

# Power Efficient Embedded Processor IP's through Application-Specific Tag Compression in Data Caches\*

Peter Petrov and Alex Orailoglu

Computer Science & Engineering Department  
University of California, San Diego  
(ppetrov,alex)@cs.ucsd.edu

## ABSTRACT

*In this paper, we present a methodology for power minimization by data cache tag compression. The set of tags being accessed by the major application loops is analyzed statically during compile time and an efficient and optimal compression scheme is proposed. Only a very limited number of tag bits are stored in the tag array for cache conflict identification, thus achieving a significant reduction in the number of active bitlines, sense amps, and comparator cells. The underlying hardware support for dynamically compressing the tags consists of a highly cost and power efficient programmable encoder, which lies outside the cache access path, thus not affecting the processor cycle time. A detailed VLSI implementation has been performed and a number of experimental results on a set of embedded applications and numerical kernels is reported. Energy dissipation decreases of up to 95% can be observed for the tag arrays, while significant energy reductions in the range of 10%-50% are observed when amortized across the overall cache subsystem.*

## 1. INTRODUCTION

Complex embedded systems are ever more prevalent in the modern electronics market. A large fraction of system functionality is typically mapped to a set of high-performance microprocessors resulting in shorter design cycles and flexible, cost efficient implementations. Yet the high performance processor solutions come at the price of highly increased power consumption. Energy dissipation is becoming a prominent characteristic for a large number of important applications, such as hand-held and wireless devices. Less energy dissipation leads not only to longer battery life, but also to larger die sizes. Consequently, overall product quality is highly dependent on techniques for minimizing system power consumption. These techniques can be applied on various design abstraction levels, from circuit level to system architecture.

The data cache subsystem is an important microarchitectural component serving to bridge the ever growing gap between memory access time and processor execution speed. Not only the increasing variance between processor speed and memory access time, but also application complexity constitute driving forces towards larger caches implemented on the same die as the microprocessor core. Both tag and data arrays are placed on the processor's die and typically account for a significant part of the transistor budget and hence the total power consumption [1].

The tag arrays are used to store a certain number of the most significant bits from the effective address in order to resolve data cache conflicts. Typically, the tag field from the effective address constitutes more than half of the entire address. The particular tag length depends on the cache organization (associativity, number of cache lines, and cache line size). Conceptually, the tags behave as keys

associated to certain memory regions and are used to distinguish each of these memory regions in the cache. Given the general assumption that the application can access arbitrary memory regions, the whole tag field from the effective address needs to be stored in the tag array and used for cache conflict identification.

Yet, as our experimental studies show, a large number of important embedded applications can be separated into a few major loops, with each loop accessing a small number of memory regions, thus generating a limited number of tags. The number of distinct tags generated from a particular application loop can be analyzed statically during compile time, given that the loop does not operate on dynamic data structures, a typical situation for embedded applications. Now that the limited set of tags being accessed by a particular application loop is identified, an efficient compression/encoding scheme can be applied, thus effectively reducing the number of tag bits that need to be stored in the tag arrays and consequently greatly reducing the power consumed in precharging/discharging bitlines in the tag array as well as reducing the energy dissipated in the corresponding sense amps and comparator cells.

In this paper, we present a methodology for optimally encoding the tags generated by the application loops, thus significantly reducing the number of tag array bitlines that need to be read for cache conflict identification. The first part of the methodology is a compile-time algorithm for finding within the set of tags the minimal number of bit positions that can be directly utilized for producing an optimal encoding; this application-specific information is provided to a programmable encoder. The second part of the proposed methodology consists of the hardware support for efficiently implementing the tag encoding scheme. The proposed hardware support consists of the aforementioned programmable encoder and the gating logic for the bitlines and the sense amps of the tag arrays. The highly cost effective encoder structure remains outside the cache access critical path. The encoder is software programmable and the application-specific setup is performed prior to entering the loop by writing to certain control registers.

In its underlying essence, the proposed methodology is an application-specific customization of the cache subsystem, which utilizes application knowledge to perform an efficient tag encoding, thus significantly reducing power consumption. The proposed technique is particularly applicable for high-performance embedded processors, for which the tags accessed by the application loops are fixed and can be easily identified during compile-time. The programmable hardware support provides the flexibility for software-controlled customization with no need of spinning new silicon. The fixed microarchitecture in turn preserves the significant benefits of high volume productions and enables the integration of the processor as a hard or soft IP block in SOC designs.

\*This work is supported by NSF Grant 0082325.

