# Tag Overflow Buffering: An Energy-Efficient Cache Architecture

Mirko Loghi, Paolo Azzoni, Massimo Poncino



## To cite this version:

Mirko Loghi, Paolo Azzoni, Massimo Poncino. Tag Overflow Buffering: An Energy-Efficient Cache Architecture. DATE'05, Mar 2005, Munich, Germany. pp.520-525. hal-00181563

## HAL Id: hal-00181563
## https://hal.science/hal-00181563

Submitted on 24 Oct 2007

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Tag Overflow Buffering: An Energy-Efficient Cache Architecture

Mirko Loghi
Dipartimento di Informatica
Università di Verona
37134 Verona, Italy
loghi@sci.univr.it

Paolo Azzoni
Dipartimento di Informatica
Università di Verona
37134 Verona, Italy
azzoni@sci.univr.it

Massimo Poncino
Dip. di Automatica e Informatica
Politecnico di Torino
10129 Torino, Italy
massimo.poncino@polito.it

## Abstract

*We propose a novel energy-efficient memory architecture which relies on the use of cache with a reduced number of tag bits. The idea behind the proposed architecture is based on moving a large number of the tag bits from the cache into an external register (*Tag Overflow Buffer*) that identifies the current locality of the memory references; additional hardware allows to dynamically update the value of the reference locality contained in the buffer. Energy efficiency is achieved by using, for most of the memory accesses, a reduced-tag cache.*

*This architecture is minimally intrusive for existing designs, since it assumes the use of a regular cache, and does not require any special circuitry internal to the cache such as row or column activation mechanisms. Average energy savings are 51% on tag energy, corresponding to about 20% saving on total cache energy, measured on a set of typical embedded applications.*

## 1. Introduction

Unlike their general-purpose counterparts, embedded systems typically execute a well-defined application mix, which results in a higher predictability of the execution and memory access patterns. This fact offers additional opportunities for optimizing the performance or the energy consumption of the system, by allowing powerful *application-specific* optimizations that have been particularly successful in the design of the memory hierarchy (see [2, 3] for a survey).
Such type of optimizations is also suitable for caches, in spite of their natural adaptation to different workloads, and thus, to different application mixes. One of the most natural way of exploiting the predictability of a given application mix in a cache is that of constructively using the high locality of its memory references. As experimentally observed in several works, typical embedded applications can be decomposed into a few localities, each one accessing a small number of memory regions. In the context of a cache, this translates to a limited number of tags which could potentially be used for comparison. The most interesting property of these localities is their size; experiments show that, regardless of how many localities an application may contain, each locality span a limited range of addresses, which very seldom exceeds the size of a few tens of cache lines. This translates into the use of no more than 5-6 tag bits for most memory accesses.
Using a reduced number of tags in a cache, however, may result in incorrectly classifying an actual miss as a hit, since matching two values will in general require the comparison of the full tags. This false positive is called *false hit*, which cannot obviously be allowed in the execution of a program. At the same time, however, this solution seems to be quite appealing, since, the area and energy consumption of the tag portion can be sizable, especially for relatively small caches, like those typically used in embedded systems.
The use of partial tags in a cache has already been proposed in the literature [11, 12, 13, 14, 15]; most solution, however, require complex cache arrangements (such as the circuitry for selectively activating or disactivating columns of the cell array), or even external hardware.
In this work, we propose an energy-efficient cache structure which relies on the partial tag mechanism and a minimal amount of additional hardware external to the cache. The scheme is based on bringing most of the tag bits *outside the cache*, into a register which works as a sort of level-0 cache, and identifies the current locality. On a memory access, this register is first checked against the most significant bits of the address to determine whether we are inside the current locality or not. On a hit, the partial-tag cache is accessed normally (yet with a small cost); on a miss, the normal miss procedure is followed, with minor modifications.
Two are the main strengths of the proposed scheme. First, it uses a fixed number of tag bits in the cache; thus, it does not require any activation circuitry for changing the number of active tags, thus allowing to use a regular, yet smaller cache. Second, it achieves a tag energy reduction comparable to other schemes, yet with a much smaller hardware overhead (one register, two comparators, and a 3 bit-counter in the most complex configuration).
The proposed architecture allows to save 51% on tag energy (corresponding to a 20% saving on total cache energy) on average, measured on a set of typical embedded applications [16].

