ABSTRACT
In this paper we present a framework for dynamic application customization for high-performance and low-power embedded processors. The proposed architecture is capable of utilizing application information to boost the performance and lower the power consumption of the most important microarchitectural components such as instruction/data caches and the memory subsystem. We present a design framework, including CAD support infrastructure and reprogrammable hardware support, for a dynamically customizable microarchitecture. We outline the underlying algorithms for compile-time extraction of the utilized application properties and we present the architectural principles of the hardware support. Extensive experimental results confirm the efficacy of this novel embedded processor architecture.

1. INTRODUCTION
The proliferation of nanometer technologies with ever decreasing geometry sizes have led to proportionally unprecedented levels of System-On-a-Chip (SOC) integration. Embedding a multitude of processor cores has become a widespread design practice due to its advantages of highly reduced time-to-market, lower design cost, and easily reprogrammable implementations. The inherent drawbacks of processor-based designs, typically stemming from the innate redundancy of the processor computational model, impose significant challenges though in achieving overall system requirements, particularly performance and power.

Embedded processors, deeply integrated and utilized for a multitude of SOC sub-systems, have already established themselves as the unquestionable leader in terms of volumes, electronic market share, and revenues, greatly surpassing their close relatives, the general-purpose processors. By virtue of their very nature, many general-purpose optimization techniques are either not directly applicable due to their non-trivial overhead, or just scratch the surface of otherwise promising optimization opportunities, being severely limited by their generality. The dramatic volumes in the embedded processor marketplace necessitate innovative approaches, particularly to surmount the obstacles posed by performance and power considerations in a diverse set of embedded processor market segments. Meeting the more stringent performance and power challenges in the embedded processor marketplace necessitates an ability to exploit the one aspect that differentiates embedded processors from general-purpose processors, namely, advance knowledge regarding their application context.

The elaborate exploitation of application knowledge and its subsequent utilization as a driving force for customizing the embedded processor microarchitecture constitute a conceptual framework that establishes new grounds, enabling a plethora of opportunities for drastically improving both system power and performance [1, 2, 3]. We show that customizable microarchitectural components, such as instruction/data caches capable of utilizing precise application knowledge regarding data reuse regularities can achieve significant performance improvements through miss rate reductions [1], while information regarding the particular memory layout can be exploited in a way that leads to highly optimized tag operations with the concomitant drastic power reductions [2]. Application information regarding data memory address streams can be efficiently exploited [3] in order to achieve significant power reductions on the address bus to data memories as more than 90% of the address traffic can be completely eliminated. In all these cases, an application-specific customizable microarchitecture takes informed decisions as to how to handle various architecture-specific actions. As this framework enables the microarchitectural utilization of global application properties identified off-line by compile/link time algorithms, traditional mainstream compiler algorithms remain unaffected.

A well known rule-of-thumb in computer architecture is that a program spends 90% of its execution cycles only in 10% of its code. As practice shows, the embedded processor applications certainly do not violate this principle. Typically every embedded application can be easily partitioned into a few major loops that contribute almost all of the execution cycles, while covering a relatively small fraction of the static program code. Furthermore, the various application hot-spots are not only quite tractable due to their limited size, but are also extremely independent, thus enabling locally applied and therefore inexpensive optimization techniques. Consequently, concentrating the performance and power optimizing customization efforts on these application “hot-spots” delivers maximum benefits with minimal compiler and hardware cost.

Since flexible and cost-efficient implementations together with ease of maintenance are the primary advantages buttressing the wide acceptance of processor-centric system designs, it is of paramount importance that the customization support be designed as a microarchitecturally reprogrammable implementation capable of dynamic recustomization. This reprogrammability is achieved on a microarchitectural level, rather than through gate-level approaches such as FPGAs, thus achieving cost-efficient performance and power improvements. We propose an architecture, capable of utilizing application-specific information in a programmable way; hence the ability to re-customize in a post-manufacturing fashion helps effectively cover diverse applications with no need for spinning new silicon, an important advantage in terms of flexibility for a number of high-performance embedded systems. Consequently, the proposed customization architecture constitutes a unified microarchitectural solution capable of handling diverse workloads through in-field re-customization.

In this paper we outline three application-specific customization approaches. These techniques aim at performance and power optimizations focused on the data/instruction cache and the interaction to data memories. The significant importance of the instruction and data caches in terms of the overall processor power consumption has been widely recognized; various researchers report ranges up to 50% [5, 7]. The tag operations are typically responsible for a significant part of the total cache power consumption. Their contribution is even larger for the commonly used low-power and highly associative caches [5]. The contribution of the tag operations to the cache power consumption can range up to 54% as reported in [6]. With the introduction of

This work is supported by NSF Grant 0082325.
deep submicron technologies, long interconnect buses in SOCs have become a significant factor in system power consumption and performance. In Section 2 we present the Partitioned Cache Architecture. This is a customization methodology that targets the elimination of cache pollution and interference for reducing the cache miss-rate by virtually partitioning the cache data arrays. Additionally, we show how application knowledge can be utilized to eliminate a large number of tag operations which results in significant power reductions. Section 3 presents a low-power customization technique for tag compression. Application information regarding code and data layout is exploited in order to drastically reduce the number of tag bits needed for cache hit/miss identification. Finally, in Section 4 we outline a customization methodology for power reduction in address buses to data memories. By exploiting knowledge regarding load/store effective address order as well as the strides of the address streams, we present an architecture capable of autonomously generating the load/store effective addresses on the memory side, thus eliminating a significant portion of the address traffic and its associated power consumption.

