lengthened to 12 cycles (from fetch to commit). The simulated processor is an *in-order* 4-wide issue processor with four integer ALUs and one complex arithmetic unit for integer multiplication and division. The conditional branch predictor is a 4K-entry *bimodal* predictor whereas the BTB has 256 entries. The simulated architecture uses a 100 nm feature size and a 1.5 GHz clock frequency.

The simulated cache organization is based on that of the XScale processor [4]: 32KB, 32-way set associative data and instruction L1 caches with 32-byte lines. There is no L2 cache. The L1 cache access latency is 2 cycles and the main memory access latency is 150 cycles. The I- and D-TLBs are fully associative and have 32 entries.

Finally, the energy-efficiency of the proposed *FMP* unit has been evaluated using the MiBench benchmark suite [7]. The benchmarks are divided in six *categories* targeting different parts of the embedded systems market: Automotive and Industrial Control (basicmath, bitcount, susan (edges, corners and smoothing), Consumer Devices (jpeg encode and decode, lame, tiff2bw, tiff2rgba, tiffdither, tiffmedian, typeset), Office Automation (ghostscript, ispell, stringsearch), Networking (dijkstra, patricia), Security (blowfish encode and decode, pgp sign and verify, rijndael encode and decode, sha) and Telecommunications (CRC32, FFT direct and inverse, adpcm encode and decode, gsm encode and decode). All the benchmarks were compiled with the -O3 compiler flag and were simulated to completion using the ‘‘large” input set.

5.2. Power models for the Wordline Segmentation approach and the FMP unit

According to [10,19], the main sources of energy consumption in a typical SRAM-tag cache are: E_{decode}, E_{wordline}, E_{bitline}, E_{senseamp} and E_{tagarray}. For CAM-tag caches, since there is no decoder, the energy consumption is:

$$E_{\text{CAM-tag cache}} = E_{\text{wordline}} + E_{\text{bitline}} + E_{\text{senseamp}} + E_{\text{tagarray}} \quad (1)$$

To correctly model CAM-based set associative caches, the original Wattch's cache energy model was changed: first, excluding the decoder for the data array, and second, using Wattch's power model for CAM structures in order to model the CAM-tag match operation that selects the matching wordline.

Both the proposed *Wordline Segmentation* approach and the *FMP* unit energy savings rely on a partial access to a subset of the instruction words within the I-cache line. In Eq. (1), only terms E_{wordline}, E_{bitline} and E_{senseamp} are proportional to the number of bits fetched from the I-cache line, therefore, the energy consumed in a CAM-based cache access as a function of the number of fetched instructions in a given cycle is:

$$E_{\text{CAM-tag partial}} = \frac{i}{W} \times (E_{\text{wordline}} + E_{\text{bitline}} + E_{\text{senseamp}}) + E_{\text{tagarray}} \quad (2)$$

where W is the total number of instructions words in the line and $i \in [0, W]$ is the number of fetched instructions in a cycle.

The power model for the *Wordline Segmentation* approach assumes a partial access to the I-cache to fetch as many instructions as determined by the issue width N: therefore, $i = N$ in Eq. (2). On the other hand, the power model for the *FMP* unit assumes a partial access to the I-cache to fetch as many instructions as determined by the control bit vector: therefore, $i = \text{next\_fetch\_mask}$ in Eq. (2). Also note that, in general, the E_{wordline} term is very small (2%), whereas both E_{bitline} and E_{senseamp} terms account for approximately 80% of the 32 Kb, 32-way I-cache energy[4] as determined by CACTI v3.2.

Regarding the power overhead of the *FMP* unit, it uses three registers and the MT table described in Section 4. Note, however, that the size of the MT table is very small compared to the size of the I-cache. The MT has the same number of entries as I-cache lines and each MT entry has $\log_2(\textit{issue\_width})$ bits. For a 4-wide issue processor with a 32 Kb I-cache, the size of the MT is just 2 Kbits[5], which is 128 times smaller than the I-cache. For this example, the power dissipated by the MT has been measured to be about 0.8% of the power dissipated by the whole I-cache, which is negligible for the total processor power consumption. In any case, this power overhead has been modeled and incorporated in the evaluation of the *FMP* unit by using CACTI and Wattch's power models.