## 2. Motivation

Tags may take a significant portion of cache area and power, especially for relatively small caches, as those typically used in resource-constrained embedded systems. Their contribution is approximately proportional to the rel-

Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)
1530-1591/05 $ 20.00 IEEE
IEEE COMPUTER SOCIETY

ative weight of the tags with respect to the total number of bits in a cache line. For instance, for a 4KB cache with a line size of 4 bytes and 32-bit address, tags take 20 bits, which roughly translates to $20/52 = 38.4\%$ of total cache power.

This figures can even be much larger in the case virtual caches (i.e., where the core accesses the cache using virtual addresses) in 64-bit address spaces, although the latter are not very common in the embedded systems domain. In these cases, all processes see the entire virtual address space and the cache must in principle store the entire tags.

The relative weight of tags in caches motivates the use of techniques that reduce the size of the tags. Experimental evidence, on the other hand, shows that it is actually possible to reduce them. Typical embedded applications, in fact, due to their high locality of execution, tend to access a small number of memory regions, which would require a limited number of tags for identification. As a matter of fact, previous works [12, 14] have shown that using as few as 5–6 tag bits for address matching provides hit rates that are almost identical to full tag matching. This is confirmed by our analysis on a different platform and using different applications, as shown in Figure 1.

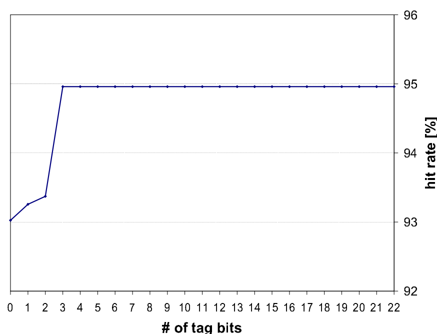


**Figure 1. Hit Ratio vs. Number of Tag Bits.**

The plot, referred to one of the PowerStone benchmarks [16], shows that four bits suffice to achieve the same hit rate of a full-tag comparison. Similar plots have been observed for the other benchmarks. Notice also that even using a zero-tag comparison would provide in this case a hit rate that is only marginally degraded. However, how it will be shown later, other issues than just hit rate must be considered in deciding the optimal number of tag bits to be considered.

## 3. Previous Work

The vast literature on application-specific memory optimizations only marginally involves caches, mainly because the terms “application-specific” and “cache” have historically been intrinsically contrasting terms. More recently, many authors have begun to use the high predictability of memory access patterns of embedded systems for cache optimization.

One class of solutions leverage a configurable cache structure in which line sizes, associativity, and other features can be dynamically reconfigured at runtime, upon detection of a decrease in performance, such as an increased miss rate ([4, 5, 6, 7, 8]).

The technique proposed by Villa et al. exploits the dominance of 0's in cached values by dynamically compressing zeroed data in the cache [9], resulting in lower energy thanks to reduced traffic. Givargis [10] proposes an application-specific cache indexing mechanism based on bit selection. Although the approach is meant for reduction of miss rate, energy efficiency is achieved as a byproduct.

A totally different class of approaches targets the optimization of the consumption of tag memory. In these approaches, power reduction is achieved through the reduction of the number of tag bits to be used for the comparisons; yet they differ in the choice of what and how many tag bits must actually be considered. The knowledge of the memory access pattern of the application may then be used to make this choice more effective.

Comparing addresses using a reduced number of tags, however, may result in incorrectly classifying an actual miss as a hit. We call this situation a *false hit*, which cannot obviously be tolerated because it does alter the correct behavior of a program.

For this reason, matching with reduced tags must be used carefully. For instance, some authors have used it where the cost of a misprediction is not critical, such as in branch prediction engines [11, 12]. The solutions proposed in [13, 14] deal instead with false hits by combining a detection scheme with a two-level comparison. On detection of a false hit, the remaining tag bits are enabled and used to complete the comparison. These solutions are indeed effective, but require a re-design of the caches, in order to accommodate the activation/deactivation mechanisms.

A completely different solution is proposed by Petrov and Orailoglu [15], in which addresses belonging to different localities of the application are *encoded* into a minimal-length tag. Encoding is implemented by an external block, which can be programmable. This solution, besides needing a relatively complex encoder, requires the capability of selectively disabling a sparse subset of the tag bits in the cache, since they can be dynamically adapted to the program locality.

Our approach is closer to [15], in that it is a combination of an architectural arrangement (the encoder) and a re-organization of the cache structure. However, it substantially differs in several aspects. First, it does not require any custom modification to the cache structure; no shutdown or selective activation circuitry whatsoever is required. We rather use a regular cache, yet with a reduced number of tags which is predefined up-front. Second, the complexity of the extra logic required by our solution is almost negligible.