2. PARTITIONED CACHE ARCHITECTURE

Caches are used to bring instructions/data from slower memories closer to the processor core, so that the time needed to fetch the data is minimized. The referenced effective address is separated into a line index (li), a cache index (ci), and a tag. The line index is used to address a word within a cache line, while the cache index is used to address the cache line; the tag field checks whether memory locations that map to the same cache index conflict. Increased set-associativity results in a shorter cache index (smaller number of sets) and thus wider tags. In the extreme case of a fully associative cache, almost the entire address (except the block index) forms the tag. The tag field associated to each cache line is stored in a separate tag memory array. Each time an access is performed to the cache, the tag associated to the cache line is read and compared to the tag of the effective address being referred.

A typical data-intensive algorithm consists of matrix and/or array operations grouped in several nested loops. For example, figure 1 shows an excerpt from the swim SPEC95fp benchmark. One can notice that the references to U and V exhibit only spatial locality. All references to PSI utilize spatial locality as well, but there is also a temporal locality between PSI[i+1,j] and PSI[i+1,j+1]. These references reuse a row of the matrix PSI along the outer loop i. The references PSI[i+1,j] and PSI[i+1,j+1] will always result in hits if the data brought by PSI[i+1,j+1] is judiciously preserved in the cache. Not only do such judicious data presentations promote performance and power improvements through the evident miss rate reduction, but furthermore a large number of power consuming tag operations can be completely avoided, if knowledge regarding whether a particular array reference will indeed hit in the cache can be captured.

```
for i=1 to N
  for j=1 to M
    U[i,j+1]=-(PSI[i+1,j+1]-PSI[i,j+1])/DY;
    V[i+1,j]=-(PSI[i+1,j+1]-PSI[i+1,j])/DX;
  end for
end for
```

Figure 1: An excerpt from the swim benchmark.

Let’s consider, for example, the write references to U and V. They are brought into the cache and a new cache line is used when the previous one is filled. This situation occurs when an array (or matrix) is traversed sequentially. In this type of access only spatial locality exists and it is not necessary to use more than one cache line to capture this locality. Such reference behavior leads to significant cache pollution and interference with the remaining working set, as using more than one cache line leads to no benefits in terms of reuse. The inherent spatial reuse can be exploited using a single cache line, if a more sophisticated caching technique were to be utilized. A similar problem of interference may occur when temporal reuse, with overlapping accesses in the data cache, amongst more than one array exists; these accesses may prevent the utilization of the temporal reuse.

A conventional cache organization suffers from the inability to distinguish different types of reuse along the loop iterations. All references are treated identically; thus significant interference between unrelated data arrays and cache pollution is introduced. Caching the data intelligently by avoiding interferences and pollution would be highly beneficial in terms of both performance and energy dissipation gains. At the same time, a significant number of tag reads and comparisons can be eliminated, were the interference to be avoided. For example, it is evident that the references PSI[i+1,j] and PSI[i,j+1] would always hit in the cache because of the reference PSI[i+1,j+1]. Consequently, a significant power reduction can be achieved through elimination of the tag operations and reduction of the miss rate (which will lead to fewer accesses to the even more power consuming L2 cache or main memory). Solving these problems of performance and power simultaneously is a rather difficult task for a general-purpose cache controller.

The group of references to the matrix PSI exhibits temporal reuse along iteration i. Namely, the usage of rows PSI[i] and PSI[i+1] is overlapped in the computation process. In order to exploit this reuse, the rows need to be protected from interferences by U and V that might affect them during iteration j. Having isolated the references to PSI from other unrelated references, the tag operations associated with the references to PSI[i] can be completely eliminated. Due to interference and volume limitations, a conventional cache organization fails to exploit the reuse and to eliminate the unnecessary tag operations. Cache interference and pollution problems can be resolved in an application-specific environment, wherein more precise information about the inherent reusability can be provided to the cache controller. If the memory instructions were to be grouped according to the inherent reuse characteristics amongst them and each group subsequently mapped to a dedicated cache partition, behaving in the same way as a distinct cache, all conflicts would be obviated and redundant tag manipulations completely avoided. The size of each cache partition can thus be reduced to no larger than the minimal sufficient size to exploit the inherent reuse for that particular group of instructions.