Finally, to better understand the effects of improving the energy-efficiency of the I-cache, Fig. 8 shows the maximum *a priori* power breakdown for each structure of the simulated 4-wide issue, *in-order* processor. It can be seen that the I-cache is responsible for about one third of the overall power dissipation, which is similar to results reported in [15]. It is important to note that this power distribution corresponds to the *maximum a priori* power of each structure. However, the actual *dynamic* power breakdown after executing an application is different since the I-cache is accessed more frequently than the D-cache. Our experimental results show that for the MiBench benchmark suite, on average, the I-cache accounts for 43% of the total processor power whereas the D-cache accounts for 23% of the overall power.

5.3. Energy-efficiency of the Fetch Mask Predictor unit

The baseline I-cache operation (similar to that of the XScale processor) is as follows supposing a N-wide issue processor. Every

[4] The other 18% cache energy is consumed in the CAM-tag match operation, the E_{tagarray} term, which is not reduced by our proposals.

[5] This I-cache has 1024 lines, each containing eight 32-bit instructions. So, the MT size is 1024$*$2 bits.

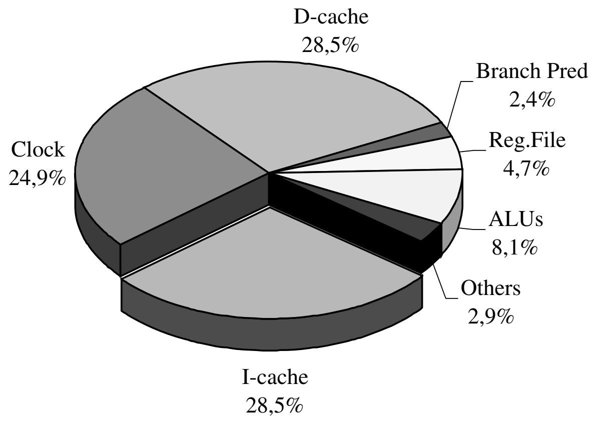

Fig. 8. Power breakdown for the simulated 4-wide issue, *in-order* processor.

eight/*N* cycles an access is performed to fetch a *whole* I-cache line (eight instructions). The desired CAM bank is first selected using 5 bits from the PC. PC high-order 22 bits are used for tag compare in the CAM array. An active CAM match line selects a data array line and the eight instructions (256 bits) are read out and placed into the instruction fetch buffer. In the baseline implementation, a multiplexor is used to select an aligned set of *N* instructions from the fetch buffer to be sent to instruction decoders.

However, the proposed *Wordline Segmentation* approach allows reading out of the cache only the desired *N*-word segment, as explained before, instead of the whole line. Additionally, the *Fetch Mask Predictor* unit identifies and enables access to a subset of the *N* words within the line whereas all other words are disabled.

Fig. 9 shows the energy savings over the baseline case for the I-cache when considering a 4-wide instruction fetch/issue. Fig. 10 shows the average energy savings for each benchmark category. On average, the *segmented wordline* approach saves 39% of the I-cache energy as compared to the baseline case. The fact that half of the data array is disabled in this case, and that the data array consumes approximately one half of the I-cache dynamic energy justifies the above results. More interesting is the fact that *FMP* provides additional energy savings, an average 48%, by disabling even more instructions that are not introduced into the pipeline. In addition, the improvement achieved by an *Oracle* mechanism is also evaluated. The *Oracle* mechanism identifies precisely all the instructions actually used within a cache line, therefore providing an upper bound on the benefits of the design proposed in this paper. It is interesting to note that *FMP* obtains energy savings very close to that of the *Oracle* experiment in all benchmarks (less than 4% on average as it can be seen in Fig. 10 and in Table 1). This shows the effectiveness of *FMP* in determining unused instructions within an I-cache line with an almost negligible hardware overhead.