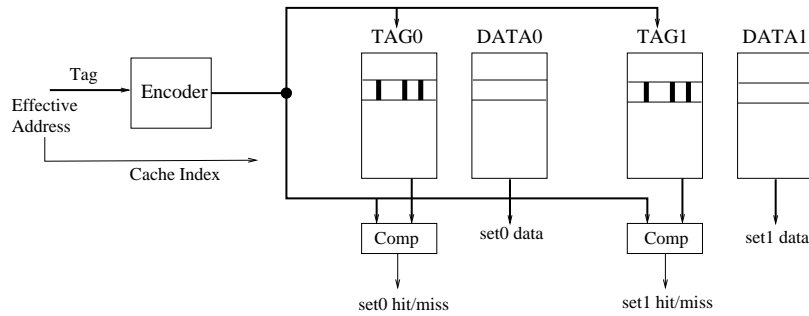


Figure 1: Tag encoding architecture

## 2. RELATED WORK

Circuit-level power minimization techniques have been the dominant approach in designing energy efficient designs so far [2, 3]. However, in recent years, architecture-level approaches have attained popularity due to their ability to eliminate redundancies on a higher, microarchitectural level, thus resulting in even larger power optimizations [4, 5]. In [6], a small and energy efficient L0 data cache has been introduced in order to reduce the power consumption of the memory hierarchy. The price paid is an increased miss rate and longer access time. A power optimization technique applied during behavioral synthesis for memory intensive applications has been presented in [7]. The behaviour of the memory access patterns is utilized to minimize the number of transitions on the address bus and decoder, thus reducing power consumption. In [5] an L0 instruction cache has been proposed with run-time techniques for accommodating only the frequently executed basic blocks. The small size of this cache translates directly to power consumption reductions. Speculative execution in modern high-end processors results in large instruction execution overhead. In [4] a technique for speculation control and pipeline gating has been presented for energy reduction in speculative processors. A technique for turning off associativity ways in a set-associative cache architecture has been proposed in [8]. The selection of associativity ways is performed by software and the authors illustrate the trade-off in performance vs. power reduction. A methodology for combining dynamic voltage scaling and dynamic power management was presented in [9]. By merging these two major algorithms for power optimizations, the paper significantly boosts the power reductions utilizing the advantages of both approaches. In [10] the authors propose a technique for gating parts of the cache lines, which contain zero values. A special hardware for gating the bitlines and sense amps is therein proposed. A new energy estimation framework for microprocessors has been proposed recently in [11]. The simulation environment employs a transition based power model and quickly achieves very precise power estimations.

## 3. METHODOLOGY OVERVIEW

### 3.1 General architecture

The proposed approach builds upon standard cache configurations. A two way associative cache organization with support for tag compression is shown in Figure 1, wherein the two tag arrays are being indexed simultaneously and the corresponding tags compared for cache hit/miss in both associativity ways. We propose instead of storing the entire tag, to store only the highly compressed values of the original tags for the application loops. The tag comparators operate upon the compressed tag values, which require significantly less number of bits. Consequently, a large number of bitlines and sense amps for all the tag arrays are disabled during loop

execution, thus significantly minimizing power. All tag arrays are looked up simultaneously and only on a cache hit is the corresponding data array read from a particular associativity way. This operational sequence for accessing tag and data arrays in set-associative organizations is frequently found in low power designs<sup>1</sup>.

The purpose of the encoder unit is to compute the compressed tags and to provide them for tag comparison or in the case of a cache miss for storing in the tag array. Since each tag from the set of tags accessed by a loop is encoded with a unique value, comparing the encoded values would be a correct identification for cache hit or miss. As the tag compression scheme is performed per application loop and thus the cache flushed before entering the loop, it can be bypassed for application loops for which it is not beneficial.

While flushing the cache before entering the major application loop may be thought to increase miss rates, the experimental results we have obtained show essentially no such effect, most probably because of the very limited data reuse across major application loops. Typically within one major loop, the application traverses several large data structures (arrays) and has access to some temporal/local data. The large data structures are rarely reused by subsequent loops since their size precludes their continued stay in the cache and their consequent availability in the next application loop.

### 3.2 Tag compression

Conceptually, our tag compression approach utilizes only a limited subset of tag bits and only their bitlines and sense amps in the tag arrays are enabled. The rest of the bitlines and their respective sense amps are disabled. The position of utilized tag bits is not necessarily restricted to a contiguous set of least or most significant bits, but can be scattered arbitrarily across the original tag field to ensure efficient and optimal solution. The number of utilized bitlines is typically in the range of 1-7 bits, compared to traditional tags, typically in the range of 20-23 bits. The same encoder unit provides compressed tags for all the tag arrays, thus increasing the benefits of the proposed power minimization technique proportionally to the number of associativity ways.

Figure 2a shows an illustrative set of tags accessed by a particular application loop. Within the proposed methodology, this original set of tags is identified by the compiler/linker and is provided to a static analysis algorithm, which complements the compile/link process, for identifying certain information concerning bit positions and relations amongst them. This application information is utilized dynamically by the special encoder circuit for efficiently computing the compressed tags. For the set presented in the example, containing only four tags, two bits suffice to provide a unique code for each of the tags, while more generally for a set of  $n$  tags,

<sup>1</sup>The alternative, but significantly more power consuming implementation, is to simultaneously read all data arrays and on cache hit select only the correct data, resulting in better access time.