2.1 Partitioning overview

The proposed partitioning analysis utilizes information about the type of reuse of each reference. A formal methodology for determining the reuse type of array references with affine indices is presented in [4]. Since the methodology we propose utilizes information about reuse type in a loop nest, we briefly review the relevant terminology.

By isolating a group of load/store instructions from other unrelated and possibly interfering groups of memory references, the detrimental effects of cache conflicts can be minimized. The grouped instructions can be considered as a set composed of a leading reference and several trailing references. The leading reference does fetch data from memory but misses only once per cache line. All the trailing references invariably hit in the cache. Consequently, no tag operations are needed for the trailing references within a partition. Furthermore, if the data access is single strided, the leading reference would miss only in the beginning of a cache line. Therefore, in this case no tag operations are needed for the leading reference as well.

2.2 Algorithm Overview

We capture the information about the inherent reuse for a particular loop dimension by constructing a Data Reuse Graph (DRG). Each node in the DRG corresponds to a particular load/store instruction or
for i=1 to N
    f(A[i], A[i+1], A[i+4], A[i+7], A[i+8], A[i+12]);
    g(B[i], B[i+2]);
    h(C[i], C[i+3]);
end for

Figure 2: Example of temporal reuse.

to an already formed group. The edges in the DRG represent data reuse between the corresponding nodes. Each edge is annotated with the particular type of reuse it represents. Additionally, an integer k is associated to every temporal reuse denotation, representing the number of iterations needed to exploit the temporal reuse denoted by the edge. The number of iterations in turn determines the cache volume needed to exploit the reuse.

The optimal cache partition size, CV (cache volume), varies depending on the reuse type. It is evident that a spatial reuse necessitates a fixed number of cache lines needed to exploit the reuse and prevent interference. Additionally, an integer k is associated to every temporal reuse denotation, representing the number of iterations needed to exploit the reuse and prevent interference.

The objective of the partitioning algorithm is to capture data reuse amongst them and map this group to a cache partition with appropriate size. From a DRG perspective, this implies selecting DRG edges and grouping the neighbouring selected edges together into partitions. While each edge selected provides a constant benefit in capturing a reuse, the cost in terms of cache volume to accommodate an edge varies. Consequently, the objective of the algorithm is to maximize the number of selected DRG edges.

An example of data reuse can be seen in Figure 2. We assume a cache line size of one word to simplify the explanation. Figure 3a shows the DRG for the example in Figure 2. The number of cache lines needed to capture the corresponding group reuse is therein shown. The initially appealing solution of a direct, greedy approach is unfortunately inadequate in determining the optimal number of partitions. If we assume an available cache volume of 9 cache lines, a direct greedy approach leads to the result shown in Figure 3b. It is evident though that a solution consisting of a single partition covering all references to the array A but the last one of A[i+12] is superior in that it covers instead four reuses in A and furthermore utilizes all 9 cache lines exactly.

In order to achieve an optimal solution we introduce a model of representation that takes into account the cause of the inadequacy, to wit, the overlap between the nodes in the DRG. In its initial phase, the algorithm separates the connected edges while keeping track of the overlap by building Overlap Dependence Trees (ODT) as shown in Figure 3c. The purpose of the ODT is to represent the overlap dependencies amongst the edges in the DRG and to show the exact cache volume needed to cover a particular edge in the DRG after removing the overlap. Each node in the ODT corresponds to an edge in the DRG. The edges in the ODT represent the overlap relation between the corresponding edges in the DRG. The ODT for the example in Figure 2 is shown in Figure 3c. Each node in the ODT is annotated with the updated CV. Given the ODT, the purpose of the algorithm is to find the maximum subset of nodes subject to the constraints that the total available cache volume not be exceeded and that overlap dependencies be preserved.

2.3 Hardware support

The proposed partitioning methodology requires special hardware support from the cache controller. The hardware support for the partitioned cache has to resolve the following problems: identification of the mapping between a memory instruction and a particular cache partition; identification of the trailing load instructions that can avoid the tag read and comparison, and calculation of the cache partition index using the pair of numbers identifying the partition. The cache is virtually partitioned into sub-caches, each of them accommodating a group of load/store instructions. Each cache partition is identified by two parameters: the number of cache lines (size) and offset (position) in the original cache array.

2.3.1 Identifying the cache partitions

In order to address a particular cache partition as a distinct cache, a slight modification of the traditional cache indexing scheme needs to be effected. Depending on the size of the cache partition, the cache index part of the address is divided into two parts. If the size of the corresponding cache partition is 2^n, then the n least significant bits from the cache index are used to form the new index. The remaining most significant bits from the cache index are replaced by a constant in the newly formed cache partition index. The value of this constant determines the offset of the cache partition in the original cache. The above reasoning evinces that each cache partition with size 2^n is identified by a pair of numbers (offset, cache partition index size(a)). The cache partition index is formed by concatenating the offset and the n least significant bits from the cache index.