In order to show the high correlation between the number of unused fetched instructions and the I-cache energy savings, Fig. 11 shows the fraction of unused fetched instructions for both the *segmented wordline* approach and the *FMP* unit. For comparison purposes, the *Oracle* mechanism identifies the exact number unused instructions, previously reported in Fig. 5. As expected, the fraction of unused instructions per cache line represents an upper bound on the I-cache energy savings that can be obtained with the proposed mechanisms.

Finally, Figs. 12 and 13 show the total processor energy savings whereas Table 2 shows the cumulative overall energy savings for each of the evaluated mechanisms. On average, the *segmented wordline* approach provides 15% of overall energy savings whereas *FMP* provides additional 4% overall energy savings. It is important to note that the I-cache dissipates a major fraction of overall processor power (around 30%) as shown in Fig. 8. However, as explained in Section 5.2, when taking into account the *dynamic* power breakdown after executing an application, the frequent accesses to the I-cache makes it a more power-hungry structure

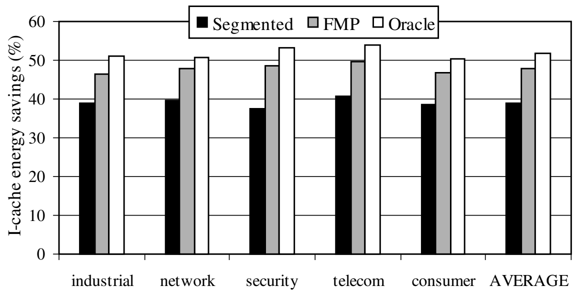

Fig. 10. Average I-cache energy savings for each benchmark category.

Table 1
Cumulative I-cache energy savings provided by the three mechanisms (*segmented*, *FMP* and *Oracle*) for each benchmark category

Category	Segmented (%)	FMP (%)	Oracle (%)
Industrial	39	8	4
Network	40	8	3
Security	37	11	4
Telecom	41	9	4
Consumer	39	8	4
Average	39	9	4

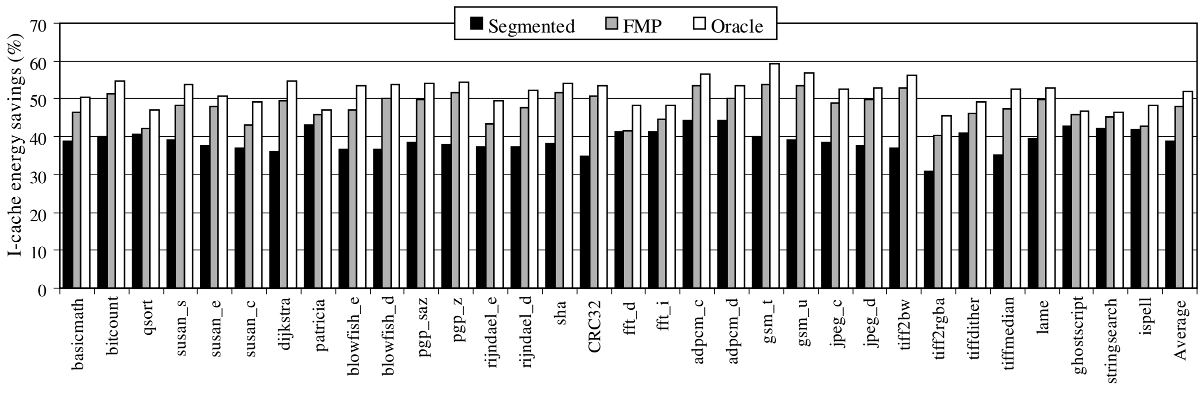

Fig. 9. I-cache energy savings for a 4-wide issue processor.