Since our approach does not require modification of the cache structure, it might seems comparable to purely architectural techniques, such as the method proposed by of Bellas et al. [17], in which a small L0 instruction cache is employed to store only the most frequently executed basic blocks. Our method, however, sensibly differs from this approach because it does not increase the depth of the memory hierarchy, and therefore, the additional register does not work as a filter of cache accesses;

## 4. Reduced-Tag Cache Architecture

Let $n$ be the number of bits of the address, split into $b$ bits of byte offset, $s$ index bits, and $t$ tag bits. The cache will thus consist of $S = 2^s$ lines, each one with size $L = 2^b$ bytes.

The proposed architecture is based on using a cache with $k < t$ tag bits; this implies that, of the $t$ tag bits of the address, only $k$ bits are actually fed to the cache. The actual value of $k$ is determined up-front by profiling of the application mix during the software development phase, using a criterion which will be discussed later.

The remaining $t-k$ bits represent the locality around which the application is supposed to evolve for a large number of memory accesses. This implies that, except when locality changes, these $t-k$ bits do not change. The idea is thus to store this value and use it as a reference one, against which each memory access is compared. We called this locality value *tag overflow value*, and *tag overflow buffer (TOB)* the


IEEE COMPUTER SOCIETY

register used to store it; the name emphasizes that fact that the TOB represents an extension of the cache tag array. In practice, since we assume that for most of the cache lines these tag bits are identical, it is more energy efficient to take it out from the cache.

### 4.1. TOB-Based Architecture

Figure 2 shows a block diagram of the reduced-tag architecture. The shaded box represents the block which implements the TOB-based operations.
The figure emphasizes the fact that the TOB does not filter the accesses to the cache; rather, the TOB and the cache are always accessed in parallel at each memory reference. The outcome of the TOB lookup tells whether an access to the reduced-tag cache is feasible.

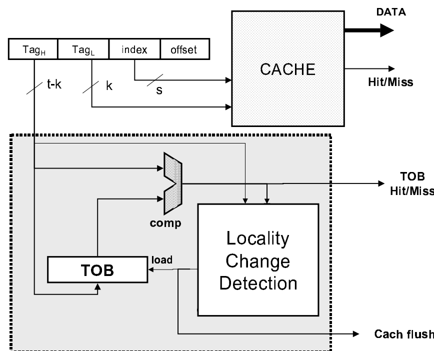


**Figure 2. Dynamic TOB-Based Architecture.**

The functional operations are as follows. On a memory reference, the $t - k$ most significant bits of the address are fed to the TOB. If the two values match (a *TOB hit*), then we can safely access the reduced-tag cache without worrying about false hits. Clearly, the lookup in the cache may result in a hit or miss as any access to a regular cache. In case of cache miss, the missed line would be replaced using some replacement strategy.
A TOB miss, conversely, regardless of the possible cache outcome, results in an equivalent cache miss. In fact, a TOB miss corresponds to a change in locality; a corresponding hit in the cache would result in a false hit, since the full tag does not actually match. There is however one important difference with respect to a regular cache miss: on a TOB miss, we do not fetch the corresponding line into the cache, that is, we do not replace the missed line in the cache. In fact, since we are referring to a single locality value, we regard a change in locality (a TOB miss) as a "sporadic" event. Therefore, the data contained in the cache are not modified. In other words, the replacement of a cache line occurs only within the context of the pre-defined locality value.
From the above discussion, it clearly emerges that a key issue for the efficient operations of this scheme is the choice of value of the TOB. The most reasonable choice, at a slight expense of hardware complexity, is to allow the TOB value to change and to *dynamically* adapt to possible changes in locality. This is the task of the block labeled *"Locality Change Detection"*, which, based on the observation of both the address and TOB miss output decides whether or not to enable the loading of a new locality value (i.e., the $t - k$ bits of the current address). When this occurs, the cache must be flushed, since all the values it contains actually refer to the previous locality. The functional operations of the dynamic architecture are summarized in Table 1.
Concerning the actual implementation of the locality change detection, we opted for a very simple realization,

| *TOB* | *Cache* | *Description* |
|---|---|---|
| H | H | Regular cache hit, no replacement |
| H | M | Regular cache miss, with replacement |
| M | – | If a change in locality is detected cache flush; otherwise, regular cache miss, without replacement. |

**Table 1. Summary of TOB-Based Operations.**

based on the sole observation of the TOB miss output. Specifically, a change in locality is determined as the number of consecutive TOB misses exceeding a given threshold. The value of the threshold may be critical, since we want to avoid an excessive number of locality switches. An explorative analysis has experimentally shown that a threshold of 7 suffices to track the actual locality changes.