The partition mapping identification is achieved by a hardware architecture utilizing two tables: the Partition Mapping Identification Table (PMIT) and the Partition Identification Table (PIT). The PMIT is used to define the mapping between load/store instructions and cache partitions. The PMIT is indexed using the least significant PC bits of the load/store instruction from the loop. The size of the PMIT corresponds to the total number of instructions within the loop nest. In practice the size of a loop nest in data intensive applications is rarely large, thus leading to implementations with a small number of entries. When a load/store instruction is decoded, the PMIT is indexed with the least significant bits of the PC. An entry in PMIT contains a value that represents an index into the PIT, which in turn contains a partition.
defining information. An additional bit, \( tr \), is also stored in the PMIT entry to indicate whether the instruction is a *trailing* reference for the partition. The main purpose of this organization is to avoid associative lookups, which are expensive in terms of power. The tables described above are directly indexed; their size, negligible compared to the size of the tag memory arrays, ensures that no significant amount of energy dissipation is introduced. All memory references in a loop nest that are left unpartitioned by the reference analysis are mapped to a dedicated cache partition. This special partition is treated in the same way as the remaining cache partitions.

### 2.3.2 Computing the cache index

The lookup into the PMIT and PIT is the first step in determining the cache partition index and is performed early in the pipeline, thus not affecting the cache access time. Right after the load/store is decoded, the lookup is performed in parallel with the effective address calculation. Figure 4 shows the implementation of the cache partition index calculation. The Cache Index Template (CIT) and control signals \( C[i] \) are computed before the actual cache access pipeline stage using the partition information found in PIT. The CIT is defined as having the offset value in its most significant bits and zeroes in its \( n \) least significant bits resulting in control signals \( C[i] \) defined as \( C[i] = 1 \) for \( 0 \leq i < n \), and \( C[i] = 0 \) for \( i \geq n \). The Effective Address Cache Index (EACI) is the traditional cache index field in the effective address. The Cache Index (CI) is computed using the simple combinatorial logic depicted in Figure 4. The delay of the two gates shown in this figure is the sole, evidently insignificant, increase in the path delay of the cache access datapath.

### 2.4 Experimental results

In our experimental evaluation of the partitioned cache architecture we have used the following benchmarks: *Swim benchmark* (swim), part of the SPEC95fp benchmark suit, characterized by a high cache miss-rate due to a large amount of interference; *Tri-diagonal system solver* (tri), a fundamental part of *tomcatv* SPEC95fp benchmark and a major contributor to the high miss rate for the *tomcatv* benchmark, with matrix size of 128x128; *Extrapolated Jacobi-iterative method* (ej) on a 128x128 grid; *Successive over-relaxation* (sor) on a matrix with size 256x256.

Figure 5 shows the results for the partitioned cache. Within each cache partition, the number of conflict misses is reduced to zero. This follows directly from the way the cache partitions are formed. Consequently, only the cold misses for the references within the partitions need to be considered in terms of cache miss behavior. Furthermore, one can observe that only the leading reference of the partition exhibits cold misses, due to its inherent role in bringing data into the partition for subsequent spatial reuse for itself and temporal reuse for the trailing references. We can conclude, therefore, that the conflict misses for the arrays targeted by partitions have been completely eliminated and reduced to zero; only a single memory reference within the partition exhibits cold misses when bringing in a new cache line.

Figure 6 shows the energy consumption results (in mJ) for an 8K cache. The PC-T column shows the energy for the partitioned cache, but with no tag optimization. The next two columns display the power improvements in percentages, compared to the direct-mapped and the 2-way set associative caches. The PC column represents the energy consumption for the partitioned cache including both the miss rate reduction and the tag optimizations. Finally, the last two columns present the total improvement in percentages. The energy consumption improvements vary from 14% to 35%. The negative result for sor in comparison to a direct-mapped cache in the case of no tag optimizations is due to the lack of miss rate reduction for this benchmark, the extended tag array, and the slight overhead of the partitioned cache in terms of PMIT and PIT. This benchmark is a worst case scenario in terms of power for a partitioned cache with no tag-related power optimization support, since it includes a large number of array references utilizing the extended tag array with no decrease in the miss rate. Nevertheless, when the tag optimizations we propose are included, the energy savings for this benchmark are not only significant, but exhibit the highest level of power reductions.

### 3. TAG COMPRESSION

Conceptually, the tag plays the role of a key for distinguishing two distinct memory addresses being mapped to the same cache line. If the referred locations neighbor in the address space, a large part of their tags would be consequently identical and therefore redundant. The cache hit/miss identification relies on the fact that memory locations conflicting in the cache would always have distinct tags; comparing the tag associated with the cache line and the tag of the referred memory address thus provides cache hit/miss identification. Typically, an application hot spot would access data/code, which occupies memory regions with significantly smaller total size compared to the entire addressable memory space. Consequently, there exists a high amount of redundancy in terms of the tag width since generally the tags are defined in such a way so as to handle accesses to arbitrary memory locations throughout the entire memory space. Therefore, a large amount of power is spent in reading, comparing and writing unnecessarily large tags, unless the aforementioned redundancy is eliminated.

