Using Data Compression for Increasing Memory System Utilization

Ozcan Ozturk, *Member, IEEE*, Mahmut Kandemir, *Member, IEEE*, and Mary Jane Irwin, *Fellow, IEEE*

***Abstract*—The memory system presents one of the critical challenges in embedded system design and optimization. This is mainly due to the ever-increasing code complexity of embedded applications and the exponential increase seen in the amount of data they manipulate. The memory bottleneck is even more important for multiprocessor-system-on-a-chip (MPSoC) architectures due to the high cost of off-chip memory accesses in terms of both energy and performance. As a result, reducing the memory-space occupancy of embedded applications is very important and will be even more important in the next decade. While it is true that the on-chip memory capacity of embedded systems is continuously increasing, the increases in the complexity of embedded applications and the sizes of the data sets they process are far greater. Motivated by this observation, this paper presents and evaluates a compiler-driven approach to data compression for reducing memory-space occupancy. Our goal is to study how automated compiler support can help in deciding the set of data elements to compress/decompress and the points during execution at which these compressions/decompressions should be performed. We first study this problem in the context of single-core systems and then extend it to MPSoCs where we schedule compressions and decompressions intelligently such that they do not conflict with application execution as much as possible. Particularly, in MPSoCs, one needs to decide which processors should participate in the compression and decompression activities at any given point during the course of execution. We propose both static and dynamic algorithms for this purpose. In the static scheme, the processors are divided into two groups: those performing compression/decompression and those executing the application, and this grouping is maintained throughout the execution of the application. In the dynamic scheme, on the other hand, the execution starts with some grouping but this grouping can change during the course of execution, depending on the dynamic variations in the data access pattern. Our experimental results show that, in a single-core system, the proposed approach reduces maximum memory occupancy by 47.9% and average memory occupancy by 48.3% when averaged over all the benchmarks. Our results also indicate that, in an MPSoC, the average energy saving is 12.7% when all eight benchmarks are considered. While compressions and decompressions and related bookkeeping activities take extra cycles and memory space and consume additional energy, we found that the improvements they bring from the memory space, execution cycles, and energy perspectives are much higher than these overheads.**

***Index Terms*—Compilers, data compression, embedded systems, memory optimization, multiprocessor-system-on-a-chip (MPSoC).**

Manuscript received October 16, 2008; revised January 6, 2009. Current version published May 20, 2009. This work is supported in part by NSF Grants 0811687, 0720645, 0720749, 0702519, a grant from Microsoft Research and a grant from GSRC. This paper extends the material presented in GLSVLSI'05 [1] and ASPDAC'06 [2] by giving more detailed information about the algorithms and by presenting an experimental analysis of the proposed approach. This paper was recommended by Associate Editor E. Martin.

O. Ozturk is with the Department of Computer Engineering, Bilkent University, Ankara 06800, Turkey (e-mail: ozturk@cs.bilkent.edu.tr).

M. Kandemir is with the Microsystems Design Laboratory, Computer Science and Engineering Department, The Pennsylvania State University, University Park, PA 16802 USA.

M. J. Irwin is with the Computer Science and Engineering Department, The Pennsylvania State University, University Park, PA 16802 USA.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2009.2017430

I. Introduction

MOST EMBEDDED systems have very tight constraints on memory space, power consumption, and performance. In particular, memory constraints are getting increasingly important as both the code complexity of embedded applications and the amount of data they process are increasing. While it is true that the memory capacity of embedded systems is continuously increasing, the increases in the application complexity and data-set sizes are far greater. In addition, many parallel embedded systems execute multiple applications simultaneously, which puts additional pressure on the on-chip memory capacity. For example, many commercial architectures today employ scratch-pad memories (SPMs). However, since an SPM is very small in size when compared to main memory and is shared by multiple data sets (e.g., different arrays) simultaneously, in many cases, some important (critical) data blocks are still left in the off-chip memory. Therefore, for many large data-intensive embedded applications, taking full advantage of the SPM is not possible. As a result, memory-space utilization remains one of the most pressing challenges for many embedded execution environments, and there is a growing need for techniques that make best use of the available memory space without hurting application performance significantly. Prior research on memory systems proposed and evaluated several techniques, which can potentially improve the memory performance of embedded software. Power and memory-space efficiency, on the other hand, have received relatively less attention so far.

One of the techniques that can be used to reduce the memory-space consumption (occupancy) of embedded applications is data compression. The goal of data compression is to represent an information source (e.g., a data file, a speech signal, an image, or a video signal) as accurately as possible using the fewest number of bits. Previous research considered efficient hardware- and software-based data-compression techniques and applied compression to different domains. While data compression can be an effective way of reducing the memory-space consumption of embedded applications, it needs to be exercised with care since performance and power costs of decompression (when we need to access data stored currently in the compressed form) can be overwhelming. Therefore, compression/decompression decisions must be made based on a careful analysis of the data access patterns of the application.

Our first goal in this paper is to study how automated compiler support can help in deciding the set of data elements to compress/decompress and the points during execution at which these compressions/decompressions should be performed. The proposed compiler support, targeting single-core-based systems, achieves this by analyzing the source code of the application to be optimized and identifying the order in which different data blocks are accessed. Based on this analysis and data-reuse information, the compiler then automatically inserts explicit compression and decompression calls in the application code. The compression calls target the data blocks that are not expected to be used in the near future, whereas the decompression calls target those data blocks with expected reuse but currently in compressed form. We discuss our compiler algorithm in detail and present an example that illustrates how it operates in practice. In providing automated compiler support for data compression, we want to achieve two objectives. First, we want to enable memory-space savings without the involvement of the application programmer, i.e., the programmer gets the benefits of reducing the memory-space consumption without any major effort on her part. Second, we want to minimize the potential impact of data compression on performance. To do this, our automated approach makes use of data-reuse analysis.

We then extend our data-compression approach to multiprocessor-system-on-a-chip (MPSoC) architectures. In the context of MPSoCs, our goal is to schedule compressions and decompressions such that they do not conflict with parallel application execution. Specifically, one needs to decide which processors should participate in the compression and decompression activities at any given point during the course of execution. This is because allowing each processor to perform compression and decompression (in addition to application execution) would put compression and decompression into the critical path execution, thereby affecting the performance of parallel execution significantly. We propose two different strategies for such a division: *static* and *dynamic*. In the static scheme, the processors are divided into two groups: those performing compression/decompression and those executing the application, and this grouping is maintained throughout the execution of the application (i.e., it is fixed). In the dynamic scheme, on the other hand, the execution starts with some grouping, but this grouping changes during the course of execution (i.e., it adapts itself to the dynamic requirements and data access pattern of the application being executed). This is achieved by keeping track of the wrongly done compressions at runtime and adjusting the number of processors allocated for compression/decompression accordingly. In the second part of this paper, our main goal is to explore these two processor space partitioning strategies, identify their pros and cons, and draw conclusions.

Our experimental results show that the proposed approach reduces both maximum and average memory occupancies significantly. In addition, we show that it can save more memory space than an alternate compiler technique that exploits lifetime information to reclaim the memory space occupied by dead data blocks, i.e., the data blocks that completed their last uses. Our experimental results obtained by simulating the execution of an MPSoC indicate that the most important problem with the static scheme is one of determining the ideal number of processors that need to be allocated for compression/decompression. The results also show that the dynamic scheme successfully modulates the number of processors used for compression/decompression according to the dynamic behavior of the application, and this, in turn, improves overall performance significantly.

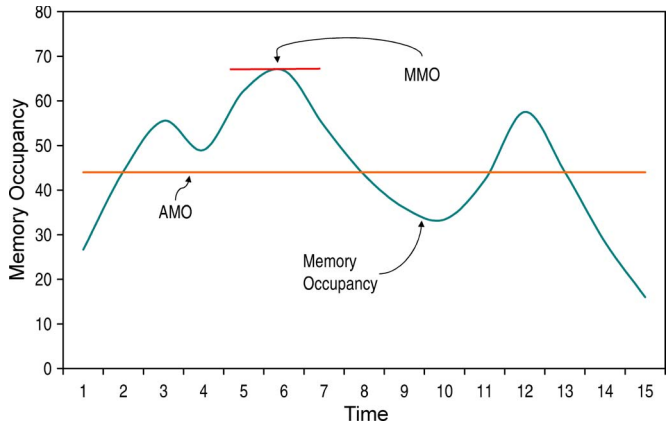

Fig. 1. Example curves for MMO and AMO.

II. Related Work

To reduce the size of a program's code segment, Cooper and Harvey [3] use pattern-matching-based techniques. A reduced instruction set computer system that can directly execute compressed programs is presented in [4]. Very long instruction word processors are also considered in compression-related work [5]–[9]. Data compression is used to reduce storage requirements, bus bandwidth, and energy consumption [10]–[19]. Using a hardware compressor enables faster compression/decompression, and this has been achieved by various techniques [10], [20]–[22]. Our work is different from these prior efforts as we give the task of the management of the compressed data blocks to the compiler. In deciding the data blocks to compress and decompress, our compiler approach exploits the data-reuse information extracted from the array accesses in the application source code. Prior work on MPSoCs discusses several advantages of these architectures over complex single-processor-based designs [23]–[30]. Using compression in an MPSoC environment has been investigated by Alameldeen and Wood [31]. In this paper, the authors introduce an adaptive policy that dynamically adapts to the costs and benefits of cache compression. In [32], this work has been extended to include an adaptive prefetching mechanism. Our work is different from these prior efforts on MPSoCs as it explores the potential of compression in parallel execution.

