Towards Effective Embedded Processors in Codesigns: Customizable Partitioned Caches*

Peter Petrov
University of California at San Diego
CSE Department
ppetrov@cs.ucsd.edu

Alex Orailoglu
University of California at San Diego
CSE Department
alex@cs.ucsd.edu

ABSTRACT

This paper explores an application-specific customization technique for the data cache, one of the foremost area/power consuming and performance determining microarchitectural features of modern embedded processors. The automated methodology for customizing the processor microarchitecture that we propose results in increased performance, reduced power consumption and improved determinism of critical system parts while the fixed design ensures processor standardization. The resulting improvements help to enlarge the significant role of embedded processors in modern hardware/software codesign techniques by leading to increased processor utilization and reduced hardware cost. A novel methodology for static analysis and a field-reprogrammable implementation of a customizable cache controller that implements a partitioned cache structure is proposed. The simulation results show significant decrease of miss ratio compared to traditional cache organizations.

Keywords: embedded processors, data cache, reprogrammable customizations

1. INTRODUCTION

Embedded processors play a significant role in the modern hardware/software codesign systems. Yet a number of drawbacks, particularly reduced performance and excessive power consumption, haunt processor-centric implementations and limit the applicability of embedded processors, resulting in satisfaction of performance requirements only through unnecessarily large codesigns.

Processor performance, power-consumption, and deterministic execution time play a significant role in modern hardware/software co-design systems [1]. The partitioning of hardware and software is a major issue in these systems [2]. Implementing larger parts of the system as software results in reduced cost, improved time-to-market, system maintainability, and flexibility. However, reduced performance and increased power-consumption are typically observed due to the nature of the general-purpose processors. Yet processor cores with all their attendant benefits of flexibility, maintainability, and high volumes are natural candidates for further utilization, if the performance and power limitation are alleviated. In this paper, we propose a methodology for application-specific customization of the data cache, one of the foremost performance and power determining components of modern, high-end embedded processors. We present an architecture capable of incorporating the application-specific information in a post-manufacturing fashion, utilizing a reprogrammable datapath.

Application-specific customization of embedded processor microarchitectures is a novel technique that transfers application information to the processor microarchitecture. In this case, the microarchitecture can perform informed decisions as to how to handle various architecture-specific actions. Fundamentally, this approach extends the communication link between compiler and processor architecture by transferring application information directly to the microarchitecture without modifying the existent instruction set. Consequently, traditional mainstream compiler algorithms remain unaffected by the proposed customization technique. The static analysis information transfer is accomplished by utilizing a reprogrammable datapath. The reprogrammable implementation we propose allows application changes to be applied in field by loading the new application information, in a manner similar to program reloading.

In this paper, we present a methodology for application-specific customization of the cache subsystem of embedded processors. The problem of high memory latency is a well-known performance bottleneck in modern processors. In the embedded system world, the main memory typically resides on the system bus. This bus is used by various ASIC peripherals that perform system-specific tasks. High system bus utilization due to frequent memory accesses leads to increased bus congestion and reduced data transfer speed between the ASIC peripherals and processor cores, thus hindering overall system performance. Moreover, increased system bus traffic leads to power consumption overhead, which is a significant drawback in embedded systems.

The domain of numerical algorithms, the cornerstone of image and voice processing and various wireless applications, suffers from significantly elevated cache miss rates, as associated algorithms operate on a number of arrays of data with large volume. Consequently, high interference amongst memory references, frequently residing in nested loops, and cache pollution caused by sequential array references are typically observed. New, more sophisticated caching schemes are needed, capable of utilizing application information for the particularities of the specific memory reference pattern.

An algorithm is demonstrated that partitions the memory access instructions of a nested loop into groups. Each group corresponds to a set of load instructions exhibiting data reuse amongst them and is mapped to a *cache partition* within a *partitioned cache* structure. The size of the cache partitions is determined according to the type

*This work is supported through an IBM fellowship.

of reuse and space needed to exploit it. We complement this paper with a discussion of an efficient hardware implementation for partitioned caches. Not only is the proposed implementation efficient but is also reprogrammable, thus providing high flexibility to re-customize the embedded processor in field. Hence, the proposed implementation constitutes a unified microarchitectural solution that is not confined to a particular application, but is capable of handling diverse workloads through in-field re-customization.