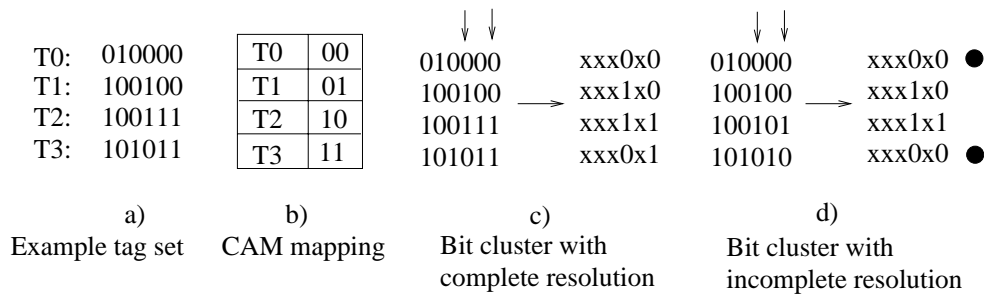


Figure 2: Tag encoding example

only  $\lceil \log_2 n \rceil$  bits are needed to compress efficiently the set of tag bits. A straightforward approach for compressing the given set of tags consists simply of associating to each of them a unique two-bit code. Unfortunately, this straightforward approach requires a CAM structure of the type shown in Figure 2b in which the tags need to be stored as keys with the corresponding encoded values associated to them. Increasing the number of tags covered comes at a direct, linear increase in the number of entries, hence requiring significant power consumption as this approach utilizes only information regarding the total number of tags, but no information regarding the particular tag values. Yet one can observe that actually most of the desired encoding values can be readily found within the tags themselves! Figure 2c illustrates this idea for a particular example of tags; the two bit words formed by least significant bit positions 0 and 2 form a unique encoding for each tag, obviating furthermore the necessity for a CAM mapping.

### 3.3 Programmable encoder

As discussed in the previous section and illustrated in Figure 2c, only a limited subset of tag bit positions can be utilized in order to distinguish the tags accessed by an application loop. A complete resolution with  $\lceil \log_2 n \rceil$  number of bit positions is not always possible, though. Figure 2d shows an illustrative set of four tags for which *none* of the possible pairs of two bit positions achieve a complete resolution; two bit unique values for tag encoding in this example are impossible. Yet, it can be observed that certain pairs of bit positions provide a significant resolution for the tags, while leaving only a small number of tags unresolved. Looking at the example in Figure 2, it is evident that bit positions 0 and 2 provide resolution for the two middle tag values, i.e., they can be completely distinguished by encoding them with their corresponding two-bit values formed from bit positions 0 and 2. Only the first and the last tag values are indistinguishable, since the value of “00” formed from bit positions 0 and 2 coincides for both tags.

The purpose of the programmable encoder is to resolve the occurrences of such conflicts and thus achieve an optimal  $\lceil \log_2 n \rceil$  encoding with no additional tag bitlines for longer codes. At the same time, its hardware overhead and even more importantly its power consumption need to be exceedingly frugal and negligibly small compared to the power savings achieved by disabling most of the bitlines and sense amps in the tag arrays.

Noteworthy is the fact that the encoder lies outside the critical path for accessing the cache, since it does not affect the cache index computation and operates only on the tag values from the effective address being generated by the processor core. The tag codes are normally computed in parallel with indexing and reading the tag arrays. The compressed tag values being read from the tag array are compared to the compressed tag value from the effective address being accessed. Because of the uniqueness of the encoded values and their full resolution capability, if there is a match, a cache hit is indicated; otherwise, a cache miss is identified.

## 4. STATIC ANALYSIS

In this section, we describe an algorithm that identifies, if possible, the  $\lceil \log_2 n \rceil$  bit positions that can provide full resolution; the algorithm alternatively provides the maximal resolution capability possible if a  $\lceil \log_2 n \rceil$  bit resolution is infeasible. This analysis is performed during application compile time in a post compile/link phase and the corresponding instructions for setting up the programmable hardware encoder are generated and inserted before the application loop entry point. The static analysis algorithm, presented in this section, is performed on a set of tag values being accessed by the major application loops. This set of tags is provided by the compiler/linker and it is performed only for the loops that operate on statically allocated data. If there are application loops that operate on dynamically allocated data, a rare situation in the domain of embedded applications, then these loops remain unaffected by our approach and they operate by using the complete address tags. At the same time, the rest of the loops for which the data is statically allocated and hence the tag set known after compile/link, can take full advantage of the proposed methodology.

Figure 3a shows an example of four tags, each of them four bits long. The proposed approach identifies whether there exists a combination of two bit positions such that their two bit values correspond to a unique encoding for the set of given tags. One can observe that each combination of two bit positions can be treated as a way to generate a partition of the boolean space  $B^4$ . For example, let's select  $X_3$  and  $X_4$  as such bit positions. By fixing a specific value at the bit position thus selected and allowing the remaining bits to assume all possible binary values, the resulting set of binary vectors becomes a boolean cube of dimension two. By fixing a different value at positions  $X_3$  and  $X_4$ , another boolean subcube of the same dimension is generated. In this way, the bit positions  $X_3$  and  $X_4$  generate a partition of the boolean space  $B^4$ . This partition consists of  $2^2$  non-overlapping boolean cubes that cover the entire space  $B^4$ . We denote such a partition as a 2-partition of the boolean space. The Karnaugh map presented in Figure 3b shows visually the  $X_3, X_4$  generated partition. After distributing the initial set of tags in the boolean space represented by the Karnaugh map, we observe that tags  $T_2$  and  $T_3$  lie in a single boolean cube of the  $X_3, X_4$  partition. This observation implies that these two