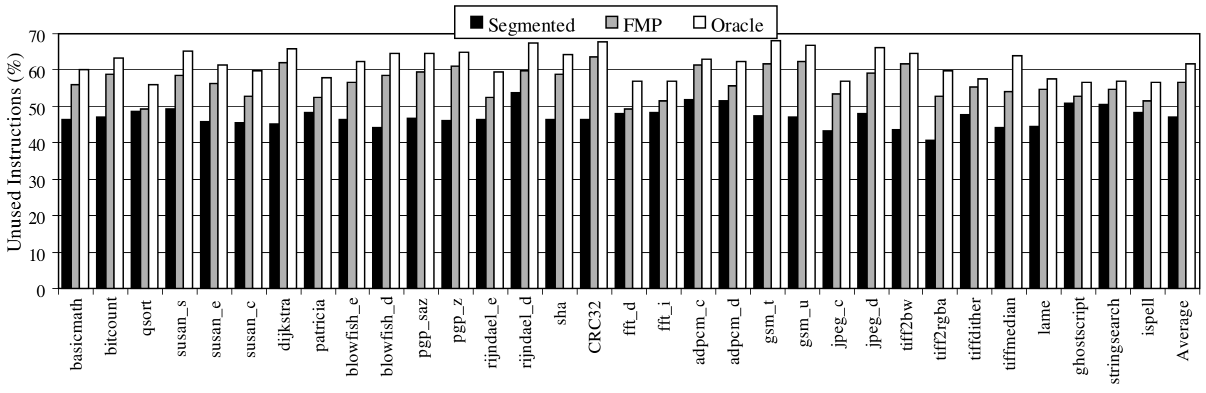

Fig. 11. Fraction of unused instructions for the proposed mechanisms.

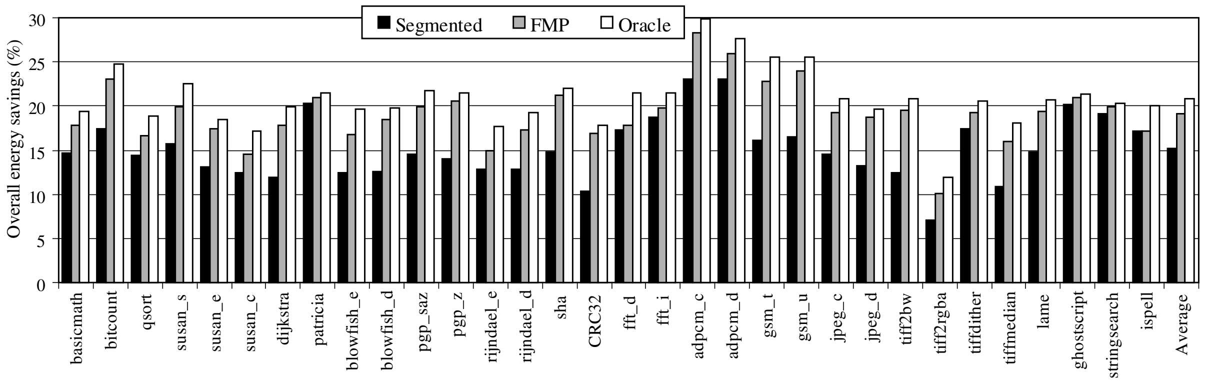

Fig. 12. Total processor energy savings for a 4-wide issue processor.

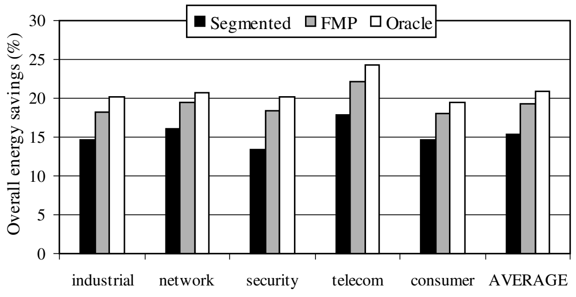

Fig. 13. Average total processor energy savings.

Table 2
Cumulative total processor energy savings for each mechanism

Category	Segmented (%)	FMP (%)	Oracle (%)
Industrial	15	4	2
Network	16	3	1
Security	13	5	2
Telecom	18	4	2
Consumer	15	3	1
Average	15	4	2

that accounts for 43% of overall processor power. For this reason, a mechanism such as *FMP* is able to provide such significant overall energy savings (an average of 19% energy savings).