Every application hot-spot accesses a limited number of memory segments, thus generating a relatively small number of distinct tags. If we consider the instruction cache functionality for a particular application hot-spot, usually a loop or a function, the memory accesses would be confined only within its own code and possibly the code of the functions being called. Consequently, the memory regions accessed within the instruction memory would be limited in numbers and furthermore would exhibit certain properties such as close proximity and adjacency. Similarly, if we consider the data cache behavior while executing a particular hot-spot, the number of memory regions being accessed would be considerably smaller compared to the entire memory space and may exhibit alternative properties that can be exploited to effectively reduce the number of tag bits needed to identify whether a hit or miss has occurred while accessing the data cache.

#### 3.1 Basic tag reduction (BTR)

Let’s assume that a particular application hot-spot accesses three data arrays, \( A \), \( B \), and \( C \). Figures 7a and 7b depict the memory layout of the three matrices \( A, B, \) and \( C \) from the example. The effective addresses of the arrays are shown and split into tag, cache index, and line index according to the cache organization. The data memory space is divided into regions that correspond to one cache size and for each of
these memory regions the tag part of the address is a constant. We denote these memory regions as 0-tag regions. The left part of Figure 7 shows a configuration in which the data set of the example resides within a single 0-tag region. It is evident that in such a case there will be no conflicts within the arrays of data in the cache; no need to perform any tag operations exists, consequently, as the tag part of the addresses is identical. Figure 7a depicts a configuration in which the dataset spans two 0-tag regions. In this case there will be conflicts in the cache; nonetheless, the tag fields of two conflicting addresses will differ on average by an exceedingly small number of bits and in the particular example differ only in the least significant bit. If the least significant bit of the tag for a given tag region is 0, then the address tag associated with the subsequent tag region in the memory will differ only in the least significant bit, which will be 1. Such pairs of 0-tag regions are denoted as 1-tag regions; a k-tag region is defined in a similar manner as a group of two (k-1)-tag regions. It is evident for the example in Figure 7b that only checking the least significant tag bit is enough to determine whether there is a cache miss or hit. Therefore, only the least significant tag bit needs to be read from the tag arrays, while all the remaining bits can be gated, thus saving an appreciable amount of power.

This tag reduction technique simply relies on the observation that if all the addresses being accessed by an application lie within a k-tag region, then the k least significant tag bits suffice for complete cache conflict identification and therefore can be used as active tags. This follows as all but the k least significant bits of a k-tag region are identical by definition. This identical part of the tags generated by the application loop carries no additional information in terms of cache conflict resolution. For instance, the data layout shown in Figure 7b implies the utilization of only a single least significant tag bit, since the data resides within a 1-tag region. This implies that all the tag operations, such as reading the tag from the tag arrays and comparing the tags, can be performed by using a single bit.

### 3.2 Unconstrained tag reduction (UTR) technique

The BTR technique identifies the smallest k-tag region, within which the accessed data set resides and essentially treats this tag region as if it is the entire memory space. This interpretation is valid since the most significant tag bits after the kth bit are identical for all the tags within the k-tag region, while the k least significant bits provide complete tag resolution. Yet, as the example in Figure 8a shows, the smallest k-tag region does not necessarily provide the minimal number of least significant tag bits for tag resolution. Figure 8 shows three different memory layouts accessed from a particular application loop. The 0-tag regions in which the blocks reside are shown on the left side for each memory configuration.

The data blocks in Figure 8a reside within a 5-tag region, while it is evident that only the 2 least significant tag bits suffice for distinguishing all three tags shown. Selection of the 2 least significant tag bits in this example illustrates that the tag bits to be eliminated are not restricted to being identical as long as the tag bits utilized suffice in completely distinguishing all the 0-tag regions. The Unconstrained Tag Reduction (UTR) technique identifies the minimal number of least significant tag bits, which are necessary to completely distinguish all the tags generated by the application hot spot and uses these bits as active tag bits.

Figure 8b illustrates a case for which the data lies within four consecutive 0-tag regions, i.e. four different tags are to be generated by the application hot spot under consideration. The BTR technique would have selected all 5 tag bits as active bits, since the data set clearly resides within a 5-tag region. Yet, it is evident that in any case of four consecutive tags, the two least significant tag bits suffice for cache conflict identification. It can be observed that these two bits differ for all the tags within such a memory region and therefore these two bits provide a full resolution for the set of memory addresses referencing this memory region. In general, if the data set resides within n consecutive 0-tag regions, to distinguish all n different tags, the ⌈log₂ n ⌉ least significant tag bits suffice, as any number of adjacent least significant bits taken from a sequence of contiguous binary numbers would in turn form a sequence of distinct, contiguous and yet shorter binary numbers.