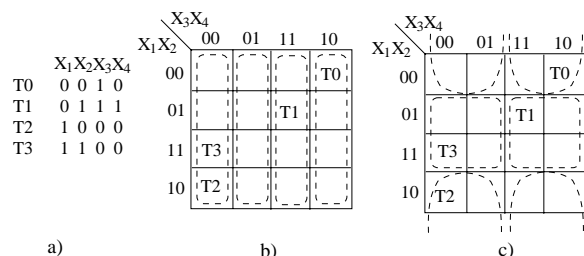


Figure 3: Tag bit identification for maximum tag resolution

tags cannot be distinguished, i.e., they are not uniquely encoded by bit positions  $X_3$  and  $X_4$ . On the other hand, as shown in Figure 3c, for the partition generated by  $X_2$  and  $X_3$  all tags from the initial set belong to a different subcube within the partition. This means that bit positions  $X_2$  and  $X_3$  provide a full resolution for the given set of tags. The problem can be consequently formulated in the following way. *Is there a 2-partition for which all tags from the initial set map to a distinct boolean cube within that partition?* It is evident that for  $B^m$  there exists a total of  $\binom{m}{2}$  such partitions.

A  $k$ -partition can be defined as a straightforward generalization of the 2-partitions defined above. A  $k$ -partition is defined by selecting a  $k$  bit position from a boolean space  $B^m$ . Consequently, the total number of  $k$ -partitions of the boolean space  $B^m$  is  $\binom{m}{k}$ .

The algorithm for finding the cluster of  $k$  bits with complete or maximal resolution can be defined in terms of  $k$ -partitions in the following way. If there is a  $k$ -partition for which all the tags from the initial set map to distinct boolean cubes within the partition, then this  $k$ -partition provides complete resolution for the loop tags. This means that the  $k$  bit positions associated to the partition can provide complete tag resolution and are sufficient for storing in the tag arrays. If there is no  $k$ -partition with this property, then the algorithm searches for the  $k$ -partition with the minimal number of conflicts. Conflicts within a partition can be quantified in two ways, primarily, by the number of conflicting sets and secondarily, by the number of conflicting tags within these sets. For example, figure 3b shows a partition with one conflicting set and two tags within the conflicting set. As will be specified later, the implementation of the programmable encoder would support a certain number of conflicting sets and tags within these sets.

The pseudocode of the algorithm for finding the optimal partition of  $k = \lceil \log_2 n \rceil$  bits is given below. In the pseudocode,  $B_t$  corresponds to a boolean cube of dimension  $t = m - k$ , where  $m$  is the dimension of the original boolean space (aka, the bit length of the original tags), and  $P_k$  denotes the corresponding partition of  $B_t$ . It is easily observed that a boolean cube of dimension  $t$  belongs to a single  $k$ -partition.

```
T={T1,T2,..., Tn}; /*the set of loop tags*/
V={T1}; T=T-{T1};
for i=2 to n
  for all Tj in V
    for all Bt, such that Ti in Bt
      if Tj in Bt
        mark conflicts for Pk;
      endif
    endfor
  endfor
  V=V+{Ti};
endfor
```

At the conclusion of the algorithm, if there is a  $k$ -partition with no conflicts marked, then this partition corresponds to  $k$  bit positions with complete tag resolution. Otherwise, the partition of maximal resolution, i.e., the minimal number of conflicts is identified.

The running time of the algorithm is  $O(n^2 \binom{m}{\lceil \log_2 n \rceil})$ , where  $n$  denotes the number of tags being accessed by the loop and  $m$  the tag bit length. Note that this algorithm corresponds to a straight search through all possible partitions.

The algorithm described above can be optimized in terms of running time, by sorting the initial set of tags in such a way, so that for a given partition only consecutive tags in the sorted sequence

may map to the same boolean cube within the partition. This can be easily achieved by using the number formed by the partition defining bit positions as a key for the sorting process. Since the set of keys is a finite set of integers, a linear time sorting algorithm can be applied. Now the search for a conflicting set can be performed by a simple linear scan through the sorted tag set. The number of equal adjacent tag pairs would correspond to the total number of conflicting tags for the particular partition. This procedure is repeated for all  $k$ -partitions, leading to time complexity of  $O(n \binom{m}{\lceil \log_2 n \rceil})$ . The pseudocode for this optimized version of the optimal-partition search algorithm is presented below.

```
T={T1,T2,..., Tn}; /* the set of loop tags */
for all Pt do
  Sort T into Ts for Pt; /*linear time sort*/
  for i=1 to n-1
    if (Ts[i]==Ts[i+1])
      mark conflicts for Pt;
    endfor
  endfor
```