III. Memory-Space Occupancy

Memory-space occupancy indicates the memory space occupied by application data at each point during the course of execution. There are two important metrics associated with memory-space occupancy. The first one is the *maximum memory occupancy* (MMO), which gives the maximum memory space occupied by data during the course of execution when considering all execution cycles. The second metric is the *average memory occupancy* (AMO), which gives the memory occupancy when averaged over all execution cycles. Fig. 1 shows these two metrics for an example case. Note that the

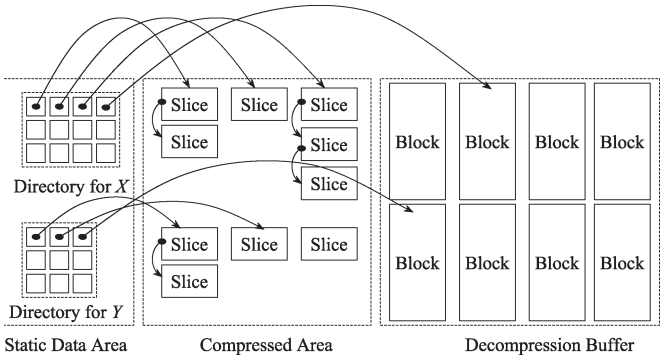

Fig. 2. Memory organization. Our compiler creates a directory with entries, each of which corresponding to a tile of array X. Each entry in the directory of array X (denoted as $X[\![\vec{I}]\!]$) contains a pointer to the memory location where the corresponding tile is stored. By using this pointer, we are able to access the correct tile (could be compressed or decompressed).

drops in the memory occupancy curve indicate either some application-level dead memory block recycling or system-level garbage collection. Both these metrics, MMO and AMO, can be important targets for optimizations. MMO is critical to minimize as it captures the amount of memory that needs to be allocated for the application if the application is to run successfully without an out-of-memory exception. The AMO metric, on the other hand, can be important in a multiprogramming-based embedded execution environment where multiple applications compete for the same memory space. In particular, in an MPSoC setting, saved memory space can be used to increase the number of parallel applications. The goal behind our compiler-directed approach is to reduce both MMO and AMO for array/loop-intensive embedded applications. Note that, array/loop-intensive applications are frequently used in embedded image/video processing [33]. Furthermore, reducing MMO and AMO can result in both performance and energy improvements.

IV. Data Compression in the Single-Core Case

A. Compiler Algorithm

Employing data compression in managing the memory space of an embedded system requires a careful analysis of the data access pattern of the application under consideration. This is because exercising data compression in an untimely manner can cause significant performance and power penalties. For example, compressing data blocks with short reuse distances can increase the number of decompressions dramatically. Furthermore, decompressing data blocks with long reuse distances prematurely can increase memory-space consumption unnecessarily. Therefore, one needs to be very careful in deciding both the set of data blocks to compress/decompress and the points in execution to compress/decompress them. Clearly, this is an area that can benefit a lot from automation.

1) Data Tiling and Memory Compression: Our scheme compresses only the arrays that can benefit from data compression (this can be determined either through profiling or via programmer annotations). These arrays are referred to as the "compressible" arrays in this paper. We do not compress scalar variables or incompressible arrays, i.e., the arrays that cannot benefit from data compression. Fig. 2 shows the organization

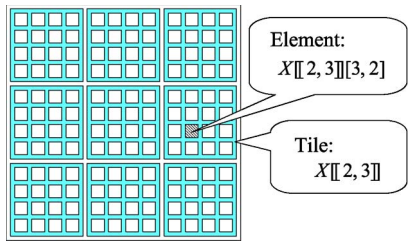

Fig. 3. Data tiling for array X.

of the memory space for supporting our compiler-directed data-compression approach. We divide the memory space into three parts: *compressed area, decompression buffer, and static data area.* The static data area contains scalar variables, incompressible arrays, and the directories for compressible arrays. The data entities in the static area are statically allocated at compilation time. The compressed area and the decompression buffer, however, are dynamically managed at runtime based on the compiler-determined schedule for compressions and decompressions. Note that, as can be seen from Fig. 2, both compressed arrays/tiles and decompressed arrays/tiles can be present at the same time. Even some blocks of a given array can be compressed while some other blocks are decompressed.

We expect our technique to be more successful in the context of embedded systems where data sets are mostly integer or fixed point (instead of floating point). It needs to be noted, however, that our approach can work with data with different compressibilities, as we manage the memory space at a tile granularity (i.e., a less compressible data set will occupy more tiles). Although, in theory, the compressibility of a data block can change during the course of execution of the program (due to updates), we found in our experiments that this happens very rarely in practice.

We divide each compressible array into equal-sized *tiles* (*blocks*). An element of a tiled array X can be indexed using the following expression: $X[\![\vec{I}]\!][\vec{J}]$, where $\vec{I}$ is the tile subscript vector, which indexes a tile of array X, and $\vec{J}$ is the intratile subscript vector, which indexes an element within a given tile. For example, Fig. 3 shows an array X that is divided into nine (3×3) tiles, and each tile contains sixteen (4×4) elements. As an example, $X[\![2,3]\!]$ refers to the tile at the second row, third column, and $X[\![2,3]\!][3,2]$ refers to the data element at the third row, second column of tile $X[\![2,3]\!]$.

Fig. 2 shows how we store the tiled arrays in the memory. For each compressible array X, our compiler creates a directory, each entry of which corresponding to a tile of array X, and can be indexed using a tile subscript vector. Each entry in the directory of array X, denoted as $X[\![\vec{I}]\!]$ ($\vec{I}$ is a tile subscript vector), contains a pointer to the memory location where the corresponding tile is stored. The directory of each array is stored in the static data area of the memory space.

An array tile can be either compressed or uncompressed. Uncompressed tiles are stored in the decompression buffer, and compressed tiles are stored in the compressed area. The decompression buffer is divided into equal-sized blocks, and the size of a block is equal to that of a tile. We use a *free table* to keep track of the free blocks in the decompression buffer. When we need to decompress a tile, we first need to allocate a free block in the decompression buffer. Compressed tiles are stored

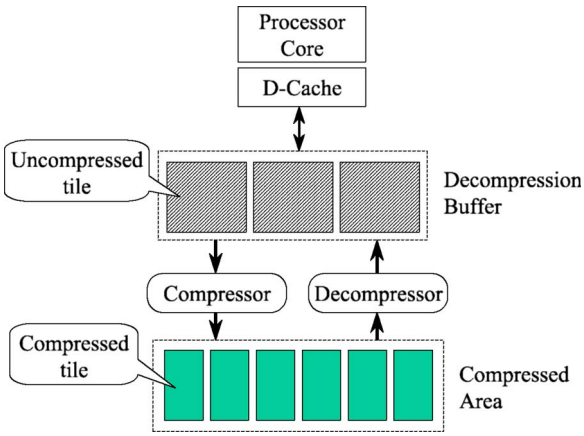

Fig. 4. Architecture supporting memory compression.

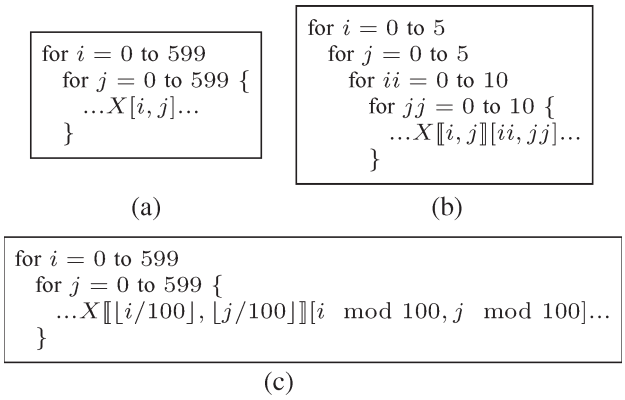

Fig. 5. Code transformation for data tiling and loop tiling. (a) Original loop nest. (b) With data tiling only. (c) With both data tiling and loop tiling.

in the compressed area. The memory in this area is divided into equal-sized *slices*. The size of a slice is smaller than that of a block in the decompression buffer. In our implementation, a slice is equal to a quarter of a block. Although the size of tiles is fixed, the compression ratio depends on the specific tile. Therefore, the number of slices required to store a compressed tile may vary from one tile to another. In Fig. 2, we can see that the slices belonging to the same tile form a link table. As in the case of the decompression buffer, the compressed area also has a free table keeping all free slices.

Fig. 4 shows the architecture of our system. When the program starts its execution, all tiles are in the compressed format and are stored in the compressed area. A compressed tile is decompressed and stored in the decompression buffer by a *decompresser* before it is accessed by the program. If this tile belongs to an array that is not written (updated) by the current loop nest, the compressed version of this tile remains in the compressed area. On the other hand, if this tile belongs to an array that might be written by the current loop nest, we discard the compressed version of this tile and return the slices occupied by this tile to the free table of the compressed area. When we need to decompress a new tile but there is no free space in the decompression buffer, we select a set of old tiles in the decompression buffer and discard them to make space for the new tile. If a victim tile (the tile to be evicted) belongs to an array that might be written by the current loop nest, we must decompress and store its compressed version in the compressed area before we evict its uncompressed version. On the other hand, if this tile belongs to an array that is not written by the current loop nest, we can discard the uncompressed version of this tile without recompressing it. The important point to note is that our approach is not tied to any specific compression/decompression algorithm, and the compressor and decompresser can be implemented either in software or hardware. In our current implementation, however, we use only software-based compression/decompression.

It should be emphasized that data tiling is required by our implementation of memory compression, not by the logic of the application. Therefore, we do not require the programmer to be aware of the data tiling performed transparently for enabling data compression. Our compiler automatically (i.e., in a user-transparent manner) tiles every array that needs to be compressed. Data tiling requires two mapping functions p and q that map an original array subscript vector to tile and intratile subscript vectors, respectively. That is, we map $X[\vec{I}]$ into $X[\![p(\vec{I})]\!][q(\vec{I})]$. In this paper, given a tile size T, we use the following mapping functions:

$$p((i_1, i_2, \ldots, i_n)) = (\lfloor i_1/N_1 \rfloor, \lfloor i_2/N_2 \rfloor, \ldots, \lfloor i_n/N_n \rfloor)$$

$$q((i_1, i_2, \ldots, i_n)) = (i_1 \bmod N_1, i_2 \bmod N_2, \ldots, i_n \bmod N_n)$$

where $N_1, N_2, \ldots, N_n$ are sizes (extents) of each dimension subscript vector such that $N_1 N_2 \ldots N_n = T$. When an array is tiled, our compiler also *rewrites* the program statements that access this array accordingly.