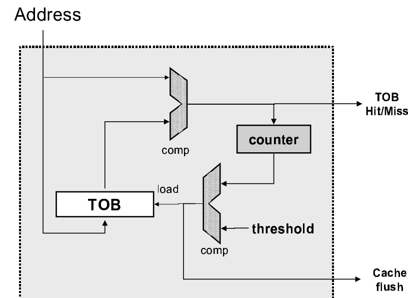


**Figure 3. Detailed TOB Implementation.**

Although more sophisticated schemes do exist for locality detection (e.g., [18]), the simple scheme of Figure 3 works reasonably fine, and meets the tight constraints of embedded systems.
It is important to emphasize that that the TOB-based architecture exhibits a tradeoff between the access to a reduced-tag, energy-efficient cache and an increase of the miss rate. In fact, a TOB miss will always be considered as a cache miss, regardless of what the result of a cache lookup would be. Clearly, the overall scheme is advantageous, energy-wise, if (i) the TOB miss rate is very low, and (ii) the access to the reduced cache (i.e., the common case) is more energy-efficient than the conventional cache. Section 4.2 discusses an analytical solution to this tradeoff.

### 4.2. Choosing the Optimal Tag Size

In the above discussion we have assumed that the value of $k$ is given; however, its choice is actually the most critical design issue for making the TOB architecture effective. In the following, we propose a derivation of the optimal value of $k$, based on an analytical formulation of the total memory energy $E_{mem}$ as a function of $k$.
Let $E_{cache}(k)$ the energy consumed by a single cache access. For simplicity, in this discussion we do not distinguish between read and write energy. $E_{cache}(k)$ is *monotonically increasing* with respect to $k$; for $k = 0$ (i.e., no tag) $E_{cache}$ is the energy of the data array only, whereas for $k = t$ $E_{cache}$ is the energy of a full-tag cache. To decouple the dependence of $E_{cache}$ versus $k$, we can express it as the sum of a constant term (the energy per access in the data array) plus the tag contribution, that is, $E_{cache}(k) = E_{data} + E_{tag}(k)$.
As discussed in Section 4, the use of the TOB trades off a reduced energy access cost in the cache ($E_{cache}(k)$) for a slight increase in the overall miss rate (in correspondence of


IEEE COMPUTER SOCIETY

TOB misses). Clearly, the miss rate will also depend on $k$: as $k$ increases and tends to $t$, the miss rate will tend to reach the miss rate value of the full-tag cache. In other terms, the miss rate is *monotonically decreasing* with respect to $k$. We denote the miss rate with $MR(k)$.
The total energy of memory accesses is thus given by:

$$E_{tot}(k) = E_{cache}(k) + MR(k) \cdot E_{miss} \quad (1)$$

where $E_{miss}$ denotes the energy cost of a miss penalty, and can be assumed as a constant. Equation 1 consists of the sum of two quantities with opposite behavior with respect to $k$. Therefore, there must be a value $k_{opt}$ of $k$ which minimizes the expression for $E_{tot}$, which is used as a value for $k$ in the architecture of Figure 2.
Figure 4 shows the plots of the quantities of Equation 1 for the same benchmark used for Figure 1, for a value of miss penalty of 5 (i.e., $E_{miss}/E_{cache} = 5$). This value, although apparently small, is typical of systems-on-chip architectures, where even the background memory is on-chip, connected through a fast interconnect.
Notice how $MR(k)$ (the middle curve) is not actually monotonically decreasing up to the value $k$ that saturates the miss rate ($k = 3$, from Figure 1). Notice also the linear behavior of $E_{cache}$.

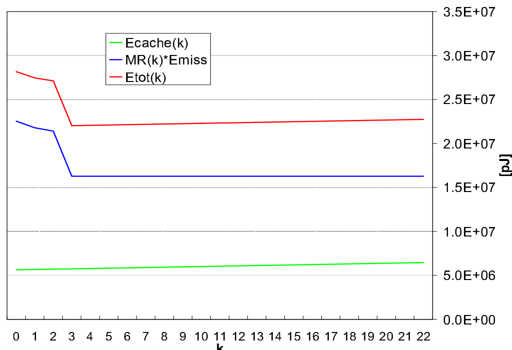


**Figure 4. Energy vs. Number of Tag Bits.**

In general, the energy is maximally reduced by applying Equation 1 for each individual application, in order to determine an optimal value of $k$ for each application. The application-specific nature of the architecture could be partially relaxed, however, by computing an "average" value of $k_{opt}$, derived from the memory access characteristic of a set of applications, at the expense of reduced energy savings.