The next step of the proposed static analysis consists of the identification of additional bit positions for resolving the conflicting set within the identified partition. Each conflicting set is now treated as a separate set of tags (with much smaller size than the initial tag set), which in turn needs to be resolved with minimal number of bit positions. The algorithm described above is run on this smaller set of tags and additional bit positions for resolving the conflicting set are found. The example presented in Figure 4 illustrates the situation. Figure 4a shows a set of four tags for which there are no two bit positions that can be used for distinguishing the tags. The partition  $X_3, X_4$  provides a resolution with only one conflicting set as shown in Figure 4b. The conflicting set contains tags  $T0$  and  $T3$ . It is evident that bit position  $X_2$  provides resolution for these two tags. This conflict resolution is the main purpose of the programmable encoder. For tags  $T1$  and  $T2$ , it will directly utilize the values of bit positions  $X_3$  and  $X_4$  for tag encoding, while for tags  $T0$  and  $T3$ ,  $X_2$  will be utilized additionally in order to produce the encoded values for these tags. We denote the main bit positions  $X_3$  and  $X_4$  as *Tag Resolution bits (TR-bits)* and the additional bit positions, such as  $X_2$  in our example, as *Conflict Resolution bits (CR-bits)*. It is evident that different conflict sets might require distinct CR-bits. Note that the encoded values for  $T0$  and  $T3$  would be produced within the  $X_3, X_4$  bit positions, as shown in Figure 4c. Ultimately the tag arrays would use only the bitlines that correspond to the TR-bits, while the rest of the bitlines and sense amps will be disabled during the loop execution.

The hardware structure has to be restricted to a certain number of conflicting sets and tags within each of the conflicting set. We have observed experimentally that in practice the number of conflicting sets and their size is limited, an observation that can be

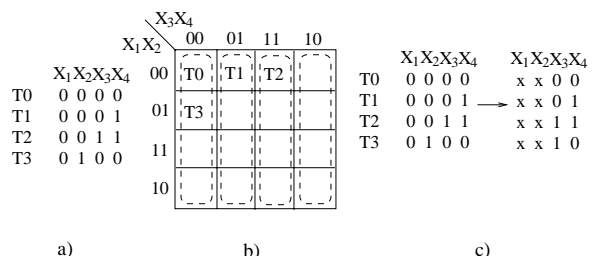
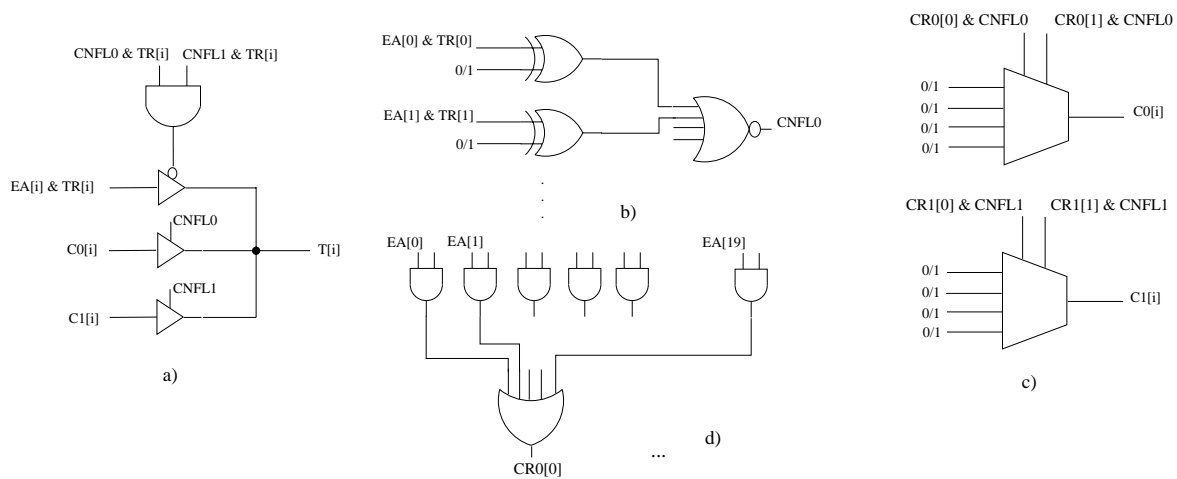


Figure 4: Conflicting sets and their resolution



**Figure 5: Hardware structure of the programmable encoder**

easily explained by the structure of the initial tag set. Typically, an application loop accesses a set of data that is packed within some memory regions. For example, an array would cover a memory region that corresponds to a sequence of arithmetically consecutive tag values. For such groups of tags, a set of least significant bits would provide a complete resolution. Therefore, if the loop accesses a number of such groups one would expect that a certain number of bit positions (mostly concentrated in the least significant part of the tag) would provide, if not a complete, but at least a very high resolution with only limited number of conflicts. Cases in which a loop accesses relatively random memory regions, thus resulting in a tag set with high amounts of conflicts within the optimal  $k$ -partition, are rare. Consequently, in the implementation we describe in the next section, the number of conflicting sets is limited to two and the number of conflicting tags within each of these sets to four. If these conditions are not met, then the algorithm will try to find a  $(k + 1)$ -partition, for which the aforementioned constraints in terms of conflicts are met. A threshold in this process can be imposed to ensure that inefficient solutions, with a large number of bitlines enabled and the savings from the remaining disabled bitlines outweighed, are not generated. While the imposition of such a threshold is important to ensure the completeness of the proposed algorithm, the experimental data we have obtained show no necessity for such a concern, as all solutions obtained were well within any such conceivable bounds.

The information about the TR-bits and the CR-bits for each conflicting set is provided to the programmable encoder by means of software. After the static analysis algorithm described above is performed on the set of tags for each major application loop, a set of instructions is inserted before the entry point of each loop. The purpose of these instructions is to store information about the TR-bits and CR-bits in special control registers within the encoder.