2) Loop Tiling: While data tiling transforms the memory layout of each compressible array, loop tiling (also known as iteration space blocking) transforms the order in which the array elements are accessed within a loop nest. If used appropriately, loop tiling can significantly reduce the number of decompressions invoked during the execution of a loop nest. Fig. 5 shows such an example. Fig. 5(a) shows the original code of a loop nest, which accesses a 600 $\times$ 600 array X. We apply data tiling to array X such that the size of each tile is 100 $\times$ 100. Fig. 5(b) shows the code after data tiling. For illustration purposes, let us assume that the decompression buffer has a space to store up to three tiles and that we use a least recently used-based policy to select the victim tiles in the decompression buffer. We can compute that, during the execution of this loop nest, we need to invoke the decompresser 100 times for each tile. Hence, the decompresser is invoked $100 \times 36 = 3600$ times. By applying loop tiling to the loop nest shown in Fig. 5(b), we obtain the tiled loop nest shown in Fig. 5(c). In this tiled code, loop iterators i and j are the *intertile iterators*, and the loop nest formed by them is referred to as the *intertile loop nest.* Similarly, the iterators ii and jj are the *intratile iterators*, and the loop nest formed by them is referred to as the *intratile loop nest.* During the execution of this loop nest, the decompresser is invoked only 36 times, once for each i, j combination. In the rest of this paper, we assume that the loop nests in the application program have been appropriately tiled according to the layout of the arrays imposed by data tiling.

3) Compression-Based Space Management: Our compiler inserts a buffer management code at each loop nest that uses

the decompression buffer. For ease of discussion, we use the following abstract form to represent a tiled loop nest:

$$\text{for } \vec{I} = \vec{L} \text{ to } \vec{U} \Big\{ T_1\left(R_1(\vec{I}), W_1(\vec{I})\right); \; T_2\left(R_2(\vec{I}), W_2(\vec{I})\right); \; \ldots \; T_n\left(R_n(\vec{I}), W_n(\vec{I})\right) \Big\}$$

where $\vec{I}$ is the iteration vector and $\vec{L}$ and $\vec{U}$ are the lower and upper bound vectors for the loop nest. T_i $(i = 1 \ldots n)$ represents an intratile loop nest. Since we focus on the access pattern of each array at a tile level, we treat each intratile loop nest as an atomic operation. $R_i(\vec{I})$ is the set of tiles that might be read, and $W_i(\vec{I})$ is the set of tiles that might be written in the intratile loop nest T_i at the intertile iteration $\vec{I}$. $R_i(\vec{I})$ and $W_i(\vec{I})$ can be computed as follows:

$$R_i(\vec{I}) = \Big\{ X_k \left[\!\left[f_j^{(i)}(\vec{I}) \right]\!\right] \mid \text{“}\ldots = \ldots X_k \left[\!\left[f_j^{(i)}(\vec{I}) \right]\!\right][\ldots] \ldots\text{” appears in } T_i \Big\}$$

$$W_i(\vec{I}) = \Big\{ X_k \left[\!\left[f_j^{(i)}(\vec{I}) \right]\!\right] \mid \text{“}X_k \left[\!\left[f_j^{(i)}(\vec{I}) \right]\!\right][\ldots] = \ldots\text{” appears in } T_i \Big\}$$

where X_k is an array accessed by T_i and $f_j^{(i)}(\vec{I})$ is a mapping function that maps intertile iteration vector $\vec{I}$ into a tile of array X_i. Note that, we must be conservative in computing $R_i(\vec{I})$ and $W_i(\vec{I})$. For intratile loop nest T_i, let us assume

$$W_i(\vec{I}) = \Big\{ X_{k_1}\left[\!\left[f_1^{(i)}(\vec{I}) \right]\!\right], X_{k_2}\left[\!\left[f_2^{(i)}(\vec{I}) \right]\!\right], X_{k_j}\left[\!\left[f_j^{(i)}(\vec{I}) \right]\!\right] \Big\}$$

$$R_i(\vec{I}) - W_i(\vec{I}) = \Big\{ X_{k_{j+1}}\left[\!\left[f_{j+1}^{(i)}(\vec{I}) \right]\!\right], X_{k_{j+2}}\left[\!\left[f_{j+2}^{(i)}(\vec{I}) \right]\!\right], X_{k_m}\left[\!\left[f_m^{(i)}(\vec{I}) \right]\!\right] \Big\}.$$

In the transformed code, we use counter c to count the number of intratile loop nests that have been executed up to a specific point. B is the set of tiles that are currently in the decompression buffer. Before entering intratile loop nest T_i, we need to decompress all the tiles in the set $R_i(\vec{I}) \cup W_i(\vec{I}) - B$. That is, all the tiles that will be used by T_i must be available in the uncompressed format before we start executing T_i. When decompressing a tile t, we may need to evict a tile from the decompression buffer if there is no free block in the decompression buffer (indicated by $|B| = D$). Each tile t in the decompression buffer is associated with an integer $t.r$, indicating when this tile will be reused in the future. Specifically, $t_1.r < t_2.r$ indicates that tile t_1 will be reused earlier than t_2. When evicting a tile, we select the one that will be reused in the furthest future. Each tile t in the decompression buffer also has a flag $t.w$ indicating whether this block has been written since its last decompression. If this victim tile has been written, we need to recompress this tile. Before entering T_i, we also update the *next reuse time* $(t.r)$ for each tile (t) used by T_i. The next reuse time of tile t is computed using $t.r = c + d_i(t)$, where c is the number of intratile loop nests that have been executed

```
for i = 1 to 2
  for j = 1 to 3 {
        c = c + 1;
        load(X[[i, j]], X[[i, j + 1]]);
    L1: for ii = 1 to 10
          for jj = 1 to 10 {
            S1: ...X[[i, j]][...]...     // reuse distance d1 = 3
            S2: ...X[[i, j + 1]][...]...// reuse distance d2 = 1
          }
        c = c + 1;
        load(X[[i, j + 1]], X[[i, j - 1]]);
    L2: for ii = 1 to 10
          for jj = 1 to 10 {
            S3: ...X[[i, j + 1]][...]...// reuse distance d3 = 1
            S4: ...X[[i, j - 1]][...]...// reuse distance d4 = ∞
          }
  }
```

(a)

i	j	c	Nest	Tiles of array X in the buffer		
1	1	1	$\mathcal{L}_1$	$*[\![1,1]\!]:4$	$*[\![1,2]\!]:2$	
1	1	2	$\mathcal{L}_2$	$[\![1,1]\!]:4$	$[\![1,2]\!]:3$	$*[\![1,0]\!]:\infty$
1	2	3	$\mathcal{L}_1$	$[\![1,1]\!]:4$	$[\![1,2]\!]:6$	$*[\![1,3]\!]:4$
1	2	4	$\mathcal{L}_2$	$[\![1,1]\!]:\infty$	$[\![1,2]\!]:6$	$[\![1,3]\!]:5$
1	3	5	$\mathcal{L}_1$	$*[\![1,4]\!]:6$	$[\![1,2]\!]:6$	$[\![1,3]\!]:8$
1	3	6	$\mathcal{L}_2$	$[\![1,4]\!]:7$	$[\![1,2]\!]:\infty$	$[\![1,3]\!]:8$
2	1	7	$\mathcal{L}_1$	$[\![1,4]\!]:7$	$*[\![2,1]\!]:10$	$*[\![2,2]\!]:8$
2	1	8	$\mathcal{L}_2$	$*[\![2,0]\!]:\infty$	$[\![2,1]\!]:10$	$[\![2,2]\!]:9$
2	2	9	$\mathcal{L}_1$	$*[\![2,3]\!]:10$	$[\![2,1]\!]:10$	$[\![2,2]\!]:12$
2	2	10	$\mathcal{L}_2$	$[\![2,3]\!]:11$	$[\![2,1]\!]:\infty$	$[\![2,2]\!]:12$
2	3	11	$\mathcal{L}_1$	$[\![2,3]\!]:14$	$*[\![2,4]\!]:12$	$[\![2,2]\!]:12$
2	3	12	$\mathcal{L}_2$	$[\![2,3]\!]:14$	$[\![2,4]\!]:13$	$[\![2,2]\!]:\infty$

(b)

Fig. 6. Example that demonstrates how our approach operates.

and $d_i(t)$ is the *reuse distance* of tile t at intratile loop nest T_i. The reuse distance of a tile is the number of intratile loop nests executed between the current and the next accesses to that tile. We use a compiler-based approach to compute $d_i(t)$—the reuse distance of tile t at intratile loop nest T_i. The interested reader is referred to [2] for a detailed analysis.

4) Example: Fig. 6(a) shows an intertile loop nest containing two intratile loop nests: $\mathcal{L}_1$ and $\mathcal{L}_2$. Intratile loop nest $\mathcal{L}_1$ uses tiles $X[\![i,j]\!]$ and $X[\![i,j+1]\!]$, with reuse distances $d_1 = 3$ and $d_2 = 1$, respectively. Intratile loop nest $\mathcal{L}_2$ uses tiles $X[\![i,j+1]\!]$ and $X[\![i,j-1]\!]$, with reuse distances $d_3 = 1$ and $d_4 = \infty$, respectively. This code uses nine tiles throughout its execution. Fig. 6(b) shows the state of the decompression buffer during the execution. We observe that each tile is decompressed only once, which indicates that our compiler-directed buffer management approach is very effective in minimizing the number of decompressions.

B. Experiments

1) Platform: We implemented our compression-based approach using the Stanford University Intermediate Format (SUIF) [34] compiler infrastructure. SUIF defines a small kernel and an intermediate representation, around which several independent optimization passes can be added. We implemented our approach as a separate pass. We observed during our experiments that the largest increase in original compilation times was about 65%, over the original case without our optimization. To gather performance statistics, we used the SimpleScalar [35] simulation environment and modeled an embedded architecture.

To obtain energy numbers, we used energy models similar to Wattch [36]; the memory energy values (which include both dynamic and leakage components) are obtained using the CACTI tool [37]. We assume that a 1-MB SRAM memory is available, and the on-chip memory space is divided into banks, each being 64 KB. Furthermore, we assume that memory access latencies for on- and off-chip memories are 2 and 100 cycles, respectively. Note that our main goal is to reduce the memory-space consumption of a given application (both MMO and AMO). We are not really concerned in this paper with the question of how to utilize the memory space saved through our compiler-based approach (the saved memory space can be used for different purposes). However, to demonstrate the potential side benefits of our approach, we also present performance and energy statistics.

As our compression/decompression method, we used the Lempel–Ziv–Oberhumer (LZO) algorithm [38]. LZO is a data-compression library which is suitable for data decompression in real time. It is very fast in compression and extremely fast in decompression. The algorithm is both thread safe and lossless. In addition, it supports overlapping compression and in-place decompression. We want to mention, however, that our compiler approach can work with any compression/decompression algorithm and is not tied to a particular one in any way.