### 4.3. Architectural Refinement

Some applications, such as loop-intensive kernels (e.g., FFT), may be highly localized so that a single main locality does exist; in these cases, the dynamic locality detection mechanism is somehow wasted, since it will not detect any significant locality change. These situations, which can be identified during the profiling phase, a *static* TOB-based architecture may be as effective as the dynamic one, yet with a smaller hardware overhead. By static we mean that the value in the TOB does not change during system operations; this fixed value represents the most "typical" locality value of the application.
Although a static arrangement seems to be overly simplistic, it actually eliminates the possibility of a cache flush: in a static scheme, since no change in locality is allowed, a TOB miss always incurs only in the penalty of an ordinary cache miss. As Section 5 will show, some embedded applications may benefit from the use of a fixed locality value.
The static scheme simplifies the architecture of Figure 2 by eliminating the counter and the comparison with the threshold value, so that the additional hardware consists just of the TOB register and a comparator; besides reduced capacitance, switching activity is also much lower, since they switch only in correspondence of TOB misses.

### 4.4. Performance Issues

One important point to emphasize is that the above architecture do not affect the cycle time of the cache, since the TOB and the cache are accessed in parallel, and the former has a much shorter delay than the cache.
Rather, these schemes actually *improve* cache access time, as a byproduct. Since the tag has reduced size, any access to the cache (be it a miss or a hit) will require a shorter time (not just less energy) than accessing a full-tag cache.

## 5. Experimental Results

We have applied to proposed architecture to the PowerStone suite, a set of widely used embedded benchmarks, which includes typical multimedia, control, filtering, decoding, and communication kernels [16]. Memory reference traces have been derived after compilation and execution of the benchmarks on the Platune simulation platform [19]. Simulations of the architecture have been carried out using a in-house cache simulator, augmented with energy and performance models. The latter are based on an empirical model, derived from interpolation of data extracted from a memory generator by STMicroelectronics in a 0.13 $\mu$m technology. The model is parameterized with respect to the number of rows and column, and has been derived by least-mean square regression of a set of energy values obtained with the memory generator for different memory sizes. The intrinsic error of the energy and cycle time models, with respect to actual data, is below 2%.
Tables 2 and 3 show the energy saving results for a 1KB, direct-mapped cache with a 4-byte line size, for instruction and data cache, respectively. Reported data include tag energy, as well as total cache energy, with the relative savings, plus, for each benchmark, the corresponding value of $k_{opt}$.
The average tag energy savings are 51% for the I-cache and 38% for the D-cache; these figures translate into a 21% and a 16% for the total cache energy.
The first comment is related to the quantification of the savings. Since the TOB-based architecture targets the reduction of tag energy only, there is an upper bound to the achievable energy savings; in the best case, we can reduce tag energy to zero, and cache energy proportionally to the ratio $t/(t + l)$, where $t$ is the original number of tag bits, and $l$ the size of a line in bits. The cache configuration of Tables 1 and 2 corresponds to a $t = 22$ bit tag cache, with a $l = 32$ bit line; therefore, we can achieve no more than $22/(22 + 32) = 40\%$ saving on cache energy, neglecting the overhead. This bound, however, is simplistic and it is based on the assumption that *energy perfectly scales with respect to the width (in bits) of the cache*; the actual bound is in fact lower because of the design rules of the memory array, which tends to keep the aspect ratio as square as possible. This fact is reflected in the memory models we have used, and explains why the tag energy savings reported are smaller than the expected value; for instance, benchmarks that use only 1 bit of tag actually save less than the simple ratio $((22 - 21)/22)$. This is also important when comparing our results to those of other approaches which are based on less accurate models ([13, 14, 15].
The second observation is concerned with the limited effectiveness of the method for data streams. This is somehow