## 5. HARDWARE ARCHITECTURE

In order to implement the methodology described in the previous sections, a hardware support is needed for both tag bitline and sense amps gating and for the programmable encoder.

The hardware support needed for gating sense amps and bitlines is described in various other works, for example, [10]; consequently we concentrate our attention on the implementation of the programmable encoder. As discussed in section 3, the encoder lies outside the critical path of indexing the cache; therefore, in analyzing the encoder critical path, increased attention will be paid to

minimizing and analyzing its power consumption. It will be evident that the energy dissipated within the encoder is incomparable to the energy dissipated even for precharging/discharging a single bitline and its corresponding sense amp. The hardware that we describe in this section assumes at most two conflicting sets with a maximum of four tags within each conflicting set.

Figure 5 depicts the design of the programmable encoder. The output of the encoder contains  $m$  bits, where  $m$  is the width of the tag array. Only the selected cluster of  $k$  bit positions contains the encoded tag value, while the remaining bit positions are de-asserted and their corresponding bitlines disabled. Figure 5a shows the logic that drives the outputs of the encoders. The mask register  $TR$  is used to provide the gating signals for the tag array bitlines and sense amps. The signal  $TR[i]$  corresponds to the  $i^{th}$  position of the  $TR$  register; it is set to one in the selected tag bitline positions and to zero otherwise. The signal  $EA[i]$  corresponds to the  $i^{th}$  bit position of the effective address tag, while  $C0[i]$  and  $C1[i]$  represent conflict resolving values for both conflicting sets allowed. The signals  $CNFL0$  and  $CNFL1$  are asserted if the incoming tag belongs to conflicting set 0 or 1, respectively. The outputs  $T[i]$  of the encoder are provided directly to the tag array bitlines for writing or to the comparator inputs for cache conflict identification.

The implementation logic for defining the  $CNFL0$  signal is shown in Figure 5b. The logic for defining the  $CNFL1$  is identical. Conceptually, this module needs to compare the selected bit position within the tag with a constant that corresponds to the conflicting value for the particular conflicting set. Notice that only the XOR gates corresponding to the bit position specified by the  $TR$  mask register are activated, while the rest are gated and perform no transitions. The constants specified at the second XOR gate input correspond to the conflicting value for the enabled XOR gates and zeroes for the disabled cells. The outputs of these cells need to be fed to the inputs of a NOR gate in order to generate the  $CNFL0$  signal. Since this logic is simply a  $m$ -bit comparator, with only a subset of  $k$  bits performing transitions, any transistor level optimized implementation for this type of hardware structure can be utilized here. Since the constant inputs to the XOR gates depend on the particular conflicting set, they are implemented as a special register loaded with the value identified during the static analysis.

The definition of the  $C0$  and  $C1$  signals is shown in Figure 5c. Each of them is simply the output of a 4-to-1 multiplexer controlled by the Conflict Resolution (CR) bits. For each one of the  $C0$  and  $C1$  signals a pair of  $CR$  signals is needed to resolve the maximum

allowed conflict of four tags. The  $CR0[0]$  and  $CR0[1]$  is the pair for  $C0$ , while  $CR1[0]$  and  $CR1[1]$  is the pair for  $C1$ . The inputs of the multiplexer are constants that depend on the particular encoding to be assigned for the conflicting tags. These inputs correspond to a software controllable register, the value of which is defined through static analysis and assigned by software prior to the loop entrance. Note that the multiplexers are gated with the  $CNFL0$  and  $CNFL1$  signals. Consequently, they will perform transitions only for tags in the conflicting set, thus consuming negligible levels of power.

Figure 5d depicts the logic used for defining the  $CR0$  and  $CR1$  signals for driving the multiplexers from Figure 5c. These signals simply need to be selected from one of the effective address tag bits. The switching network depicted in the figure is a power efficient implementation for performing this operation. Note that the gating signals are defined prior to loop execution and that there are no signal transitions on the outputs of the *and* gates, but only on the selected one. The only signal that incurs transitions is the signal that comes from the selected bitline position. At the same time, the OR network performs transitions only in the part of it that corresponds to the line coming from the selected bit position. The gating signals of the *and* gates are directly defined from a control register, whose purpose is to select the bit position for the corresponding  $CR$  signal. The  $EA[i]$  inputs are directly the tag bit positions from the effective address. The example assumes a 20 bit long effective address tag field.

The encoder performs its operation while the tag array is being indexed and read. Therefore, it does not introduce any delay in the cache critical path. Furthermore, due to the simplicity of the proposed implementation and the fact that a large part of it is gated, the power consumed in this encoder is significantly smaller than the power consumed in the large tag array memories as only the comparator cells from Figure 5b are active during each cache access.

## 6. VLSI IMPLEMENTATION

The data cache tag subsystem, including the programmable encoder was implemented and simulated in SPICE, version 3f5. The complete encoder and the tag array were designed in TSMC 0.25 $\mu$ m CMOS process operating at 2.5V.

The tag array was designed as traditional SRAM blocks of 64, 128, 256 and 512 wordlines, and 18-21 bitlines, corresponding to the various cache organizations that were analyzed. Each bit-line is composed of the precharge unit, the SRAM cell and the sense amp. Precharge is implemented with three CMOS type-n transistors, while a traditional six transistor SRAM cell is utilized.