For each benchmark code in our experimental suite, we performed experiments with five different versions, which can be summarized as follows.

1) *BASE*: The base execution does not employ any data compression or decompression, but only uses iteration space tiling. The specific loop tiling strategy used is due to Coleman and McKinley [39]. The memory saving and performance overhead results presented in the rest of this section are all *normalized* with respect to this base version.
2) *LF*: This is similar to the *BASE* scheme in that it does not employ any data compression or decompression. The difference between *LF* and *BASE* is that the former uses a *lifetime analysis* at a data block level and reclaims the memory space occupied by dead data blocks, i.e., the block that will not be used during the rest of the execution. Consequently, as compared to the *BASE* version, one can expect this scheme to reduce both MMO and AMO. This version implements an optimal strategy in recycling the dead blocks, i.e., the memory space allocated to a given data block is recycled into free space as soon as the data block is dead (i.e., completed its last reuse).
3) *AGG*: This is a compression/decompression-based scheme that uses data compression very aggressively. Specifically, as soon as an access to a data block is completed, it is compressed. While one can expect this approach to reduce memory-space consumption significantly, it can also incur significant performance penalties.
4) *CD*: This is the compiler-directed scheme proposed in this paper. As discussed earlier in detail, it uses compression and decompression based on the data-reuse information extracted from the program code by the automatic compiler analysis.
5) *CD+LF*: This is a scheme that combines our compression-based approach with dead block recycling. In principle, this version should generate the best memory occupancy savings.

TABLE I
OUR BENCHMARKS AND THEIR CHARACTERISTICS

Benchmark	Functionality	MMO (KB)	AMO (KB)	Cycles (M)
facerec	face recognition	219.6	160.4	434.63
jacobi	Jacobi iteration	293.4	181.0	577.44
lu	LU decomposition	424.8	372.4	886.53
mpeg-2	MPEG-2 encoding	488.8	366.4	961.58
simi	pattern recognition	518.1	411.9	984.30
spec	spectral analysis	267.3	187.3	448.11
wave	wave equation solver	397.2	304.1	798.76
wibi	3D game program	793.6	527.7	1,488.19

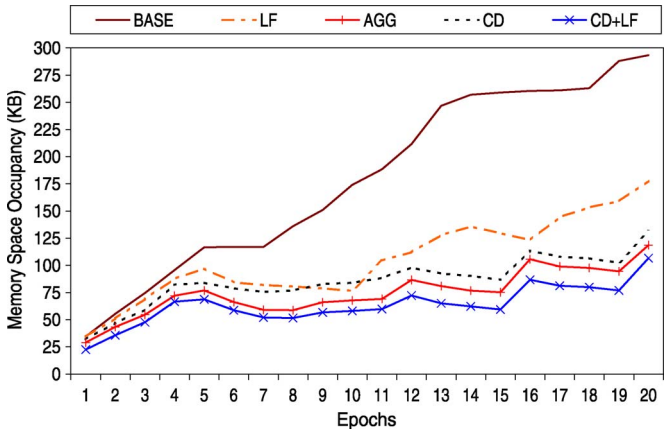

Fig. 7. Memory-space occupancy for benchmark Jacobi's.

The set of benchmark codes used in this part of our experiments are given in Table I. The second column of this table explains the functionality implemented by each benchmark. The next two columns show MMO and AMO in kilobytes for real input data. The last column gives the execution cycles under the base version, i.e., the one without any data compression/decompression or lifetime analysis. As aforementioned, our results are presented as normalized values with respect to those obtained with the *BASE* version. In our experiments, all arrays in each application are marked as compressible.

2) Results: Our first set of results are shown in Fig. 7 for a representative benchmark: Jacobi's. This graph gives the memory-space occupancy during execution for the five different versions explained earlier. The execution time of each benchmark is divided into 20 epochs (each epoch has the same number of execution cycles). The value corresponding to an epoch is the maximum value seen in that epoch. One can make several important observations from this graph. First, as expected, the memory occupancy trend of the *BASE* version continuously increases. In comparison, the *LF* version has much better memory occupancy behavior. Note that the sudden drops in its curve correspond to dead block recycling. The *AGG* scheme also shows savings over the *BASE* version, due to its aggressive compression. When we look at the behavior of our approach (*CD*), we see that, while it is not as good as the *AGG* scheme, its results are competitive with those obtained using the *LF* scheme. In other words, data compression can be as effective as dead block recycling. Finally, we see that the best space savings are achieved with the *CD+LF* version since it combines the advantages of both data compression and

TABLE II
MMO AND AMO VALUES (IN KILOBYTES) WITH DIFFERENT SCHEMES

Bench-mark	*LF*		*AGG*		*CD*		*CD+LF*	
	MMO	**AMO**	**MMO**	**AMO**	**MMO**	**AMO**	**MMO**	**AMO**
facerec	143.3	118.3	114.7	88.2	126.0	97.4	98.8	77.1
jacobi	177.7	105.4	118.7	74.8	131.3	86.0	106.7	63.4
lu	328.0	299.8	253.2	218.8	281.1	236.9	217.8	194.8
mpeg-2	367.4	281.4	307.6	229.9	340.5	254.1	281.5	205.4
simi	427.2	352.0	376.8	298.6	399.0	328.3	352.1	277.1
spec	151.3	114.5	126.3	83.2	143.6	98.1	109.4	71.6
wave	311.5	245.1	258.1	197.7	303.0	257.8	234.0	180.3
wibi	585.4	399.2	498.7	320.3	518.7	338.3	466.7	301.9

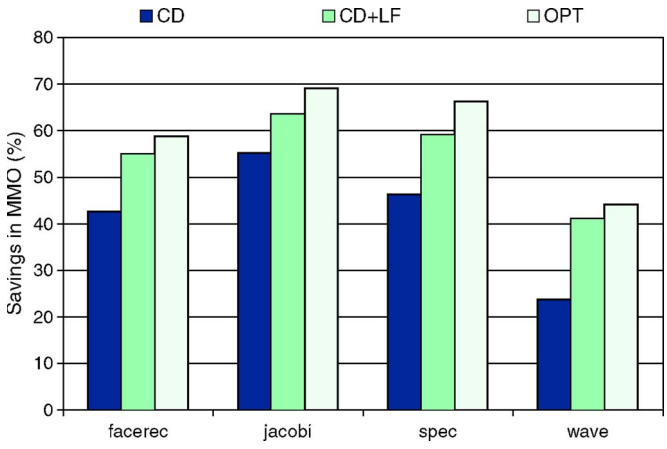

Fig. 8. Comparison with the optimal scheme.

dead block recycling. Table II summarizes the MMO and AMO values (in kilobytes) for all our benchmarks. We see that both *CD* and *CD+LF* schemes are able to reduce both MMO and AMO significantly.

It is also interesting to see how close our approach can come to optimal. In order to check this, we implemented an integer-linear-programming-based tool that can give us the optimal memory-space savings, given the input tile access pattern. Unfortunately, this tool is slow in finding an optimal result; therefore, we were able to run it only for four of our benchmarks. The results (the percentage savings in MMO) are shown in Fig. 8 (the results for the *CD* and *CD+LF* are reproduced for ease of comparison). We see from these results that, for these four benchmarks, the average MMO savings are 41.9%, 54.7%, and 59.6% for the *CD*, *CD+LF*, and optimal schemes, respectively. In other words, our approach, when combined with a lifetime-based memory saving strategy, can come close to optimal.

While these results clearly show that our approach is able to reduce memory-space occupancy significantly, the memory consumption is only one part of the whole picture. The other part is the impact on execution cycles and energy consumption. The performance and energy overheads incurred by our approach depends largely on the underlying execution model. For our experiments, we assumed that the same thread executes both application and compressions/decompressions. Fig. 9 shows the percentage increase in execution cycles for our approach (*CD*) with respect to the *BASE* scheme. As can be seen from these results, the average degradation in performance is 8.02%. The largest increases occur with the *mpeg-2* and *wave* benchmarks since they exhibit the lowest data reuse among our benchmarks. Since the compiler-based approach exploits

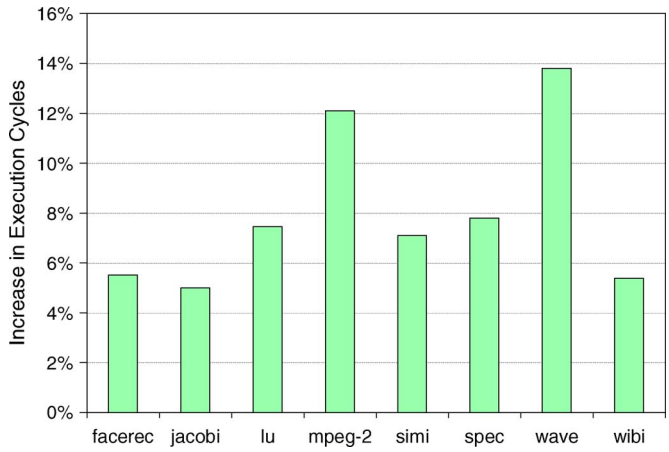

Fig. 9. Increase in execution cycles.

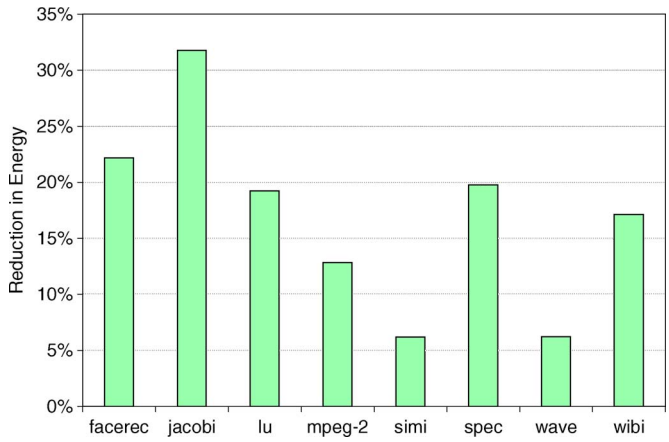

Fig. 10. Savings in energy consumption due to our approach.

data-reuse information in minimizing the performance overheads due to decompressions, it is less successful with these two codes. The performance degradations with the remaining benchmarks on the other hand are not very large. The energy results with the *CD* scheme are shown in Fig. 10. In performing the experiments from which we collected the energy numbers, we assumed that each memory bank can be placed into a low power (low leakage) mode when it is not used in the past 2000 cycles (using a mechanism similar to that discussed in [40], except that we turn on/off the memory at a bank granularity). While in the low leakage mode, a bank is assumed to consume 5% of its original leakage energy (per cycle). Since data compression reduces the total memory demand, the optimized programs operate with fewer banks on average, which, in turn, reduces the energy consumption. Consequently, as can be seen from Fig. 10, our approach saves energy over the BASE case in all benchmarks, and the average energy saving is 16.9%. Note that, we did not explicitly show the energy savings brought by the *CD+LF* scheme as they are very similar to the ones brought by the *CD* scheme. This is due to the fact that both *CD+LF* and *CD* exercise the same banks except some slight differences.