In summary, the proposed *FMP* unit is able to provide significant I-cache and overall energy savings at a minimal hardware cost for next-generation embedded processors implementing highly associative CAM-based caches by not reading out from the data array of the I-cache those instructions that are not going to be used due to taken branches.

6. Conclusions

Modern embedded processors fetch, but may not use, many instructions from an I-cache line due to the high frequency of taken branches. Two energy-efficient mechanisms for a highly associative CAM-based caches have been presented by means of using a *segmented wordline* I-cache data array organization and a FMP (*Fetch Mask Prediction*) unit able to detect such unused instructions. The proposed *FMP* unit provides a bit vector to control the pass transistors and drivers in order to perform a partial access to the *segmented wordline*. Neither mechanism has an impact on performance.

A combination of both proposed approaches has been shown to reduce the energy consumption of the instruction cache by 48% and to provide overall processor energy savings of 19% for a 4-issue next generation embedded processor. Future embedded processors are likely to have even larger caches, branch predictors and branch target buffers. This will further improve the energy-efficiency of the proposed *Fetch Mask Predictor* unit.

Finally, the *FMP* unit is a mechanism orthogonal to other power-aware techniques such as fetch gating or way-prediction, and it can be used in conjunction with such techniques, therefore, providing additional energy savings.

Acknowledgements

This work has been jointly supported by the Spanish MEC and European Comission FEDER funds under Grants "Consolider Ingenio-2010 CSD2006-00046" and "TIN2006-15516-C04-03"; and by the Center for Embedded Computer Systems at the University of California, Irvine.