2. RELATED WORK

Various techniques have been proposed in the compiler and computer architecture communities to attack the problem of cache conflicts and cache pollution for data-intensive applications. Loop interchange, skewing and tiling [3, 4, 5] all constitute compiler optimization techniques for improving data locality in loop nests. While useful in exploiting a large amount of inherent reuse, controlling the data interference inside the tile is still a significant challenge [6]. Loop transformations introduce extra control code as well, which may preclude their usage in applications with stringent time and power budgets.

Architectural support for distinguishing between memory references that exhibit spatial, temporal or no reuse whatsoever, has been proposed through the introduction of the dual data cache [7]. The approach results in improved cache utilization, but the cache interference problem still remains. A static analysis approach for avoiding data cache interference is presented in [8]. Therein, memory references that can cause cache conflicts are annotated as non-cacheable. Additionally, a cache volume analysis for facilitating the feasibility of exploiting particular data reuse is described in [8].

A reconfigurable cache design has been presented in [9]. This approach allows the cache array to be divided dynamically into partitions that can be utilized by the processor for various purposes. In a similar vein, [10] proposes a cache organization that dynamically partitions and shuts down parts of the cache according to application demand, hence trading off performance for reduced power consumption.

3. MOTIVATION

A typical numerical 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 $PSI[i, j+1]$ and $PSI[i+1, j+1]$. These references reuse a row from the matrix PSI along the outer loop i.

```
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;
 endfor
endfor
```

Figure 1: An excerpt from the *swim* benchmark.

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 arrays and cache pollution is introduced. Caching the data intelligently by avoiding interferences and minimizing the effective cache size needed to exploit the existent reuse is a rather difficult task for a general-purpose cache controller.

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.

On the other hand, the group of references to 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. Due to interference and volume limitations, a conventional cache organization fails to exploit this reuse. The significant miss rates for the *swim* benchmark for a typical direct-mapped cache [8] are a salient indicator of such limitations.

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 obliterated and effective cache volume needed to exploit a reuse would be reduced. By partitioning the cache in this way, significant amounts of interference are avoided. The size of each cache partition can thus be reduced to no larger than the minimal sufficient size for exploitation of the inherent reuse for that particular group of instructions. The combination of a novel compile time analysis together with reprogrammable, partitioned cache architectures provides an effective solution for alleviating the aforementioned problems.

4. REFERENCE ANALYSIS

The proposed partitioning analysis utilizes information about the type of reuse exhibited by each reference. Analysis of the inherent reuse in a loop nest can be used to group memory instructions and to associate each group to a cache partition. 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.

A memory reference instruction is said to have *self-temporal* (*st*) reuse if in a later loop iteration it accesses the same memory address. A *self-spatial* (*ss*) reuse refers to an instruction that accesses data inside a single cache line in two subsequent loop iterations. Two load/store instructions are said to have *group-temporal* (*gt*) reuse if both of them access the same memory address; they are denoted as *group-spatial* (*gs*) if both access memory addresses that map to the same cache line.

The type of reuse can vary across loop dimensions. A reuse occurs in a particular loop dimension and is qualified by the number of iterations within which the reuse is exploited. Figure 2 shows the reuse types for all memory access instructions in the example in Figure 1.

The reference analysis that we present in this section works on a loop nest and it assumes that all the references to a data array in a loop nest have indices with the same multiplicative coefficient (for example, references A[i] and A[i+3], but not A[i] and A[3*i+2]). In the case of differing multiplicative coefficients, the corresponding load/store instructions are not targeted for partitioning and are

Memory Instruction	Iteration i	Iteration j
U[i,j+1]	no reuse	ss
V[i+1,j]	no reuse	ss
PSI[i,j+1]	gt	ss
PSI[i+1,j]	gt	ss, gt
PSI[i+1,j+1]	gt	ss, gt

Figure 2: Data reuse

```
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]);
endfor
```

Figure 3: Example of group temporal reuse.

cached into a special cache partition that is dedicated for all unpartitioned references. In any case, references of the latter type are in practice quite uncommon. The identical multiplicative coefficients, the inherent characteristics of the partitioning algorithm we present, and the usage of a write-through caching policy eliminate the need for data consistency considerations.