The purpose of the programmable encoder is to select the TR-bits, provide the gating signals to the tag arrays and in case of tags belonging to a conflicting set, to supply new encoded values within the TR-bit position, while utilizing the CR-bits in order to resolve the conflict. The encoder circuit was implemented as a custom design. In order to minimize the size and power consumed by the encoder, the number of conflicting sets was limited to two, along with a maximum of two extra TR bits for each conflicting set. The complete design of the encoder consists of 2,964 transistors and has a maximum delay of 3.9 nsecs.

## 7. EXPERIMENTAL RESULTS

The goal of our experimental study was to test the proposed methodology on a set of six real-life embedded applications. Three applications from the Mediabench collection of benchmarks [12] were utilized, namely *g721*, *gsm* and *mpeg*. The first two benchmarks are speech encoders, while *mpeg* is an mpeg video encoder.

	mpeg	mp3	g721	gsm	mmul	ej
#accesses	341.57	313.88	48.27	52.16	12.11	3.02
#loops	3	5	1	1	1	1
#loop part	93.06%	97.31%	100%	100%	100%	100%

Figure 6: Benchmark characteristics

*Mp3* is an open-source audio compression utility<sup>2</sup> for producing mp3 file format. Furthermore, we added to our set of benchmarks two numerical kernels: *mmul*, a matrix multiplication code on 100x100 matrices and the *Extrapolated Jacobi-iterative method (ej)* [13] on a 100x100 grid.

Figure 6 presents some benchmark characteristics. The first row gives the total number of memory accesses (in millions) for each application. The cache simulator from the SimpleScalar toolset [14] was used to simulate the benchmarks. All of them were simulated until completion. The second row of Figure 6 shows the number of major application loops, selected for applying the tag compression methodology, while the third row shows the percentage of execution time spent in these loops. It is noticeable that for such complex applications as *mpeg* and *mp3*, a relatively small number of loops accounts for more than 90% of total execution time. As discussed in the previous section, the cache needs to be flushed prior to entering each of these loops. In order to assess the impact of this cache invalidation and to collect the cache statistics for the application loops, we modified the SimpleScalar cache simulator, so that we can gather these statistics. Six cache configurations were evaluated: 16K and 32K caches with direct-mapped, 2-way set-associative, and 4-way set-associative organizations. For the *mpeg* benchmark and for the benchmarks with a single application loop, the miss rate increased by less than 0.01%, while for *mp3* the increase was within 0.12%.

In order to evaluate the cache energy dissipation, first we utilize the Cacti tool [15] in order to find the optimal subbanking for the tag and data cache arrays. The VLSI implementation and the subsequent SPICE simulations that were performed yielded the energy dissipation within the programmable encoder and the gated tag array. Since the energy for performing a write into the tag array is different from the energy to read, the cache miss and hits were distinguished and accounted accordingly.

Figure 7 shows the energy dissipation in uJ for the six cache configurations. The first table shows the results for the 16K caches, while the next one corresponds to the 32K caches. Since our approach targets the cache subsystem and does not practically impact the cache miss rate, the data in these tables corresponds only to the cache energy and does not account for the main memory accesses. For each cache configuration there are three columns. The first column gives the energy dissipation for the basic cache configuration with no support for tag encoding. The second column gives the energy dissipation support for tag encoding. The third column shows the percentage improvement relative to the base cache configuration. As expected, the increased cache associativity leads to larger power reduction since the power contribution of the tag arrays increases. The significant power reductions for the *g721* and *gsm* benchmarks stem from the fact that these applications operate on a very small memory region. Consequently, the number of different tags being accessed by them is limited and only two tag bits are needed for cache conflict identification for all cache configurations. Figure 8 illustrates the energy dissipation numbers only for the tag array of the utilized cache organizations. The organization of these two tables presented in Figure 8 is identical to the table or-

<sup>2</sup><http://www.sulaco.org/mp3/>

	DM cache			2-SA cache			4-SA cache		
	Energy	EnergyO	Impr.	Energy	EnergyO	Impr.	Energy	EnergyO	Impr.
mpeg	84232	79491	5.63%	93247	81230	12.89%	115631	89984	22.18%
mp3	77759	71550	7.99%	87916	74563	15.19%	107898	79334	26.47%
g721	8865	7918	10.68%	10020	7975	20.41%	12447	8091	35.00%
gsm	12404	11213	9.61%	14022	11374	18.88%	17419	11698	32.84%
mmul	996	921	7.52%	1125	946	15.88%	1398	1028	26.42%
ej	775	716	7.57%	875	749	14.42%	1087	822	24.77%

	DM cache			2-SA cache			4-SA cache		
	Energy	EnergyO	Impr.	Energy	EnergyO	Impr.	Energy	EnergyO	Impr.
mpeg	89247	83981	5.90%	99241	87572	11.76%	121945	94766	22.29%
mp3	83011	76792	7.49%	92598	78817	14.88%	113525	85016	25.11%
g721	9518	8575	9.91%	10679	8636	19.13%	13125	8758	33.27%
gsm	13325	12145	8.86%	14950	12316	17.62%	18371	12658	31.10%
mmul	1049	966	7.84%	1184	1010	14.67%	1470	1075	26.85%
ej	831	765	7.91%	932	795	14.74%	1146	862	24.77%