The example in Figure 8a that we examined already suggests that the UTR technique can handle such cases quite efficiently. Yet further analysis is needed to establish the benefits of the UTR in this more general situation. Figure 8c shows another such data layout. The first part of the data occupies two 0-tag regions, while the rest of the code resides in another two 0-tag regions, but separated from the first two. If one considers the unused memory space between the two groups of data as part of the entire group, then the data spans a total of 32 0-tag regions, thus necessitating a total of five least significant tag bits for cache conflict identification. It is evident that this is a worst case scenario, and as the example shows, in this particular case actually only two least significant bits suffice for complete tag resolution, as the least significant two bits happen to be able to differentiate all four differing tags independent of the intervening unused 0-tag regions.

### 3.3 General tag reduction (GTR) technique

Conceptually, the tag reduction approaches that we have so far described utilize only a subset of tag bits, which are able to completely distinguish the set of tags generated by the application hot spot. The tag bits selected for tag resolution have hitherto always been confined to a contiguous set of least significant bit positions. Neither contiguity nor a clustering in the least significant bits is necessary for the selected tag bits, though! Figure 9 shows an illustrative set of tags accessed by

```
T0: 010000 000000 a1x0x
T1: 000110 000x1x
T2: 010111 011111 a1xx1x
T3: 101100 100000 a0xx0x
```

a) Complete tag resolution with least significant bits
b) Complete tag resolution with arbitrary bits

Figure 9: GTR tag compression
a particular application loop. This set of tags corresponds to the four 0-tag regions accessed by the loop. Utilizing the UTR technique results in selecting the least significant tag bits for complete tag resolution, as can be seen in Figure 9. Nonetheless, if the bit selection process was not restricted to least significant bits only, an optimal solution in terms of the number of tag bits could be identified. If tag bits at positions 1 and 4 are to be selected instead, as in Figure 9, it is evident that complete tag resolution can be achieved with these two bits only, as the bit strings formed by these two bit positions happen to differ for all the tags.

Generally, a solution comprising of $\log_2 n$ tag bit positions providing complete resolution for a set of n tags, is an optimal solution. Essentially, the UTR technique achieves this optimality only if the 0-tag regions being utilized happen to be positioned in such a way so that the tags differ in their $\log_2 n$ least significant bits. In general, when the application hot spot accesses data scattered throughout the memory, the tag resolution solution identified by the UTR might be far from the optimal one. The GTR technique allows the selection of arbitrary tag bits, so as to achieve a solution comprising of minimal number of tag bits with complete tag resolution.

### 3.4 Hardware support

We present an efficient hardware for manipulating the tag memory array so that only the required minimal number of tag bits are used per application loop. The tag array in the cache subsystem is typically implemented as an SRAM array, possibly divided into multiple banks. The SRAM data array contains wordlines for each tag data and a bitline for each bit within the tag word.

By eliminating most of the bitline precharge and discharge operations, our approach greatly reduces the energy dissipation in the tag SRAM array. This is achieved by gating the bitlines according to the minimal number of tag bits required to check for cache conflicts. Only the needed bitlines, if any, are precharged and discharged, thus effectively eliminating the redundant reads. The sense amplifiers for the disabled bitlines are gated as well. Furthermore, the tag comparator cells are gated in order to perform the comparison only on the required tag bits.

The number of tag bits for each loop needs to be determined before entering the loop, so that the appropriate number of bitlines are enabled. Since this number is fixed for the loop, it can be stored in a special control register, the Tag Enable (TE) register, before entering the loop. Each bit in the TE directly corresponds to an enable signal of bitline and sense amplifiers. The default value of this register specifies that all tag bitlines are enabled. The actual value of this register is used to determine the number of bitlines to enable. The only delay imposed by this implementation consists of the insignificant delay of the gating logic, roughly comparable to the delay of a simple and gate. However, this delay is hidden by the delay of the address decoder since they operate in parallel and the address decoder has significantly higher delay compared to the gating logic for the bitlines.

### 3.5 Experimental results

Our experimental results indicate the power reductions obtained for the tag arrays of instruction and data caches. In our experimental studies we have used the media benchmarks adpcm, g2t1, gsm, epic, jpeg, mpeg [8] and an additional benchmark, namely, the mp3 encoder benchmark. In this paper we present the power reduction results for the GTR technique only. Extended experimental results can be found in [2, 9].

Figure 10 shows the results obtained by applying the GTR technique on the set of benchmarks. The first triplet of rows shows the number of tag bitlines per application hot-spot, the energy consumption of the tag arrays of instruction and data caches. In our experimental studies we have used the media benchmarks adpcm, g2t1, gsm, epic, jpeg, mpeg [8] and an additional benchmark, namely, the mp3 encoder benchmark. In this paper we present the power reduction results for the GTR technique only. Extended experimental results can be found in [2, 9].