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 to an already formed group of load/store instructions. The edges in the DRG represent data reuse between the corresponding nodes. A connected component in the DRG corresponds to a set of memory references to a particular array. Each edge is annotated with the particular types 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 type of reuse. It is evident that a self-spatial reuse necessitates only a single cache line. It can be shown that a group-spatial reuse requires only one cache line as well by noting that it is equivalent to a group-temporal reuse with the distance between the referred data addresses being less than a cache line size. In the case of temporal reuse (self or group) though, a fixed (but varying amongst memory instructions) number of cache lines are needed in order to exploit the reuse. More generally, the cache volume (number of cache lines) needed to exploit a group-temporal data reuse (denoted $CV^{(g)}$) is calculated by formula (1), with l denoting the cache line size, k the number of loop iterations in which the reuse occurs, j the loop dimension, and d_i the size of the i^{th} loop dimension. Equation (2) defines the cache volume for self-temporal reuse (denoted $CV^{(s)}$).

$$CV^{(g)}(k,j) = \frac{1}{l}(k+1)\prod_{i<j} d_i \quad (1)$$

$$CV^{(s)}(j) = \frac{1}{l}\prod_{i<j} d_i \quad (2)$$

For example, the pair of loads A[i] and A[i+k], representing array traversal with loop index i, requires k/l cache lines to exploit the group-temporal reuse between these references. If the reuse occurs in the outer loop iterations, such as $PSI[i+1, j+1]$ and $PSI[i, j+1]$ from the example in Figure 1, all data referred along iteration j by both instructions need to be preserved. Intuitively, this corresponds to keeping the $PSI[i]$ row in the cache while $PSI[i+1, j+1]$ is "pre-fetching" the next row.

The main objective of the partitioning algorithm is to group load and store instructions with data reuse amongst them and map this group to a cache partition with appropriate size. The cache is direct-mapped and virtually partitioned into subcaches. Since the goal of the cache partitioning is avoiding interference, no set associativity is required within the partitions.

From a DRG perspective, the suggested methodology implies grouping of connected nodes together in a partition with no unconnected nodes present; interference and cache pollution is thus prevented. Since the goal is to exploit as much as possible of the existing data reuse, the number of edges from the DRG covered by partitions needs to be maximized. The total amount of available cache memory in a particular processor implementation cannot be exceeded in the process. Since every data reuse (edge in DRG) has a particular requirement for cache volume, the partitioning algorithm needs to judiciously select edges so as to meet the aforementioned goals and constraints.

The algorithm starts from the innermost loop dimension, as the frequency of the data reuse there is maximal (equal to the product of the size of all loop dimensions). The data reuse frequency wanes as the traversal proceeds towards the outermost dimension. Therefore, the algorithm proceeds iteratively on the loop dimensions starting from the innermost loop. It commences forming partitions of single nodes with a self-spatial reuse edge or by combining a pair of nodes. Spatial reuse requires a minimum of cache volume, i.e. one cache line. For the group-temporal reuse, which is typically the prevalent reuse, more sophisticated approaches are needed as shown below.

An example of group-temporal reuse can be seen in Figure 3. We assume a cache line size of one word to simplify the explanation. Figure 4a shows the DRG for the example in Figure 3. The number of cache lines needed to capture the corresponding group reuse is shown. If we assume an available cache volume of 9 cache lines, a direct greedy approach leads to the result shown in Figure 4b. The solution produced consists of three partitions, covering only three edges with a total of 7 cache lines used. 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. It covers four temporal reuses in A and utilizes all 9 cache lines exactly. The latter solution evidently constitutes an improvement, as a combination of two edges with a common node in a single partition "saves" the storage of the overlapping node. As the counterexample illustrates that the straightforward, greedy approach is inadequate in attaining a consistently optimal result, we proceed to outline an improved model of representation that relies on separating the edges in the DRG and updating their CVs and overlap information. The overlap is one cache line for the innermost loop dimension or a unit cache volume for the outer loop dimensions.