Fig. 11 shows, for each benchmark, the breakdown of overheads incurred by our approach into three categories: compression, decompression, and bookkeeping overheads (which include all table management activities). We see that, while the contribution of each type of overhead varies with the benchmark used, all types of overheads contribute to the final

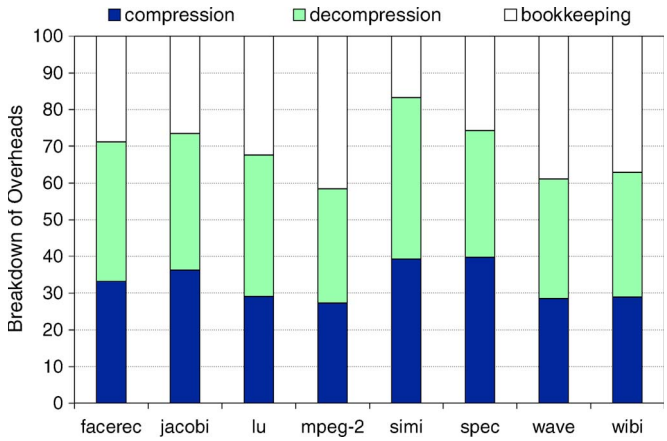

Fig. 11. Breakdown of the performance overheads incurred by our approach into three categories: compression, decompression, and bookkeeping overheads.

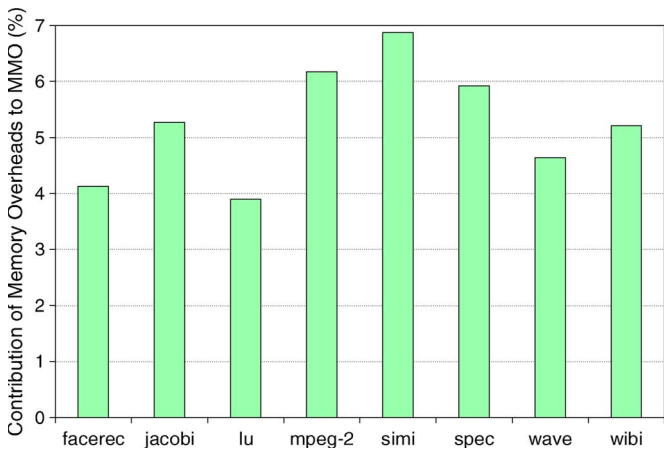

Fig. 12. Memory overheads brought by our approach as a fraction of MMO.

result. It needs to be noted, however, that these overheads are already captured in the MMO and AMO results presented earlier.

Fig. 12 shows the extra memory overheads brought by our approach, as a fraction of MMO (when the codes are optimized using our approach). Note that this value ranges from 3.9% to 6.9%, averaging on 5.3%. Note also that these overheads are already included in both the MMO and AMO results presented.

V. Data Compression in the MPSoC Case

Accessing off-chip memory presents at least three major problems in an MPSoC architecture. First, off-chip memory latencies are continuously increasing due to increases in processor clock frequencies. Consequently, large performance penalties are paid even if a small fraction of memory references go off-chip. Second, the bandwidth between the MPSoC and the off-chip memory may not be sufficient to handle simultaneous off-chip access requests coming from multiple processors. Third, frequent off-chip memory accesses can increase overall power consumption dramatically. In the remaining part of this paper, we propose and evaluate an on-chip memory management scheme for MPSoCs based on *data compression*. A critical issue in this context is to schedule compressions and decompressions intelligently so that they do not conflict with ongoing application execution. In particular, one needs to decide which processors should participate in the compression and decompression activities at any given point during the course of execution. While it is conceivable that all processors can participate in both application execution and compression/decompression activities, this may not necessarily be the best option. This is because, in many cases, some processors are idle (and therefore cannot take part in application execution anyway) and can be utilized entirely for compression/decompression and related tasks, thereby allowing other processors to focus solely on application execution. Therefore, an execution scheme that carefully *divides* the available computing resources between application execution and online compression/decompression can be very useful in practice.

A. Our Approach

1) Architecture and Code Parallelization: The MPSoC architecture that we consider in this paper is a shared multiprocessor-based system, where a certain number of processors (typically, on the order of 4–32) share the same memory address space. In particular, we assume that there exists an on-chip (software-managed [41]–[43]) memory space shared by all processors. We keep the subsequent discussion simple by using a shared bus as the interconnect, although one could use more sophisticated interconnects as well. The processors also share a large off-chip memory space. It should be noted that there is a trend toward designing domain-specific memory architectures [33], [44]–[47]. Such architectures are expected to be very successful in some application domains, where the software can analyze the application code, extract the regularity in data access patterns, and optimize the data transfers between on-chip and off-chip memories. Such software-managed memory systems can also be more power efficient than a conventional hardware-managed cache-based memory hierarchy [42], [43]. In this study, we assume that the software is in charge of managing the data transfers between the on-chip memory space and the off-chip memory space, although, as will be discussed later, our approach can also be used with a cache-based system.

We employ a *loop-nest-based code parallelization strategy* for executing array-based applications in this MPSoC architecture. In this strategy, each loop nest is parallelized for the coarsest grain of parallelism where the computational load processed by the processors between global synchronization points is maximized. We achieve this as follows. First, an optimizing compiler, again built on top of the SUIF infrastructure [34], analyzes the application code and identifies data reuses and data dependences. Then, the loops with data dependences and reuses are placed into inner positions (in the loop nest being optimized). This ensures that the loop nest exhibits a decent data locality, and the loops that remain the outer positions (in the nest) are mostly dependence free. After this step, for each loop nest, the outermost loop that does not carry any data dependence is parallelized. Since this type of parallelization tends to minimize the frequency of interprocessor synchronization and communication, we believe that it is very suitable for an MPSoC architecture. We use this parallelization strategy irrespective of the number of processors used for parallel execution and irrespective of the code version used. It should be emphasized, however, that, when some of the processors are reserved

for compression/decompression, they do not participate in the parallel execution of loop nests. While we use this specific loop parallelization strategy in this work, its selection is actually orthogonal to the focus of this work. In other words, our approach can work with different loop parallelization strategies.

2) Our Objectives: We can itemize the major objectives of our compression/decompression-based on-chip memory management scheme for MPSoCs as follows.

1) We would like to compress as much data as possible. This is because the more data are compressed, the more space we have in the on-chip memory, which will be available for new data blocks.
2) Whenever we access a data block, we prefer to find it in an uncompressed form. This is because, if it is in a compressed form during the access, we need to decompress it (and spend extra execution cycles for that) before the access could take place.
3) We do not want the decompressions to come into the critical path of execution. That is, we do not want to employ costly algorithms at runtime to determine which data blocks to compress or to use complex compression/decompression algorithms.

It is to be noted that some of these objectives conflict with each other. For example, if we aggressively compress each data block as soon as the current access to it terminates, this can lead to a significant increase in the number of cases where we access a data block and find it compressed. Therefore, an acceptable execution model based on data compression and decompression should exploit the tradeoffs between these conflicting objectives. Note also that, even if we find the data block in the on-chip memory in the compressed form, depending on the processor frequency and the decompression algorithm employed, this option can still be better than not finding it in the on-chip storage at all and bringing it from the off-chip memory. Moreover, our approach tries to take decompressions out of the critical path (by utilizing idle processors) as much as possible, and it thus only compresses the data blocks that will not be needed for some time. Furthermore, the off-chip memory accesses keep getting more and more expensive in terms of processor cycles and power consumption. Therefore, one might expect a compression-based MPSoC memory management scheme to be even more attractive in the future.

3) Compression/Decompression Policies and Implementation Details: We explore two different strategies, explained in the following, for dividing the available processors between compression/decompression (and related activities) and application execution.

1) *Static Strategy*:[1] In this strategy, a fixed number of processors are allocated for performing compression/decompression activity, and this allocation is not changed during the course of execution. The main advantage of this strategy is that it is easy to implement. Its main drawback is that it does not seem easy to determine the ideal number of processors to be employed for compression and decompression. This is because this number depends on several factors such as the application's data access pattern, the number of total processors in the MPSoC, and the relative costs of compression and decompression and off-chip memory access. In fact, as will be discussed later in detail, our experiments clearly indicate that each application demands a different number of processors (to be allocated for compression/decompression and related activities). Furthermore, it is conceivable that, even within an application, the ideal number of processors to employ in compression/decompression could vary across the different execution phases.
2) *Dynamic Strategy:* The main idea behind this strategy is to eliminate the optimal processor selection problem of the static approach aforementioned. By changing the number of processors allocated for compression and decompression dynamically, this strategy attempts to adapt the MPSoC resources to the dynamic application behavior. Its main drawback is the additional overhead it entails over the static one. Specifically, in order to decide how to change the number of processors (allocated for compression and decompression) at runtime, we need a metric that allows us to make this decision during execution. In this section, we make use of a metric, referred to as the *miscompression rate,* which gives the rate between the number of accesses made to the compressed data and the total number of accesses. We want to reduce the miscompression rate as much as possible since a high miscompression rate means that most of the data accesses find the data in the compressed form, and this can degrade overall performance by bringing decompressions into the critical path.