The number of tag bits for each loop needs to be determined before entering the loop, so that the appropriate number of bitlines are enabled. Since this number is fixed for the loop, it can be stored in a special control register, the Tag Enable (TE) register, before entering the loop. Each bit in the TE directly corresponds to an enable signal of bitline and sense amplifiers. The default value of this register specifies that all tag bitlines are enabled. The actual value of this register is used to determine the number of bitlines to enable. The only delay imposed by this implementation consists of the insignificant delay of the gating logic, roughly comparable to the delay of a simple and gate. However, this delay is hidden by the delay of the address decoder since they operate in parallel and the address decoder has significantly higher delay compared to the gating logic for the bitlines.

### 4. DATA MEMORY COMMUNICATION

In typical data intensive applications, such as DSP or numerical analysis algorithms, the program execution is concentrated within a small number of heavily executed loops. These application loops frequently contain multiple nesting levels, reflecting the dimension of the problem being solved. The data being processed resides in large array structures and is commonly accessed using affine index calculations [10]. In order to eliminate or minimize to a large extent the amount of information that needs to be sent on the address bus to the data memory, application knowledge about the strides of the memory accesses as well as the execution order of these accesses is required on the data memory side. Knowledge of address strides for each load/store instruction and the relative order of execution within the data memory
controller can help autonomously generate the next effective address with no need of an effective address sent by the processor, simply by adding the corresponding stride and processing the load/store request. Let’s consider, for example, the following loop.

```
for i=1 to 100
    for j=1 to 100
        A[2i,j]=B[i,2j+5]+C[i+1,3j+1];
    endfor
endfor
```

The references to $B$ and $C$ are loads from the data memory, while the reference to $A$ is a store. Nonetheless, all three references need to send an effective address to the data memory in order to perform their memory operation. Each of these three references constitutes a series of regular accesses with a constant stride. In the inner loop, for instance, the reference $A[2i,j]$ generates a sequence of incremental effective addresses, while the references to $B[i, 2j + 5]$ and $C[i + 1, 3j + 1]$ generate streams with strides 2 and 3, respectively. Executing the next iteration of the outer loop seemingly introduces a discontinuity to this pattern; nonetheless, it is easy to notice that fundamentally the regularity is not violated, but needs to be examined a bit more subtly by taking into consideration the two-level structure of the loop. The steady state loop execution strides as well as the outer loop execution strides in any case are fixed and their values can be easily computed during compile time by analyzing the index expression; in neither case do the multiplcative constants before either subscript effect the computation method, except for altering the constant value of the relevant stride. Consequently, this stride regularity can be easily identified and exploited by simply identifying the direction of the corresponding loop branches. Autonomous generation of these addresses necessitates transmission to the data memory controller of information regarding these strides and the point at which the outer loops are executed. While the effective addresses of these three memory accesses are intertwined on the address bus to the data memory, the order of their appearance is fixed and regular throughout the loop execution, ensuring their easy identification and communication.

Given the application knowledge about these references, i.e., their execution order and address strides, an enhanced data memory controller can utilize this additional information in order to generate the needed addresses internally, obviating completely the processor core responsibility to provide these addresses to the data memory subsystem. The only limited amount of information that the processor needs to send is the type of memory request, and an indication as to whether the innermost loop is in its steady state of execution or an outer loop is to be executed, so that the memory controller can access which set of strides to use for computing the next effective address. Therefore, by utilizing application information within the data memory controller, an efficient address communication scheme can be established, such that the memory controller is able to generate the appropriate effective addresses by itself by utilizing statically obtained information about the load/store order and their strides across the loop dimensions.

### 4.1 Customized communication

The proposed methodology essentially transfers application information regarding the load/store instructions to the data memory controller, so that it can generate effective addresses by itself, thus obviating the need for receiving these addresses from the processor core. The general architecture of the proposed approach is depicted in Figure 12. The programmable interface associated to the data memory, the CDMI, is responsible for generating the addresses of the load/store instructions executed within the application loop. In the case of complete knowledge of the memory reference execution order and their strides, the traffic on the address bus is completely eliminated. The application information is provided to the CPSI and CDMI, as shown in Figure 12, by software before starting the loop execution.

If the CDMI is aware of the load/store execution order, no information whatsoever is needed from the processor to identify the memory reference. Cognizant of the starting memory reference, the CDMI can generate its effective address by adding its stride and automatically updating its state to the next memory operation for the application loop, thus obviating the need to send an address for the next instance of that memory operation.

Attaining such implicit communication of the required data addresses necessitates knowledge of the appropriate strides and of the initial values of the effective addresses for all the load/store instructions. These can be determined at compile time and either stored a priori or sent to the CDMI before entering the loop. A table, traversed sequentially, contains the current effective addresses for the memory reference instructions, while the strides are stored in similar tables. The number of entries corresponds to the total number of memory reference instructions that can be handled by the proposed approach. If any load/stores remain uncovered, they can be executed in their usual way by sending the complete address to the data memory. Each time a load/store instruction is executed, the CPSI sends the stride index and asserts a special additional line denoted as $N_A$ for not an address, associated to the address bus, which signals the CDMI that the data memory request received does not contain an effective address. A fixed number of least significant address bus lines hold the stride index, an index that points to the appropriate stride array corresponding to the loop nesting level. Subsequently, the CDMI utilizes the information about this particular load/store array and computes its effective address by simply adding its current address and stride. The address computed thus is used for accessing the data memory, while the CDMI stores it for utilization in the next loop iteration.