References

[1] I. Bahar, G. Albera, S. Manne, Power and performance trade-offs using various caching strategies, in: Proc. Int. Symp. on Low-Power Electronics and Design, 1998.
[2] D. Brooks, V. Tiwari, M. Martonosi, Wattch: a framework for architectural-level power analysis and optimizations, in: Proc. 27th Annual International Symposium on Computer Architecture, 2000.
[3] L.T. Clark, B. Choi, M. Wilkerson, Reducing translation lookaside buffer active power, in: Proc. 2003 International Symposium on Low Power Electronics and Design, ACM Press, 2003.
[4] L. Clark, E. Hoffman, J. Miller, M. Biyani, L. Liao, S. Strazdus, M. Morrow, K. Velarde, M. Yarch, An embedded 32-b microprocessor core for low-power and high-performance applications, IEEE Journal of Solid-State Circuits 36 (11) (2001) 1599–1608.
[5] A. Efthymio, J. Garside, An adaptive serial-parallel cam architecture for low-power cache blocks, in: Proc. Int. Symp. on Low-Power Electronics and Design, 2002.
[6] K. Ghose, M.B. Kamble, Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation, in: Proc. Int. Symp. on Low Power Electronics and Design, ACM Press, 1999.
[7] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, Mibench: A free, commercially representative embedded benchmark suite, in: Proc. IEEE 4th Annual Workshop on Workload Characterization, 2001.
[8] A. Hasegawa et al., Sh3: High code density, low power, IEEE Micro 15 (6) (1995) 11–19.
[9] K. Inoue, T. Ishihara, K. Murakami, Way-predicting set-associative cache for high performance and low energy consumption, in: Proc. Int. Symp. on Low Power Electronics and Design, 1999.
[10] M.B. Kamble, K. Ghose, Analytical energy dissipation models for low power caches, in: Proc. Int. Symp. on Low Power Electronics and Design, 1997.
[11] J. Kin, M. Gupta, W.H. Mangione-Smith, The filter cache: an energy efficient memory structure, in: Proc. Int. Symp. on Microarchitecture, 1997.
[12] A. Klaiber, The technology behind crusoe processors, in: Technical Report, Transmeta Corporation, 2000.
[13] A. Ma, M. Zhang, K. Asanovic, Way memoization to reduce fetch energy in instruction caches, in: Proc. ISCA Workshop on Complexity-Effective Design, 2001.
[14] G. Memik, G. Reinman, W.H. Mangione-Smith, Reducing energy and delay using efficient victim caches, in: Proc. Int. Symp. on Low Power Electronics and Design, 2003.
[15] J. Montanaro, R.T. Witek, K. Anne, A.J. Black, E.M. Cooper, D.W. Dobberpuhl, P.M. Donahue, J. Eno, G.W. Hoeppner, D. Kruckemyer, T.H. Lee, P.C.M. Lin, L. Madden, D. Murray, M.H. Pearce, S. Santhanam, K.J. Snyder, R. Stephany, S.C. Thierauf, A 160-MHz, 32-b, 0.5-w CMOS RISC microprocessor, IEEE Journal of Solid-State Circuits 31 (11) (1996) 1703–1714.
[16] D. Nicolaescu, A. Veidenbaum, A. Nicolau, Reducing power consumption for high-associativity data caches in embedded processors, in: Proc. Int. Conf. on Design, Automation and Test in Europe (DATE'03), 2003.
[17] K. Pagiamtzis, A. Sheikholeslami, A low-power content-addressable memory (CAM) using pipelined hierarchical search scheme, IEEE Journal of Solid-State Circuits 39 (9) (2004) 1512–1519.
[18] M.D. Powell, A. Agarwal, T. Vijaykumar, B. Falsafi, , K. Roy, Reducing set-associative cache energy via way-prediction and selective direct-mapping, in: Proc. Int. Symp. on Microarchitecture, 2001.
[19] P. Shivakumar, N.P. Jouppi, Cacti 3.0: an integrated cache timing, power, and area model, Technical Report 2001/2, Digital Werstern Research Lab, 2001.
[20] C. Su, A. Despain, Cache design tradeoffs for power and performance optimization: a case study, in: Proc. Int. Symp. on Low Power Design, 1995.
[21] W. Tang, A.V. Veidenbaum, A. Nicolau, R. Gupta, Integrated i-cache way predictor and branch target buffer to reduce energy consumption, in: Proc. Int. Symp. on High Performance Computing, Springer LNCS 2327, 2002.
[22] A.V. Veidenbaum, D. Nicolaescu, Low energy, highly-associative cache design for embedded processors, in: Proc. IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors (ICCD), 2004.
[23] M. Yoshimito, K. Anami, H. Shinohara, T. Yoshihara, H. Takagi, S. Nagao, S. Kayano, T. Nakano, A divided word-line structure in the static ram and its application to a 64k full CMOS RAM, IEEE Journal of the Solid-State Circuits SC-18 (1983) 479–485.
[24] C. Zhang, F. Vahid, J. Yang, W. Najjar, A way-halting cache for low-energy high-performance systems, ACM Transactions on Architecture and Code Optimization (TACO) 2 (1) (2005) 34–54.
[25] C. Zhang, A low power highly associative cache for embedded systems, in: Proc. IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors (ICCD), 2006.
[26] M. Zhang, K. Asanovic, Highly-associative caches for low-power processors, in: Proc. Kool Chips Workshop, 33rd Int. Symp. on Microarchitecture, 2000.

Juan L. Aragón received his M.S. degree in Computer Science in 1996 and his Ph.D. degree in Computer Engineering in 2003, both from the Universidad de Murcia, Spain, followed by a 1-year postdoctoral stay as a Visiting Assistant Professor and Researcher in the Computer Science Department at the University of California, Irvine. In 1999 he joined the Computer Engineering Department at the Universidad de Murcia, where he currently is an Associate Professor. His research interests are focused on CMP architectures, processor microarchitecture, and energy-efficient and reliable systems. He is a member of the IEEE Computer Society.

Alexander V. Veidenbaum received a Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 1985. He is a Professor in the Department of Computer Science at the University of California, Irvine. His research interests include high-performance and parallel architectures, low-power design, and compilation.