Irrespective of whether we are using the static or dynamic strategy, we need to keep track of the accesses to different data blocks to determine their access patterns so that an effective on-chip memory-space management can be developed. In our architecture, this is done by the processors reserved for compression/decompression (see [1] for more details). The processors reserved for compression and decompression maintain reuse information at the data block granularity. For a data block, we define the *interaccess time* as the gap (in terms of intervening block accesses) between two successive accesses to that block. Our approach predicts the next interaccess time to be the same as the previous one, and this allows us to rank the different blocks according to their next estimated accesses. Then, using this information, we can decide which blocks to compress, which blocks to leave as they are, and which blocks to send to the off-chip memory. Consider Fig. 13(a) which shows the different possible cases for a given data block. After the current use of the block is over, we estimate its next access. If it is soon enough (in relative to the other on-chip blocks)—denoted Next use (1)—we keep the block in the on-chip memory as it is (i.e., without any compression). On the other hand, if the next access is not that soon [as in the case marked Next use (2)], we compress it (but still keep it in the on-chip memory). Finally, if the next use of the block is

[1] *Static* and *Dynamic* are used to describe how we select the processors which will perform the compression and those which will do the actual application processing. In *Static*, the number of processors are preset to a certain value regardless of what happens at runtime. On the other hand, *Dynamic* decides this at runtime considering the workload. However, both techniques use a compiler to decide which blocks to compress/decompress and when this will be performed.

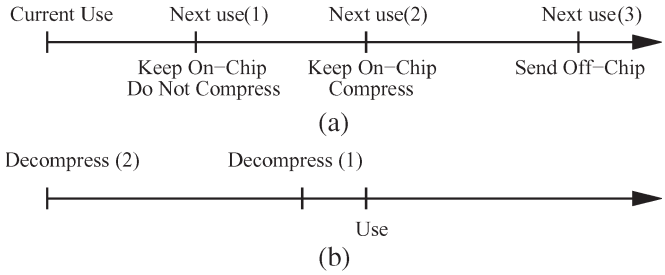

Fig. 13. (a) Different scenarios for data when its current use is over. (b) Comparison of on-demand decompression and predecompression. Arrows indicate the execution timeline of the program.

predicted to be really far [see Next use (3) in Fig. 13(a)], it is beneficial to send it to the off-chip memory (the block can be compressed before being forwarded to the off-chip memory to reduce transfer time/energy).

Our implementation of this approach is as follows. When the current use of a data block is over, we predict its next use and rank it along with the other on-chip blocks. Then, using two threshold values (Th_1 and Th_2) and taking into account the size (capacity) of the on-chip memory, we decide what to do with the block. More specifically, if the next use of the block is (predicted to be) T_n cycles away, we proceed as follows: keep the block in the on-chip memory uncompressed, if $T_n \leq Th_1$, or else, keep the block in the on-chip memory compressed, if $Th_1 < T_n \leq Th_2$, or else, send the block to the off-chip memory if $T_n > Th_2$.

It is to be noted that this strategy clearly tries to keep data with high reuse in on-chip memory as much as possible, even doing so requires compressing the data. As an example, suppose that we have just finished the current access to data block DB_i, and the on-chip memory currently holds s data blocks (some of which may be in a compressed form). We first calculate the time for the next use of DB_i (call this value T_n). As explained earlier, if $T_n \leq Th_1$, we want to keep DB_i in the on-chip memory in an uncompressed form. However, if there is no space for it in the on-chip memory, we select the data block DB_j with the largest next-use distance, compress it, and forward it to the off-chip memory. We repeat the same procedure if $Th_1 < T_n \leq Th_2$ except that we leave DB_i in the on-chip memory in a compressed form. Finally, if $T_n > Th_2, DB_i$ is compressed and forwarded to the off-chip memory. This algorithm is executed after the completion of processing any data block. Furthermore, a similar activity takes place when we want to bring a new block from the off-chip memory to the on-chip memory or when we create a new data block.

While this approach takes care of the compression part, we also need to decide when to decompress a data block. Basically, there are at least two ways of handling decompressions. First, if a processor needs to access a data block and finds it in the compressed form, the block should be decompressed first before the access can take place. This is termed as *on-demand decompression* in this paper, and an example is shown in Fig. 13(b) as Decompress (1). In this case, the data block in question is decompressed just before the access is made. A good memory-space-management strategy should try to minimize the number of on-demand decompressions since they incur performance penalties.

The second strategy is referred to as *predecompression* in this paper and is based on the idea of decompressing the data block before it is really needed. This is akin to software-based data prefetching [48] employed by some optimizing compilers. In our implementation, predecompression is performed by the processors allocated for compression/decompression since they have the next access information for the data blocks. An example predecompression is marked as Decompress (2) in Fig. 13(b). We want to maximize the number of predecompressions for the compressed blocks so that we can hide as much decompression time as possible. Notice that, during predecompression, the processors allocated for application execution are not affected, i.e., they continue with application execution. Only the processors reserved for compression and decompression participate in the predecompression activity.

The compression/decompression implementation explained previously is valid for both the static and the dynamic schemes. However, in the dynamic strategy case, an additional effort is needed for collecting statistics on the rate between the number of on-demand compressions and the total number of data block accesses (as mentioned earlier, this is called the miscompression rate). Our current implementation maintains a *global counter* that is updated, within a protected memory region in the on-chip storage, by all the processors reserved for compression/decompression. An important issue that is to be addressed is when do we need to increase/decrease the number of processors allocated for compression/decompression and related activities. For this, we adopt two thresholds Mr_1 and Mr_2. If the current miscompression rate is between Mr_1 and Mr_2, we do not change the existing processor allocation. If it is smaller than Mr_1, we decrease the number of processors allocated for compression/decompression. In contrast, if it is larger than Mr_2, we increase the number of processors allocated for compression/decompression. The rationale behind this approach is that, if the miscompression rate becomes very high, this means that we are not able to decompress data blocks early enough; therefore, we put more processors for decompression. On the other hand, if the miscompression rate becomes very low, we can reduce the resources that we employ for decompression. *To be fair in our evaluation, all the performance data presented in* Section V-B *include these overheads as well.*

It is important to measure miscompression rate in a low-cost yet accurate manner. One possible implementation is to calculate/check the miscompression rate after every T cycles. The important issue then is to select the most appropriate value for T. A small T value may not be able to capture miscompression rate accurately and incurs significant overhead at runtime. In contrast, a large T value does not cause much runtime overhead. However, it may force us to miss some optimization opportunities by delaying potential useful compressions and/or decompressions. In our experiments, we implemented this approach and also measured the sensitivity of our results to the value of the T parameter. Finally, it should also be mentioned that keeping the access history of the on-chip data blocks requires some extra space. Depending on the value of T, we allocate a certain number of bits per data block and update them each time a data block is accessed. In our

TABLE III
BASE SIMULATION PARAMETERS USED IN OUR EXPERIMENTS

Parameter	Default Value
Hardware Parameters	
Number of Processors	8
Clock Frequency	400MHz
On-Chip Memory Size	1MB; divided into 64KB banks
On-Chip Memory Latency	2 cycles
Off-Chip Memory Size	16MB
Off-Chip Memory Latency	100 cycles
Software Parameters	
Compression/Decompression Algorithm	LZO
Compression/Decompression Rate	20MB/sec
Block Size	2KB
Th_1, Th_2	5000 cycles, 50000 cycles
Mr_1, Mr_2	0.2, 0.6
Sampling Period (T)	20000 cycles
Starting $\|C\|$ Value for the Dynamic Scheme	2

implementation, these bits are stored in a certain portion of the on-chip memory, reserved just for this purpose. While this introduces both space and performance overhead, we found that these overheads are not really excessive. In particular, the space overhead was always less than 4%. Furthermore, all the performance numbers given in the next section include the cycle overheads incurred for updating these bits. Apart from this bookkeeping required to keep track of reuse blocks and capture miscompressions, the space management for blocks is similar to the case with a single core (explained earlier).

TABLE IV
BENCHMARK CODES USED AND IMPORTANT STATISTICS. IN OBTAINING THESE STATISTICS, THE REFERENCE INPUT SETS ARE USED

Benchmark	Cycles-1	Cycles-2
swim	91,187,018	118,852,504
apsi	96,822,310	127,028,682
fma3d	126,404,189	161,793,882
mgrid	87,091,915	96,611,130
applu	108,839,336	139,955,208
ocean	337,421,084	389,711,573
raytrace	387,509,892	446,109,638
BLU	273,128,538	309,784,541

B. Experimental Evaluation

1) Setup: We used Simics [49] to simulate an on-chip multiprocessor environment. Simics is a simulation platform for hardware development and design space exploration. It supports modifications to the instruction set architecture, architectural performance models, and devices. We use a variant of the LZO compression/decompression algorithm [38] to handle compressions and decompressions; the decompression rate of this algorithm is about 20 MB/s. It is to be emphasized that while, in this particular implementation, we chose LZO as our algorithm, our approach can work with any algorithm. In our approach, LZO is executed by the processors reserved for compression/decompression. To collect energy numbers, we enhanced the base Simics platform with Wattch-like [36] energy models. Table III lists the base simulation parameters used in our experiments.

Since most of the applications used in Section IV-B are not amenable to parallelism, we could not use them in our MPSoC experiments. Instead, we tested the effectiveness of our approach using five randomly selected applications (swim, apsi, fma3d, mgrid, and applu) from the SpecFP2000 suite [50] and three applications (ocean, raytrace, and BLU) from the Splash2 suite [51]. For each application, we fastforwarded the first 500 million instructions and simulated the next 250 million instructions. Two important statistics about these applications are given in Table IV. Since the original applications work with floating-point data (which is not amenable to compression), we converted their data to integers. The second column in Table IV (labeled Cycles-1) gives the execution time, in terms of cycles, of the original applications. The values in this column were obtained by using our base configuration (Table III) and using eight processors to execute each application without any data compression/decompression. In more details, the results in the second column of this table are obtained by using a parallel version of the software-based on-chip memory management scheme proposed in [43]. This scheme is a highly optimized dynamic approach that keeps the most reused data blocks in the on-chip memory as much as possible. In our opinion, it represents the state-of-the-art in software-managed on-chip memory optimization if one does not employ data compression/decompression. The performance (execution cycles) results reported in the next section are given as fractions of the values in this second column, i.e., they are *normalized* with respect to the second column of Table IV. The third column (named Cycles-2), on the other hand, gives the execution cycles for a compression-based strategy where each processor both participates in the application execution and performs on-demand decompression. In addition, when the current use of a data block ends, it is always compressed and kept on-chip. The on-chip memory space is managed in a fashion which is very similar to that of a fully associative cache. When we compare the results in the last two columns of this table, we see that this naive compression-based strategy is not any better than the case where we do not make use of any compression/decompression at all (the second column of the table). That is, in order to take advantage of data compression, one needs to employ smarter strategies. Our approach goes beyond this simplistic compression-based scheme and involves dividing the processor resources between those that do computation and those that perform compression/decompression-related tasks.