When the algorithm proceeds onto the next (outer) loop dimension, the overlap between the nodes in the DRG corresponds to the size of the previous loop dimension, i.e. the memory volume needed for an iteration of the inner loop dimension. In terms of matrix traversal, the overlap corresponds to one row from the matrix as can be seen in Figure 4c. The algorithmic framework can be simplified if we work with a unit overlap system throughout; consequently, at every level, CV values are presented as multiples of the overlap at this level. Of course, these values will have to be

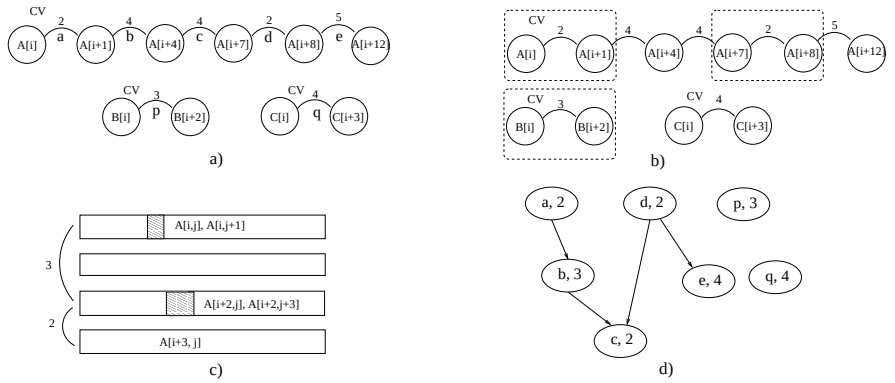

Figure 4: DRG and ODT for the group-temporal reuse example.

retranslated to actual cache lines for cache allocation accounting. We present an efficient algorithm that quickly finds an optimal solution for a loop dimension with unit overlap between the nodes in the DRG. As a reference we use the innermost loop dimension for which the overlap is exactly one cache line. The partitioning methodology consists of two steps:

Eliminating the overlap between edges with a common node in DRG. In this phase, the algorithm separates the connected edges while keeping track of the overlap by building Overlap Dependence Trees (ODT) as shown in Figure 4d. 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.

```
repeat
  e=FindEdge with min CV in DRG;
  RemoveEdgeFromDRG(e);
  Decrement CV of Neighbours(e);
  ODT(e)=CreateNode in ODT for e;
  for all n in Neighbours(e) and n in ODT
     Connect ODT(e) as a child of ODT(n);
until no edges left in DRG
```

If there is a tie of the minimal CV values in the DRG, then reduced priority is given to the edges on the boundary of the reuse chain. A random pick otherwise is guaranteed to preserve overall optimality. The ODT for the example in Figure 3 is shown in Figure 4d. Each node in the ODT is annotated with the updated CV.

Constructing the optimal solution. Given the ODT, the purpose of the algorithm is to find the maximum subset of nodes subject to constraints that the total available cache volume not be exceeded and that overlap dependences be preserved. The following pseudocode defines this part of the algorithm.

```
CacheVolume=TotalCacheVolume;
repeat
  V={v1,...,vn}=FindNodes with min CV in ODT;
  for all vi in V & vi is a root
    if CV(vi)>CacheVolume then exit;
    Select vi; ODT=ODT-{vi}; V=V-{vi};
  endfor
  repeat
    Find vk in V with minimal
     number of ancestral nodes;
    for w in Path to vk from the root
      if CV(w)>CacheVolume then exit;
      Select(w); ODT=ODT-{w}; V=V-{w};
    endfor
  until V is empty
until CacheVolume<0
Form partitions from the selected nodes;
```

Selecting the root from the tree that contains a minimal node with the smallest number of ancestors maintains optimality at this instant while paving the route for eventually incorporating the truly minimal CV node into the partition. The algorithm terminates when there remains insufficient cache volume to accommodate the next data reuse.