| Benchmark | $k_{opt}$ | Tag Energy (pJ) | | | Cache Energy (pJ) | | |
|---|---|---|---|---|---|---|---|
| | | Original | Optimized | Δ | Original | Optimized | Δ |
| *adpcm | 1 | 5,65e05 | 3,25e05 | 42,6% | 1.37e6 | 9.21e5 | 32.8% |
| bcnt | 2 | 9,76e3 | 5,43e3 | 44,4% | 2.34e5 | 1.89e5 | 19.4% |
| blit | 0 | 1,50e5 | 0 | 100% | 3.58e5 | 2.81e5 | 21.5% |
| compress | 3 | 1,05e6 | 6,08e5 | 41,9% | 2.51e6 | 2.05e6 | 18.4% |
| crc | 2 | 2,49e5 | 1,39e5 | 44,1% | 5.95e5 | 4.79e5 | 19.4% |
| *des | 1 | 9,69e5 | 5,60e5 | 42,3% | 2.34e6 | 1.74e6 | 25.7% |
| engine | 0 | 2,73e6 | 0 | 100% | 6.54e6 | 5.15e6 | 21.3% |
| fir | 2 | 1,17e5 | 6,50e4 | 44,4% | 2.81e5 | 2.26e5 | 19.4% |
| g3fax | 2 | 7,51e6 | 4,20e6 | 44,1% | 1.80e7 | 1.45e7 | 19.4% |
| jpeg | 3 | 3,08e7 | 1,79e7 | 41,8% | 7.37e7 | 6.02e7 | 18.4% |
| pocsag | 2 | 3,52e5 | 1,95e5 | 44,4% | 8.45e5 | 6.80e5 | 19.5% |
| qurt | 2 | 8,75e3 | 4,83e3 | 44,8% | 2.11e4 | 1.70e4 | 19.5% |
| ucbqsort | 2 | 1,70e6 | 9,41e5 | 44,5% | 4.09e6 | 3.29e6 | 19.4% |
| v42 | 3 | 1,94e7 | 1,12e7 | 42,3% | 4.68e7 | 3.81e7 | 18.5% |

**Table 2. Energy Results (1KB Direct-Mapped I-Cache, Line Size = 4B).**

| Benchmark | $k_{opt}$ | Tag Energy (pJ) | | | Cache Energy (pJ) | | |
|---|---|---|---|---|---|---|---|
| | | Original | Optimized | Δ | Original | Optimized | Δ |
| adpcm | 10 | 1,31e5 | 9,51e4 | 27,3% | 3.06e5 | 2.70e5 | 11.7% |
| bcnt | 0 | 4,45e3 | 0 | 100% | 1.05e4 | 7.23e3 | 31.6% |
| *blit | 0 | 5,91e4 | 0 | 100% | 1.41e5 | 5.88e4 | 58.5% |
| compress | 10 | 5,59e5 | 4,05e5 | 27,6% | 1.32e6 | 1.17e6 | 11.7% |
| crc | 10 | 2,18e4 | 1,58e4 | 27,5% | 5.12e4 | 4.52e4 | 11.7% |
| des | 10 | 2,17e5 | 1,56e5 | 27,9% | 5.16e5 | 4.56e5 | 11.7% |
| engine | 10 | 1,44e6 | 1,05e6 | 27,2% | 3.37e6 | 2.98e6 | 11.6% |
| fir | 10 | 4,00e4 | 2,91e4 | 27,3% | 9.38e4 | 8.28e4 | 11.7% |
| g3fax | 10 | 2,02e6 | 1,46e6 | 27,7% | 4.77e6 | 4.21e6 | 11.7% |
| jpeg | 10 | 1,04e7 | 7,53e6 | 27,5% | 2.44e7 | 2.16e7 | 11.7% |
| pocsag | 10 | 1,02e5 | 7,38e4 | 27,4% | 2.38e5 | 2.10e5 | 11.7% |
| qurt | 10 | 4,32e3 | 3,11e3 | 27,9% | 1.01e4 | 8.97e3 | 12.0% |
| ucbqsort | 10 | 4,33e5 | 3,15e5 | 27,3% | 1.01e6 | 8.95e5 | 11.6% |
| v42 | 10 | 5,68e6 | 4,11e6 | 27,6% | 1.34e7 | 1.19e7 | 11.7% |

**Table 3. Energy Results (1KB, Direct-Mapped D-Cache, Line Size = 4B).**

expected, since it is a well-known fact that data references exhibit smallesmaller locality than instruction ones.

Entries in the tables which are marked with an asterisk denote benchmarks for which the static scheme proved more effective than the dynamic one. For the I-cache, these cases (*adpcm* and *des*) correspond to the situation in which the locality detection hardware would force to use an unnecessarily large number of bits in the tag memory, to avoid a large number of cache flushes (402 for *adpcm* and 191 for *des*, using a 1-bit dynamic TOB).