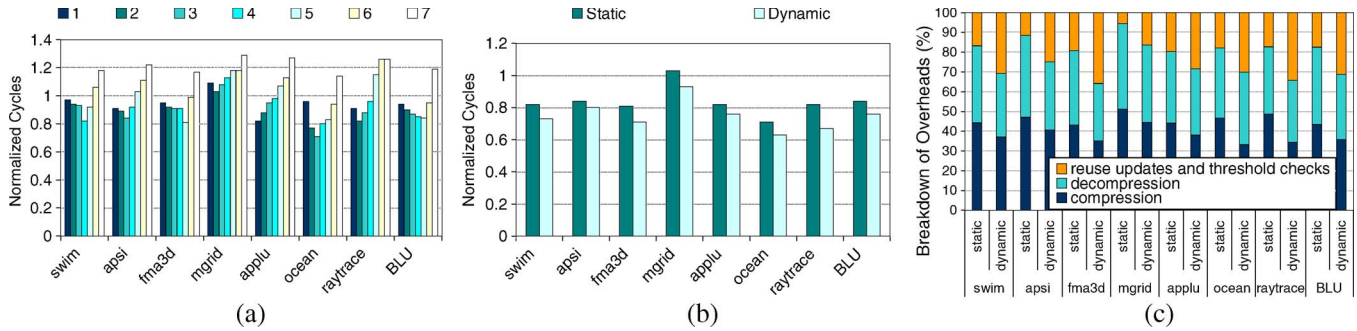

Fig. 14. (a) Normalized execution cycles (with respect to our base configuration without any data compression/decompression) for the static strategy using different $|C|$ values. (b) Comparison of the best static strategy (for each benchmark) and the dynamic strategy. Results are normalized with respect to our base configuration without any data compression/decompression. (c) Breakdown of the overheads into three different components for the static and dynamic schemes.

2) Results With the Base Parameters: Fig. 14(a) shows the behavior (normalized execution cycles with respect to our base configuration without any data compression/decompression) of the static approach with different $|C|$ values ($|C| = n$ means n out of eight processors are used for compression/decompression). As can be seen from the x-axis of this graph, we changed $|C|$ from one to seven. One can observe from this graph that, in general, the different applications prefer different $|C|$ values for the best performance characteristics. For example, while *apsi* demands three processors dedicated for compression/decompression for the best results, the corresponding number for *applu* is one. This is because each application has typically a different degree of parallelism in its different execution phases. That is, not all the processors participate in the application execution, e.g., as a result of data dependences or due to load imbalance concerns, and such otherwise idle processors can be employed for compression and decompression. We further observe from this graph that increasing $|C|$ beyond a certain value causes performance deterioration in all applications. This is due to the fact that employing more processors for compression and decompression than necessary prevents the application from exploiting the inherent parallelism in its loop nests, and that, in turn, hurts the overall performance. In particular, when we allocate six processors or more for compression and decompression, the performance of all eight applications in our suite becomes worse than the original execution cycles.

It is important to explain at this point why, in some cases, as can be observed from Fig. 14(a), smaller $|C|$ values generate better results, as compared to using larger $|C|$ values. It needs to be emphasized that, in all the experiments of Fig. 14(a), we used all eight processors in the MPSoC architecture. However, the $|C|$ value used (which is varied between one and seven) indicates how many of these processors are used exclusively for compression/decompression-related activities. In some cases, asking all (or most of) processors to do compression/decompression (in addition to executing their part of the application) can put these compression/decompression activities in the critical path, and this, in turn, increases the overall execution latency. Instead, reserving a certain set of processors for compression/decompression (i.e., not asking them to participate at the application execution) can take compression/decompression out of the critical path and improve performance by making sure that the data required by processors that execute the application code become available (i.e., decompressed) in time. It is also important to note that compressing data practically increases the effective on-chip memory capacity and allows the application to keep more tiles in on-chip memory, and this improves performance if we are able to take decompressions out of the critical path of execution.

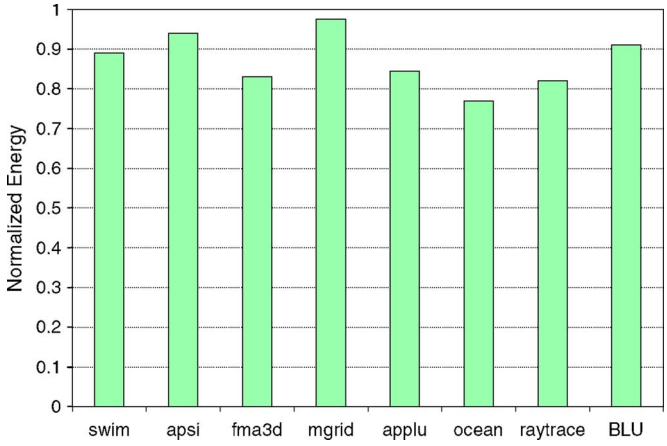

Fig. 15. Processor usage for the dynamic strategy over the execution period.

The graph in Fig. 14(b) shows a comparison of the static and dynamic strategies. The results in this graph are normalized execution cycles with respect to our base configuration without any data compression/decompression. The first bar for each benchmark gives the *best* static version, i.e., the one that is obtained using the ideal $|C|$ value for that benchmark. The second bar represents the normalized execution cycles for the dynamic scheme. One can see from these results that the dynamic strategy outperforms the static one for all the eight applications tested; the average performance improvement across all benchmarks is 16.3% and 25.1% for the static and dynamic strategies, respectively. That is, the dynamic approach brings additional benefits over the static one. To better explain why the dynamic approach generates better results than the static one, we show in Fig. 15 the execution behavior of the dynamic approach. More specifically, this graph divides the entire execution time of each application into 20 epochs and, for each epoch, shows the most frequently used $|C|$ value in that epoch. One can clearly see from the trends in this graph that the dynamic approach changes the number of processors dedicated to compression/decompression over the time, and in this way, it successfully adapts the available computing resources to the dynamic execution behavior of the application being executed.

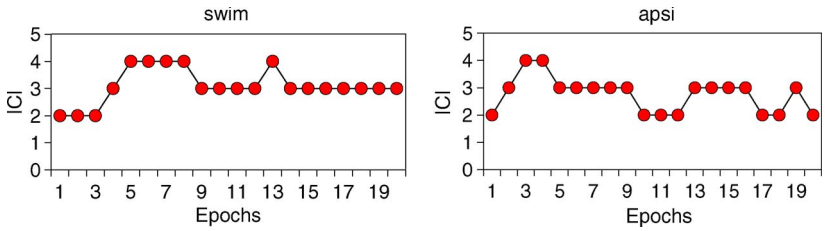

Fig. 16. Normalized energy consumption values.

Fig. 16 shows the normalized energy consumption results under our approach. As in the single-core case, we assumed that the memory space is divided into equal-sized banks (each is 64 KB) and that each bank can be transitioned to a low leakage mode if it is idle for a certain period of time. We see that our approach reduces energy consumption, i.e., it is beneficial from the energy perspective as well. However, the energy savings are lower than the performance savings, as the energy overheads incurred due to compression/decompression cannot be hidden (unlike the performance overhead which can be hidden most of the time during parallel execution). Still, we also see that the average energy saving is 12.7% when all eight benchmarks are considered.

We, next, present how the overheads incurred by our approach [effects of which are already shown in Fig. 14(a) and (b)] are decomposed into different components. In the bar chart shown in Fig. 14(c), we give the individual contributions of the three main sources of overheads: compression, decompression, and reuse updates, threshold checks, and other bookkeeping activities. We see from these results that, in the static approach case, compression and decompression activities dominate the overheads, most of which are actually hidden during parallel execution. In the dynamic approach case, on the other hand, the overheads are more balanced, since the process of determining the $|C|$ value to be used currently incurs additional overheads. Again, as in the static case, an overwhelming percentage of these overheads are hidden during parallel execution.

VI. Conclusion and Future Work

This paper has presented a compiler-directed approach that inserts compression and decompression calls in the application code to reduce maximum and average memory-space consumption. In this approach, the compiler analyzes a given application code and extracts data-reuse information at the data block level. It then uses this information in deciding the set of data blocks to be compressed/decompressed as well as the points at which these actions need to be invoked. This paper also shows how data compression can be used in MPSoC-based architectures to reduce the number of off-chip accesses. Specifically, it explores two different approaches: static and dynamic. To test the effectiveness of our approach in reducing memory-space occupancy, we applied it to several array-based benchmarks. Our experimental results reveal that the proposed approach is very effective in reducing the memory-space consumption. The results also show that the proposed approach outperforms other schemes tested. We believe that these results are encouraging and motivate further research on compiler-directed data compression. As future work, we also would like to compare our compiler-directed approach with techniques using a hardware compressor.

References

[1] O. Ozturk, M. Kandemir, and M. J. Irwin, "Using data compression in an MPSoC architecture for improving performance," in *Proc. 15th ACM GLSVLSI*, 2005, pp. 353–356.

[2] O. Ozturk, G. Chen, and M. Kandemir, "Compiler-guided data compression for reducing memory consumption of embedded applications," in *Proc. Conf. Asia South Pacific Des. Autom.*, Jan. 2006, pp. 814–819.

[3] K. D. Cooper and T. J. Harvey, "Compiler-controlled memory," in *Proc. 8th Int. Conf. ASPLOS*, 1998, pp. 2–11.

[4] A. Wolfe and A. Chanin, "Executing compressed programs on an embedded RISC architecture," in *Proc. 25th Annu. Int. Symp. Microarchitecture, MICRO*, 1992, pp. 81–91.

[5] M. Ros and P. Sutton, "Code compression based on operand-factorization for VLIW processors," in *Proc. Conf. Data Compression*, 2004, p. 559.