When proceeding to the outer loop dimensions though, the normalized overlap might not be exactly a unit, as partitions may exist in the inner loop dimension that are already accounted for. However, the partitions formed in the inner loops are typically much smaller in size compared to the new dimension overlap unit, thus resulting in a normalized overlap ratio quite close to unity. For example, a few cache lines that have formed a partition for the innermost loop iteration of a matrix traversal algorithm pale in comparison (of course only in size, but not in utility) to the number of cache lines needed to cover a row from the matrix. The normalized unit assumption, with insignificant variation for outer loops and exact for the most useful innermost loop, ensures practical optimality, consequently.

5. IMPLEMENTATION

The proposed partitioning methodology requires special hardware support from the cache controller. The hardware needs to be able to capture the information provided by the compiler about load/store instruction partitioning and to effectively map these references to the corresponding part of the partitioned cache.

The cache is virtually partitioned into sub-caches, each of them accommodating groups 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.

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 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

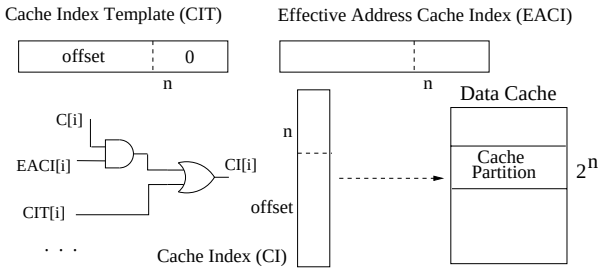

Figure 5: Partition cache index calculation

of this constant determines the offset of the cache partition inside the original cache as can be seen in Figure 5.

The above reasoning evinces that each cache partition with size 2^n is identified by a pair of numbers {*offset*, *cache partition index size*(n)}. The *cache partition index* is formed by concatenating the *offset* and the n least significant bits from the cache index.

Fundamentally, the hardware support for a partitioned cache has to resolve two problems: identify the mapping between a load/store instruction and a particular cache partition, and calculate the *cache partition index* using the pair of numbers identifying the partition.

These goals are achieved by a hardware architecture utilizing two tables: the *Partition Mapping Identification Table* (PMIT) and the *Partition Identification Table* (PIT). PMIT identifies the load/store instructions and their corresponding partitions. The PMIT structure can be implemented in various ways. One alternative is to have associative lookup. Each entry from the *PMIT* in this case will contain two fields: the *program counter* of the load/store instructions, and a *partition index* field - an index that points to the *PIT*, which defines the cache partitions. Another alternative is to have a larger PMIT that will contain an entry for each possible instruction in the loop-nest. The entries corresponding to partitioned load/store instructions will contain an index into the PIT. Hardware cost and power consumption must be considered in choosing the appropriate implementation between the two suggested PMIT forms. The partition information is stored in the PIT. Each entry in the PIT contains the pair of numbers that define a particular cache partition. The information from this table is used to calculate the partition cache index. PMIT and PIT contents are loaded into the processor at the same time that the code of the embedded application is stored in the main memory.

The lookup into 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 5 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. The control signals $C[i]$ are 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 5. Only the delay of the two gates shown, insignificant for the datapath of the cache access path, is added to the delay of accessing a traditional data cache.

The cache partitions behave as stand-alone caches. Since a particular cache line can participate in various cache partitions for different loop nests, the length of the needed tag associated with every cache line will vary. In the general case, only a fraction of the original *cache index* from the data address is used to calculate the partition cache index. A conservative solution would be to increase the size of all cache tags with the *cache index* bits. An alternative solution is to provide varying tag sizes across the cache lines. For cache lines accommodating partitions of size 2^n and higher, the tag is extended with only $bitwidth(CacheIndex) - n$ bits to cover the unused bits from the original cache index part of the address.