For the D-cache, the saving for the *blit* benchmark actually exceeds the above mentioned upper bound. As a matter of fact, this is due to the cache pollution that occurs for the full-tag cache, while the use of a reduced tag avoids the loading of non useful data into the cache.

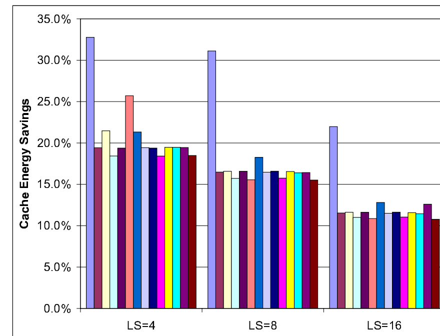


**Figure 5. Effect of Line Size (1 KB Direct-Mapped I-Cache).**

## 5.1. Sensitivity to Cache Parameters

The data of Tables 2 and 3 refer to a good cache configuration for the TOB scheme, since it has a large number of tags bits. In this section we report cache energy results for different cache configurations, in order to assess when and how much the proposed architecture can be reasonably used as an energy-efficient optimization.

Figure 5 shows how energy savings scales with respect to line size. Larger lines imply a reduced weight of the tags compared to the number of bits of a cache line. The plot confirms this, by showing a consistent decrease of cache energy savings (from an average 20% for a 4-byte line to an average 10% for a 16-byte line).

Figure 6 shows how energy savings scales with respect to cache size. Larger cache also imply a reduced weight of the tags because of the increased number of index bits. In addition, since the energy model depends on both the number of rows and columns of the cache, the increase in the number of rows (i.e, cache lines) partially offsets the decrease in the number of columns due to reduced tags. The plot shows a progressive decrease of the energy savings from 20% to about 10%.

Finally, Figure 7 shows how energy savings scales with respect to cache associativity. In principle, associativity should not affect the savings significantly, since doubling the number of ways should even reduce the number of index bits, and each way has its own set of tags. The plot confirms


IEEE COMPUTER SOCIETY

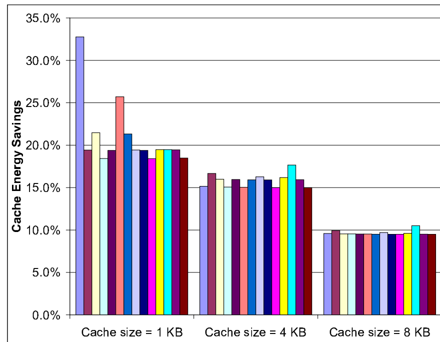


**Figure 6. Effect of Cache Size (Direct-Mapped I-Cache, Line Size = 4).**

this trend, showing that the effectiveness of the method is nearly unchanged for 2- and 4-way associative caches.

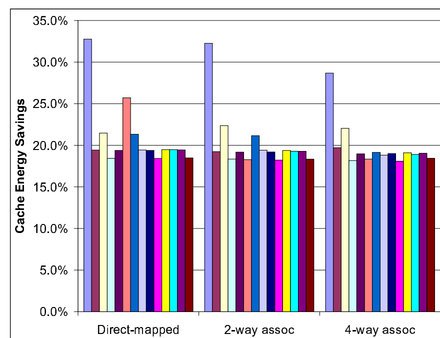


**Figure 7. Effect of Associativity (1KB I-Cache, Line Size = 4).**

From the above analysis we can conclude that the proposed technique is maximally effective in the case of small caches and with small line sizes, that is, the typical scenario of small-scale, resource-constrained embedded systems. Large caches, conversely, tend to rapidly nullify the benefits of the TOB-based architecture, since their large access cost reduces the "importance" of tag energy reduction.

### 5.2. Encoder Implementation

We have implemented the TOB encoder, synthesizing a RTL description of the architecture of Figure 2 using Synopsys DesignCompiler and a $0.13\mu m$ technology library by STMicroelectronics. An encoder with a 15-bit TOB consumes about 0.09 pJ per access.

This figure compare favorably to the energy per access of the cache. As a reference value, the energy per access a 16KB, direct-mapped cache with 16 bytes per line is 166.8 pJ. This makes the worst case overhead around 0.6%, and 0.0048% for the best case.

Concerning delay, the arrival time of the TOB hit output is 80 ps, and 140 ps for the cache flush output. As a reference figure, the access time for a 16KB cache (on a low-power version of the technology library) is 5ns.

## 6. Conclusions

Embedded applications which exhibit high locality may benefit from an architectural arrangement where most of the cache tag bits are moved out of the cache into a special device called *tag overflow buffer* (TOB).