[6] S. Debray and W. Evans, "Profile-guided code compression," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2002, pp. 95–105.

[7] B. Tunstall, "Synthesis of noiseless compression codes," Ph.D. dissertation, Georgia Inst. Technol., Atlanta, GA, 1967.

[8] Y. Xie, W. Wolf, and H. Lekatsas, "Code compression for VLIW processors using variable-to-fixed coding," in *Proc. 15th Int. Symp. Syst. Synthesis*, 2002, pp. 138–143.

[9] C. H. Lin, Y. Xie, and W. Wolf, "LZW-based code compression for VLIW embedded systems," in *Proc. Conf. Des., Autom. Test Eur.*, 2004, pp. 76–81.

[10] B. Abali, M. Banikazemi, X. Shen, H. Franke, D. E. Poff, and T. B. Smith, "Hardware compressed main memory: Operating system support and performance evaluation," *IEEE Trans. Comput.*, vol. 50, no. 11, pp. 1219–1233, Nov. 2001.

[11] L. Benini, D. Bruni, A. Macii, and E. Macii, "Hardware-assisted data compression for energy minimization in systems with embedded processors," in *Proc. Conf. Des., Autom. Test Eur.*, 2002, pp. 449–453.

[12] C. D. Benveniste, P. A. Franaszek, and J. T. Robinson, "Cache-memory interfaces in compressed memory systems," *IEEE Trans. Comput.*, vol. 50, no. 11, pp. 1106–1116, Nov. 2001.

[13] A. Macii, E. Macii, F. Crudo, and R. Zafalon, "A new algorithm for energy-driven data compression in VLIW embedded processors," in *DATE Conf. Expo.*, 2003, pp. 10 024–10 029.

[14] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *Proc. 33rd Annu. ACM/IEEE Int. Symp. Microarchitecture*, 2000, pp. 258–265.

[15] E. Ahn, S.-M. Yoo, and S.-M. S. Kang, "Effective algorithms for cache-level compression," in *Proc. 11th Great Lakes Symp. VLSI*, 2001, pp. 89–92.

[16] Y. Zhang and R. Gupta, "Enabling partial cache line prefetching through data compression," in *32nd ICPP*, 2003, pp. 277–285.

[17] J. S. Lee, W. K. Hong, and S. D. Kim, "Design and evaluation of a selective compressed memory system," in *Proc. IEEE ICCD*, 1999, pp. 184–191.

[18] O. Ozturk and M. Kandemir, "ILP-based energy minimization techniques for banked memories," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, no. 3, pp. 1–40, Jul. 2008.

[19] O. Ozturk, M. Kandemir, and G. Chen, "Access pattern-based code compression for memory-constrained systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, no. 4, pp. 1–30, Sep. 2008.

[20] D. J. Craft, "A fast hardware data compression algorithm and some algorithmic extensions," *IBM J. Res. Develop.*, vol. 42, no. 6, pp. 733–745, Nov. 1998.

[21] P. A. Franaszek, L. A. Lastras-Montano, S. Peng, and J. T. Robinson, "Data compression with restricted parsings," in *Proc. DCC*, 2006, pp. 203–212.

[22] J. L. Nunez, C. Feregrino, S. Bateman, and S. Jones, "The x-matchlite FPGA-based data compressor," in *Proc. ACM/SIGDA 7th Int. Symp. FPGA*, 1999, p. 255.
[23] *MAJC-5200*. [Online]. Available: http://www.sun.com/microelectronics/MAJC/5200wp.html
[24] *MP98: A Mobile Processor*. [Online]. Available: http://www.am.necel.com/news/newsdetail.html?page=000207a
[25] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," *SIGPLAN Not.*, vol. 31, no. 9, pp. 2–11, 1996.
[26] F. Gharsalli, S. Meftali, F. Rousseau, and A. A. Jerraya, "Automatic generation of embedded memory wrapper for multiprocessor SoC," in *Proc. 39th DAC*, 2002, pp. 596–601.
[27] S. Meftali, F. Gharsalli, F. Rousseau, and A. A. Jerraya, "An optimal memory allocation for application-specific multiprocessor system-on-chip," in *Proc. 14th ISSS*, 2001, pp. 19–24.
[28] V. Krishnan and J. Torrellas, "A chip-multiprocessor architecture with speculative multithreading," *IEEE Trans. Comput.*, vol. 48, no. 9, pp. 866–880, Sep. 1999.
[29] M. Collin, R. Haukilahti, M. Nikitovic, and J. Adomat, "Socrates—A multiprocessor SoC in 40 days," in *Conf. Des., Autom. Test Eur.*, 2001, pp. 410–441.
[30] B. A. Nayfeh, L. Hammond, and K. Olukotun, "Evaluation of design alternatives for a multiprocessor microprocessor," in *Proc. 23rd Annu. ISCA*, 1996, pp. 67–77.
[31] A. R. Alameldeen and D. Wood, "Adaptive cache compression for high-performance processors," in *ISCA*, 2004, pp. 212–223.
[32] A. R. Alameldeen and D. Wood, "Interactions between compression and prefetching in chip multiprocessors," in *Int. Symp. HPCA*, 2007, pp. 228–239.
[33] F. Catthoor, E. de Greef, and S. Suytack, *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Norwell, MA: Kluwer, 1998.
[34] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: An infrastructure for research on parallelizing and optimizing compilers," *SIGPLAN Not.*, vol. 29, no. 12, pp. 31–37, 1994.
[35] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, Feb. 2002.
[36] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proc. 27th Annu. Int. Symp. Comput. Archit.*, 2000, pp. 83–94.
[37] G. Reinman and N. P. Jouppi, "Cacti 2.0: An integrated cache timing and power model," Compaq, Palo Alto, CA, Tech. Rep., Feb. 2000.
[38] *LZO Algorithm*. [Online]. Available: http://gnuwin32.sourceforge.net/packages/lzo.htm
[39] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," in *Proc. ACM SIGPLAN Conf. PLDI*, 1995, pp. 279–290.
[40] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," in *Proc. 28th Annu. ISCA*, 2001, pp. 240–251.
[41] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Proc. Conf. Des., Autom. Test Eur.*, 2002, pp. 409–415.
[42] L. Benini, A. Macii, E. Macii, and M. Poncino, "Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation," *IEEE Des. Test Comput.*, vol. 17, no. 2, pp. 74–85, Apr.–Jun. 2000.
[43] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "Dynamic management of scratch-pad memory space," in *Proc. 38th Conf. Des. Autom.*, 2001, pp. 690–695.
[44] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood, "Lx: A technology platform for customizable VLIW embedded processing," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 203–213, May 2000.
[45] *M-core—MMC2001 Reference Manual*, Motorola Corporation, Denver, CO, 1998. [Online]. Available: http://www.motorola.com/SPS/MCORE/info_documentation.htm
[46] *CPU12 Reference Manual*, Motorola Corporation, Denver, CO, 2000. [Online]. Available: http://www.freescale.com/files/microcontrollers/doc/ref_manual/CPU12RM.pdf
[47] *TMS370CX7X 8-bit Microcontroller*, Texas Instruments, Dallas, TX, Feb. 1997. Revised. [Online]. Available: http://www-s.ti.com/sc/psheets/spns034c/spns034c.pdf
[48] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proc. 5th Int. Conf. ASPLOS*, 1992, pp. 62–73.
[49] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.
[50] J. L. Henning, "SPEC CPU2000: Measuring CPU performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28–35, Jul. 2000.
[51] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. 22nd Annu. ISCA*, 1995, pp. 24–36.
[52] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee, "Improving locality using loop and data transformations in an integrated framework," in *Proc. 31st Annu. ACM/IEEE Int. Symp. Microarchitecture, MICRO*, 1998, pp. 285–297.

Ozcan Ozturk (S'00–M'04) received the B.Sc. degree from Bogazici University, Istanbul, Turkey, the M.S. degree from the University of Florida, Gainesville, and the Ph.D. degree from The Pennsylvania State University, University Park, all in computer engineering.

He is an Assistant Professor with the Department of Computer Engineering, Bilkent University, Ankara, Turkey. Prior to joining Bilkent, he was a Software Optimization Engineer with the Cellular and Handheld Group, Intel (Marvell). His research interests are in the areas of chip multiprocessing, computer architecture, many-core architectures, and parallel processing.

Dr. Ozturk is a recipient of the Marie Curie Fellowship from the European Commission.

Mahmut Kandemir (S'98–A'99–M'03) is an Associate Professor with the Computer Science and Engineering Department, The Pennsylvania State University, University Park, where he is a member of the Microsystems Design Laboratory. His research interests are in optimizing compilers, runtime systems, embedded systems, I/O and high-performance storage, and power-aware computing. He is the author of more than 300 papers in these areas.

Mr. Kandemir is a recipient of the National Science Foundation (NSF) Career Award and the Penn State Engineering Society Outstanding Research Award. He is a member of the Association for Computing Machinery. His research is funded by the NSF, the Defense Advanced Research Projects Agency, and Semiconductor Research Corporation.

Mary Jane Irwin (F'95) received the M.S. and Ph.D. degrees in computer science from the University of Illinois, Urbana–Champaign, in 1975 and 1977, respectively.

She is an Evan Pugh Professor and the A. Robert Noll Chair of Engineering with the Computer Science and Engineering Department, The Pennsylvania State University, University Park. She was the Editor-in-Chief of the Association for Computing Machinery (ACM)'s *Transactions on Design Automation of Electronic Systems* from 1998 to 2004 and the Coeditor-in-Chief of ACM's *Journal of Emerging Technologies in Computing Systems* from 2005 to 2006. Her research and teaching interests include computer architecture (power constrained and application specific) and computer arithmetic, reliable systems design, and very large scale integration systems design and design automation.

Dr. Irwin was the recipient of an Honorary Doctorate from Chalmers University, Göteborg, Sweden, in 1997, was named a Fellow of ACM in 1996, and was elected to the National Academy of Engineering in 2003. She is currently serving as a member of the National Research Council's Board of Army Science and Technology and the Computing Research Association (CRA) Committee on the Status of Women in Computing Research Steering Committee. In the past, she has served as an elected member of the IEEE Computer Society's Board of Governors and of ACM's Council, as Vice President of ACM, and on CRA's Board of Directors.