The proposed methodology works on an individual loop nest level. A typical numerically intensive embedded application contains multiple loop nests. In order to be able to perform the cache partitioning methodology for all loop nests, multiple partition mappings associated with each loop nest have to be stored. A straightforward solution is to have multiple PMITs. Each time the application finishes with one loop nest and proceeds to another, a switch between the PMITs needs to be performed. During the switch, the cache content has to be invalidated because a new mapping between load/store instructions and cache partitions is defined. The switch of partition mappings when proceeding to different loop nests can be controlled by software using a special control register that determines the active partition mapping. Another alternative is to have a system level controller that switches between different partition mappings according to the current application (or function within an application), thus dynamically reconfiguring the cache subsystem. The number of PMITs supported determines the number of loop nests that can be handled. Usually this number is small and limiting it to the range of 5-8 will practically satisfy any real application.

6. EXPERIMENTAL RESULTS

We evaluate and analyze experimentally the ability of the partitioned cache to reduce the total number of data cache misses. The miss-ratio and total number of misses are examined and compared. In this experimental study, we use benchmarks from the floating-point SPEC95fp suite and a number of frequently used numerical kernels. Significant improvements are suggested for large classes of data intensive applications as the numerical kernels used are frequently utilized parts of them.

We compare the performance of the partitioned cache structure against a number of typical L1 cache configurations in modern embedded processors. Specifically, our comparison configurations include: 4K direct-mapped and 2-way set-associative L1 cache with line size of 4 words, 8K direct-mapped and 2-way set-associative L1 cache with line size of 8 words. We include in our comparative study 2-way set-associative cache configurations, as increased cache associativity is a classical approach for reducing cache conflicts. Two configurations for a partitioned cache are examined: 4K direct-mapped with line size of 4 words, and 8K direct-mapped with line size of 8 words.

The SimpleScalar toolset [11] has been used to examine the cache behavior for the baseline cache architectures. The partitioning algorithm presented in Section 4 has been performed on the source-code level and each load/store reference has been associated to a particular partition. The assembly code has been instrumented and the cache simulator modified so as to take advantage of this partitioning information. The size of the cache partitions is provided as separate information in a configuration file and the partitioned cache is modeled and simulated.

Six benchmarks are used in our experiments: *Matrix multiplication (mmul)* of matrices with size 256x256; *Matrix inversion (minv)* of a 128x128 matrix; *LU decomposition (lu)* [12] on a matrix with

	4K DM p-cache		8K DM p-cache	
	#Misses	MR	#Misses	MR
mmul	4,227,938	6.29%	2,113,983	3.14%
minv	4,394,798	17.06%	1,342,947	5.21%
lu	1,419,613	6.35%	718,029	3.21%
ej	1,479,593	16.62%	617,945	6.94%
tri	806,256	26.01%	789,412	25.47%
swim	5,280,121	14.82%	3,209,979	9.01%

Figure 6: Partitioned cache results

size 256x256; *Extrapolated Jacobi-iterative method (ej)* [12] on a 128x128 grid; *Tri-diagonal system solver (tri)*, a fundamental part of the *tomcatv* SPEC95fp benchmark and a major contributor to the high miss rate for the *tomcatv* benchmark, with matrix size of 256x256; *Swim benchmark (swim)*, part of the SPEC95fp benchmark suit, characterized by a high cache miss-ratio due to a large amount of interference [8].

The total number of partitions for the examined benchmarks is relatively small ranging from 3 for *mmul* to 25 for *swim*. Figure 7 depicts a graphical comparison for the miss-ratio of the base configurations and the partitioned cache configurations. For *mmul*, *minv*, and *lu*, the 2-way set associative cache leads to a significant decrease in the number of misses. Yet, for *tri* and *swim*, set-associativity does not help to the same extent due to the high level of cache pollution for these benchmarks. Figure 6 shows the results for the partitioned cache. A significant improvement in the miss-ratio can be observed for *minv*, *tri*, *ej* and *swim* compared to both the DM and the 2-SA caches from the base configurations. The *mmul* and *lu* benchmarks show significant improvements compared to DM base configurations, while for 2-SA, *mmul* exhibits a relatively small decrease in the miss-rate and the miss-ratio for *lu* is unchanged for the 2-SA cache configurations. This small decrease in miss-rate is due to the relatively small amount of interference in these applications, which can be accommodated to some extent with increased associativity. As expected from the theoretical analysis of the partitioning algorithm, the miss-ratio is significantly reduced for applications with large working sets and high amount of cache pollution and interference.