The TOB-based scheme allows to use a reduced-tag cache on most memory references, thus providing a low-overhead and minimally intrusive technique to improve the energy consumption of the memory subsystem. The proposed technique is orthogonal to many similar cache optimization techniques, and provides tag energy savings of 51% on average, corresponding to a 20% average saving of total cache energy.

## References

[1] P. Panda, N. Dutt, *Memory Issues in Embedded SoC Optimization and Exploration*, Kluwer, 1999.

[2] A. Macii, L. Benini, M. Poncino, *Memory Design Techniques for Low-Energy Embedded Systems*, Kluwer Academic Publishers, 2002.

[3] W. Wolf, M. Kandemir, "Memory System Optimization of Embedded Software," *Proceedings of the IEEE*, Vol. 91, No. 1, pp. 165-182, January 2003.

[4] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, S. Dwarkadas, "Memory Hierarchy Reconfiguration for Energy and Performance in General Purpose Processor Architectures," *IEEE International Symposium on Microarchitecture*, pp. 245–257, Dec. 2000

[5] A. Malik, W. Moyer, D. Cermak, "A Low-Power Unified Cache Architecture Providing Power and Performance Flexibility," *ISLPED'00: International Symposium on Low Power Electronics and Design* pp. 241–243, July 2000.

[6] P. Petrov, A. Orailoglu, "Towards Effective Embedded Processors in Codesigns: Customizable Partitioned Caches", *CODES'01: IEEE Workshop on Hardware/Software Codesign*, pp. 79–84, Apr. 2001.

[7] C. Zhang, F. Vahid, R. Lysecky, "A Self-Tuning Cache Architecture for Embedded Systems," *DATE'04: Design, Automation and Test in Europe Conference*, pp. 142–147, February 2004.

[8] A. Gordon-Ross, F. Vahid, N. Dutt, "Automatic Tuning of Two-Level Caches to Embedded Applications," *DATE'04: Design, Automation and Test in Europe Conference*, pp. 208–213, February 2004.

[9] L. Villa, M. Zhang and K. Asanovic, "Dynamic Zero Compression for Cache Energy Reduction," *MICRO'33: International Symposium on Microarchitecture*, pp. 214–220, December 2000.

[10] T. Givargis, "Improved Indexing for Cache Miss Reduction in Embedded Systems," *DAC-40: 40th ACM/IEEE Design Automation Conference*, pp. 875–880, Jun. 2003.

[11] B. Fagin, "Partial Resolution in Branch Target Buffers," *IEEE Transactions on Computers*, Vol. 46, No. 10, Oct. 1997 pp. 1142–1145.

[12] B-S. Choi, D-I. Lee, "Cost-Effective Value Prediction Micro-Operation using Partial Tag and Narrow-Width Operands," *PACRIM'01: IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Aug. 2001, pp. 319–322.

[13] L. Liu, "Partial Address Directory for Cache Access", *IEEE Transactions on VLSI Systems*, Vol. 2, No. 2, June 1994, pp. 226–240.

[14] R. Min, Z. Xu, Y. Hu, W.-B. Jone, "Partial Tag Comparison: A New Technology for Power-Efficient Set-Associative Cache Designs *VLSID'04: 17th International Conference on VLSI Design*, pp. 183–188, Jan. 2004.

[15] P. Petrov, A. Orailoglu, "Data Cache Energy Minimizations Through Programmable Tag Size Matching to the Applications," *ISSS'01: International Symposium on System Synthesis*, Sept./Oct. 2001, pp. 113–117.

[16] A. Malik, B. Moyer, D. Cermak, "A Lower Power Unified Cache Architecture Providing Power and Performance Flexibility," *ISLPED'00: International Symposium on Low Power Electronics and Design*, 2000, pp. 241–243.

[17] N. Bellas, I. Hajj, C. Polychronopoulos, "Using Dynamic Cache Management Techniques to Reduce Energy in a High-Performance Processor," *IEEE Transactions on VLSI Systems*, Vol. 8, No. 6, pp. 693–708, December 2000.

[18] T. L. Johnson, M. C. Merten, W.-M. W. Hwu "Run-Time Spatial Locality Detection and Optimization," *30th ACM/IEEE International Symposium on Microarchitecture*, pp. 57–64, December 1997.

[19] T. Givargis, F. Vahid, "Platune: A Tuning Framework for System-on-a-Chip Platforms," *IEEE Transactions on Computer Aided Design*, Vol. 21, No. 11, Nov. 2002.



IEEE COMPUTER SOCIETY