For the proposed customization methodology, we target data intensive loops with load/store effective addresses computed as affine linear functions of the loop indices. During compile-time, the strides for the load/store instructions are computed for subsequent run-time utilization within the CDMI. The compile-time analysis includes the examination of the affine index expressions and the identification of the strides for all the loop dimensions.

### 4.2 Hardware support

Figure 13 represents the architecture of the CDMI module. The Load/Store Table (LST) is used to store the current addresses for the memory reference instructions. An entry in the LST corresponds to a load/store instruction and contains its current address. The LST is addressed by the LST Index (LI) register. Initially, its value is reset to 0, so that it refers to the first entry in the LST, where the address of the first load/store instruction to be executed in the loop is stored. The strides for the memory references are stored in the Stride Arrays (SA). For simplicity, Figure 13 shows a CDMI architecture that supports only two strides (i.e., two loop nesting levels). After reading in the current address from the LST, the stride value is added to it and the resulting address is sent to the data memory for completing the ac-
5. EXPERIMENTAL RESULTS

In our experimental studies we have measured the effectiveness of the proposed approach by observing the reduction of the transitions on the address bus to the data memory. We have utilized seven benchmarks. The first five comprise numerical and DSP kernels: Matrix multiplication (mmul) of matrices with size 256x256; successive over-relaxation (sor) on a matrix with size 256x256; extrapolated Jacobi iterative method (ej) on a 256x256 grid; fast discrete cosine transform (fdct) kernel on a block of data with 8 samples; and the energy loop in fast fourier transform (efft) working on a block of 512 samples. The last two benchmarks are the speech coding (adpcm_enc) and decoding (adpcm_dec) applications [8], selected for their frequent utilization in many voice processing embedded applications.

The second column in Figure 14 reports the total number of transitions in millions on the address bus to the data memory in the general case. It is notable that \( \text{mmul} \) generates, due to its higher execution complexity, a significantly larger number of transitions compared to the rest of the benchmarks. The next column of the table represents the minimized number of transitions after utilizing the methodology we propose. The fourth column shows the total number of memory reference instructions residing in the loop nest and targeted by the proposed methodology. Finally, the last column shows the reduction in percentage of the total number of transitions on the address bus. For the first four benchmarks containing data-only loops, the reductions consistently exceed 99%; this extremely high level of reductions has been anticipated, given the fact that only the two stride indices are sent and changed on the outer loop iterations. Since the \( \text{efft} \) and \( \text{adpcm} \) benchmarks contain a loop with control altering instructions, additional traffic of the LST indices had to be introduced. Three out of all four references in \( \text{adpcm} \) were control independent for both the encoder and the decoder, while for \( \text{efft} \) three out of all eight references were subject to a predicate resolution.

### Table: Data memory address bus transitions

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#TR*10^9</th>
<th>#TR Opt</th>
<th>#Rel</th>
<th>Reduction(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmul</td>
<td>203.06</td>
<td>261,633</td>
<td>3</td>
<td>99.88</td>
</tr>
<tr>
<td>sor</td>
<td>1.56</td>
<td>1,020</td>
<td>5</td>
<td>99.94</td>
</tr>
<tr>
<td>ej</td>
<td>46.91</td>
<td>15,360</td>
<td>11</td>
<td>99.97</td>
</tr>
<tr>
<td>fdct</td>
<td>0.041</td>
<td>256</td>
<td>6</td>
<td>99.37</td>
</tr>
<tr>
<td>efft</td>
<td>0.01</td>
<td>2,039</td>
<td>8</td>
<td>78.67</td>
</tr>
<tr>
<td>adpcm_enc</td>
<td>3.37</td>
<td>738,868</td>
<td>4</td>
<td>78.05</td>
</tr>
<tr>
<td>adpcm_dec</td>
<td>3.26</td>
<td>589,932</td>
<td>4</td>
<td>81.88</td>
</tr>
</tbody>
</table>

Figure 14: Data memory address bus transitions

6. CONCLUSIONS

In this paper, we present a novel, application-specific customization approach for embedded processors. Increasing processor performance and reducing power consumption have been identified as essential goals towards achieving cost-efficient and flexible system implementations. This customization framework uses a novel approach for transferring application information to the processor microarchitecture and exploiting it dynamically. The ability to re-customize the processor microarchitecture in field is a significant advantage that preserves the flexibility of general-purpose processors. The methodology is evaluated on real-life applications and significant performance and power improvement are shown.

Customizing the processor core with application-specific information promises to be a powerful technique towards higher performance and lower power consumption. It allows the processor-centric implementation paradigm and its concomitant advantages to be extended to large classes of complex hardware/software codesign systems implementing various modern applications.

7. REFERENCES