7. CONCLUSION

We have presented a novel methodology for application-specific customization of the cache subsystem of embedded processors in this paper. A precise static analysis of the application has been demonstrated to be capable of identifying the optimal solution for grouping memory access instructions and mapping them to cache partitions with optimal size. Preventing cache interference and cache pollution by utilizing precise application information have been the main objectives of the proposed methodology. The achievement of these goals has been confirmed by extensive experimental results. A significant increase in the cache hit rate has been demonstrated by a representative set of simulation results. The proposed technique has significant implications in SOC designs utilizing embedded processor cores, as it significantly reduces the number of system bus transactions, thus resulting in higher system performance and reduced power.

Customizing the embedded processor architecture utilizing a reprogrammable hardware promises to be a powerful technique towards lower power consumption, and higher and deterministic performance in hardware/software systems. At the same time, it retains the processor-centric paradigm and extends its advantages to a large class of modern codesign applications.

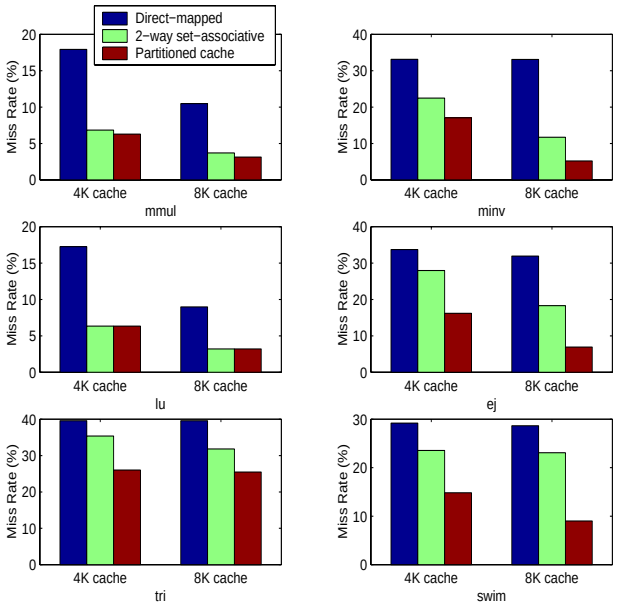

Figure 7: Comparative benchmark miss ratios

8. REFERENCES

[1] W. H. Wolf, "Hardware-Software Co-Design of Embedded Systems", *Proceedings of the IEEE*, vol. 82, n. 7, pp. 967–989, July 1992.

[2] J. Henkel and R. Ernst, "A Hardware/Software partitioner using a dynamically determined granularity", in *34th DAC*, pp. 691 – 696, June 1997.

[3] M. S. Lam, E. E. Rothberg and M. E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms", in *4th ASPLOS*, pp. 63–74, April 1991.

[4] M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm", in *PLDI*, pp. 30–44, June 1991.

[5] M. J. Wolfe, "More Iteration Space Tiling", in *Supercomputing*, pp. 655–664, November 1989.

[6] O. Temam, C. Fricker and W. Jalby, "Cache awareness in blocking techniques", in *Journal of Programming Languages*, 1998.

[7] A. Gonzalez, C. Aliagas and M. Valero, "A data cache with multiple caching strategies tuned to different types of locality", in *International Conference on Supercomputing*, pp. 338–347, July 1995.

[8] J. Sanchez, A. Gonzalez and M. Valero, "Static Locality Analysis for Cache Management", in *PACT*, pp. 261–271, November 1997.

[9] P. Ranganathan, S. Adve and N. P. Jouppi, "Reconfigurable Caches and their Application to Media Processing", in *27th ISCA*, June 2000.

[10] D. H. Albonesi, "Selective Cache Ways: On-Demand Cache Resource Allocation", in *32nd MICRO*, pp. 248–259, November 1999.

[11] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0", Technical Report 1342, University of Wisconsin-Madison, CS Department, June 1997.

[12] S. Nakamura, *Applied Numerical Methods with Software*, Prentice-Hall, Englewood Cliffs, N.J., 1991.