Figure 7: Energy (uJ) results and improvements for 16/32K direct mapped and 2,4-way set associative organizations

	DM cache			2-SA cache			4-SA cache		
	Energy	EnergyO	Impr.	Energy	EnergyO	Impr.	Energy	EnergyO	Impr.
mpeg	9885	5144	47.96%	20441	8424	58.79%	42957	17310	59.70%
mp3	9125	2916	68.05%	19267	5914	69.30%	40079	11515	71.27%
g721	1040	93	91.06%	2196	151	93.13%	4623	266	94.24%
gsm	1456	264	81.82%	3073	426	86.13%	6471	750	88.41%
mmul	116	41	64.10%	246	67	72.46%	519	149	71.15%
ej	90	32	64.51%	191	65	65.82%	403	139	65.52%

	DM cache			2-SA cache			4-SA cache		
	Energy	EnergyO	Impr.	Energy	EnergyO	Impr.	Energy	EnergyO	Impr.
mpeg	9739	4474	54.07%	20439	8771	57.09%	43147	15969	62.99%
mp3	9060	2841	68.64%	19075	5293	72.25%	40174	11665	70.96%
g721	1039	96	90.73%	2200	157	92.84%	4646	279	93.98%
gsm	1454	274	81.16%	3079	445	85.55%	6500	787	87.89%
mmul	114	32	71.87%	244	70	71.22%	520	125	71.22%
ej	90	25	72.39%	192	55	71.50%	405	121	69.97%

Figure 8: Tag only energy (uJ) results and improvements for 16/32K direct mapped and 2,4-way set associative organizations

ganization in Figure 7. As expected, the *gsm* and *g721* benchmarks exhibit extremely high levels of energy reduction due to the highly limited number of tag bits, 2 and 1, respectively. The data in these tables can be used for analyzing the total cache energy reduction for various data/tag cache configurations, being utilized.

## 8. CONCLUSION

Power consumption is becoming an important characteristic for product quality in various modern applications. In this paper we have proposed a novel technique for data cache tag compression. By utilizing very few tag bits within the major application loops, a significant amount of energy dissipation in the tag arrays is eliminated. We proposed a compile-time algorithm for finding the minimal number of tag bit positions for completely distinguishing the set of tags accessed by an application loop. The information about these bit positions is provided to a software controllable tag encoder. An efficient hardware implementation for the programmable encoder is proposed and evaluated in terms of time and power impact. An extensive set of experimental results on real-life embedded applications is presented. The experiments show significant energy dissipation reductions and confirm the applicability of the proposed approach on real-life modern applications.

## 9. REFERENCES

- [1] J. Montanaro et al., "A 160Mhz, 32b 0.5W CMOS RISC Micro-processor", in *IEEE ISCC Digest of Technical Papers*, pp. 214–229, February 1996.
- [2] M. B. Kamble and K. Ghose, "Analytical energy dissipation models for low-power caches", in *ISLPED*, pp. 143–148, August 1997.
- [3] K. Ghose and M. B. Kamble, "Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line seg-

mentation", in *ISLPED*, pp. 70–75, August 1999.

- [4] S. Manne, A. Klauser and D. Grunwald, "Pipeline gating: speculation control for energy reduction", in *25th ISCA*, pp. 132–141, June 1998.
- [5] N. Bellas, I. Hajj and C. Polychronopoulos, "Using dynamic cache management techniques to reduce energy in a high-performance processor", in *ISLPED*, pp. 64–69, August 1999.
- [6] J. Kin, M. Gupta and W. H. Mangione-Smith, "The filter cache: an energy efficient memory structure", in *30th MICRO*, pp. 184–193, April 1997.
- [7] P. R. Panda and N. D. Dutt, "Low-power memory mapping through reducing address bus activity", *IEEE Transactions on VLSI Systems*, vol. 7, pp. 309–320, 1999.
- [8] D. H. Albonese, "Selective Cache Ways: On-Demand Cache Resource Allocation", in *32nd MICRO*, pp. 248–259, November 1999.
- [9] T. Simunic, L. Benini, A. Acquavia, P. Glynn and G. De Micheli, "Dynamic voltage scaling and power management for portable systems", in *38th DAC*, pp. 524–529, June 2001.
- [10] L. Villa, M. Zhang and K. Asanovic, "Dynamic zero compression for cache energy reduction", in *33rd MICRO*, pp. 214–220, December 2000.
- [11] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim and W. Ye, "Energy-driven integrated hardware-software optimizations using SimplePower", in *27th ISCA*, pp. 95–106, June 2000.
- [12] C. Lee, M. Potkonjak and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", in *30th MICRO*, pp. 330–335, December 1997.
- [13] S. Nakamura, *Applied Numerical Methods with Software*, Prentice-Hall, Englewood Cliffs, N.J., 1991.
- [14] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0", Technical Report 1342, University of Wisconsin-Madison, CS Department, June 1997.
- [15] G. Reinman and N. Jouppi, "An Integrated Cache Timing and Power Model", Technical report, Western Research Lab, 1999.