Data Compression for Improving SPM Behavior *

O. Ozturk, M. Kandemir
Pennsylvania State University
University Park, PA 16802

{ozturk, kandemir}@cse.psu.edu

I. Demirkiran
Syracuse University
Syracuse, NY 13244

idemirki@eecs.syr.edu

G. Chen, M. J. Irwin
Pennsylvania State University
University Park, PA 16802

{gchen, mji}@cse.psu.edu

ABSTRACT

Scratch-pad memories (SPMs) enable fast access to time-critical data. While prior research studied both static and dynamic SPM management strategies, not being able to keep all hot data (i.e., data with high reuse) in the SPM remains the biggest problem. This paper proposes data compression to increase the number of data blocks that can be kept in the SPM. Our experiments with several embedded applications show that our compression-based SPM management heuristic is very effective and outperforms prior static and dynamic SPM management approaches. We also present an ILP formulation of the problem, and show that the proposed heuristic generates competitive results with those obtained through ILP, while spending much less time in compilation.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*

General Terms

Algorithms, Performance

Keywords

scratch-pad memory, compilers, data compression

1. INTRODUCTION

While cache memories can speed up embedded programs by exploiting data and instruction locality, there are several problems associated with them (in particular, with data caches). First, the success of a cache-based system depends strongly on the compatibility between the data access pattern and the hardware-guided cache line replacement policy. Second, since data transfers in and out of the cache are managed by hardware, there is no guarantee that a required data item will be in the cache at the time of access. This may be an important problem in real-time embedded systems. Third, in many situations, the cache management policy is too general for a given application. Because of these problems, as compared to general-purpose systems, data caches are not as prevalent in embedded systems where execution behavior/time predictability is often paramount.

In comparison, software-controlled *scratch-pad memory* (SPM) can guarantee fast access to time-critical data and instructions. A large data structure can be accessed through the SPM by dynamically loading its blocks on a need-basis. Prior research efforts on scratch-pad memories have mainly focused on different management strategies (e.g., static versus dynamic, or instruction SPM versus data SPM) and on hybrid architectures that make use of both conventional caches and SPMs. While the studies in [16, 11, 5, 19, 10, 7, 9, 12, 17, 20] have focused on data accesses, [13, 4, 17] have proposed strategies to optimize for instruction accesses. Today, many commercial architectures employ SPMs [15, 8, 18]. However, since 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.

In this paper, we explore the possibility of using *data compression* to increase the effective SPM space. The idea is to let the compiler derive access patterns of different data blocks, and come up with a reasonable SPM management strategy based on data compression/decompression. Specifically, the compiler builds a *schedule* that indicates (at each time step) which data blocks need to be brought from the off-chip memory to the SPM, which blocks should be sent from the SPM to the off-chip memory, and which blocks need to be compressed/decompressed within the SPM. Focusing on a set of array-intensive embedded applications, this paper reports that such a SPM management strategy can generate much better results than the current state-of-the-art, that does not employ compression/decompression. We also present an ILP (integer linear programming) based formulation that determines an optimal set of compressions/decompressions as well as the optimal points in execution to perform them. The experimental results with six embedded applications show that the compiler-based heuristic is much faster than the ILP-based solution, and generates competitive results. Based on our implementation and experimental results, we believe that the compiler is in a very good position to decide the program points at which data blocks need to be compressed/decompressed.

The rest of this paper is structured as follows. The next section discusses SPM-based execution models for both the default case (without compression) and the compression/decompression based case. Section 3 presents the details of our approach. Section 4 gives our compiler algorithm. Section 5 introduces our experimental platform, describes benchmarks, and reports experimental data that illustrate the effectiveness of our new SPM management strategy. Section 6 discusses how our approach compares to an ILP-based strategy that determines the optimal compressions and decompressions. Section 7 concludes the paper with a summary of our contributions.

2. EXECUTION MODEL

We assume a single CPU-based embedded architecture with a SPM. While the architecture can also have data and instruction caches, in this work we exclusively focus on SPM management and on data accesses. The execution model in a SPM-based environment depends strongly on the order in which *data blocks* (also
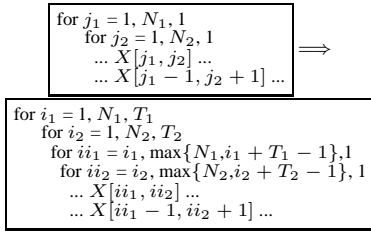

```
for j_1 = 1, N_1, 1
    for j_2 = 1, N_2, 1
        ... X[j_1, j_2] ...
        ... X[j_1 - 1, j_2 + 1] ...
```
$\Longrightarrow$

```
for i_1 = 1, N_1, T_1
    for i_2 = 1, N_2, T_2
        for ii_1 = i_1, max{N_1, i_1 + T_1 - 1}, 1
            for ii_2 = i_2, max{N_2, i_2 + T_2 - 1}, 1
                ... X[ii_1, ii_2] ...
                ... X[ii_1 - 1, ii_2 + 1] ...
```

Figure 1: *An example loop nest written in a pseudo high-level language and its blocked (tiled) version. Each data tile is of size $T_1 \times T_2$ array elements, and the transformed loop nest is structured based on that. Note that, for clarity, we do not show the explicit data block transfers between the SPM and off-chip memory (which occur across the iterations of i_1 and i_2).*

called *data tiles*) are accessed. If no compression is used, the only task that needs to be performed by the compiler is to schedule explicit data block transfers between the SPM and the off-chip memory. This can be done statically or dynamically. In the static approach, the compiler analyzes the code and identifies the data blocks with high data reuse (i.e., hot blocks). These blocks are then permanently assigned to the SPM. In other words, once they are in the SPM, they remain there until the end of the program execution. As a result, this approach can be expected to be most successful in situations where there exist a few data blocks with very high reuse. In comparison, in the dynamic approach, the contents of the SPM are varied as the execution moves from one phase of the computation to another. The goal behind this scheme is to capture the dynamic variations in data reuse. In either case, the success of a SPM management strategy depends strongly on an important factor: being able to keep data with high reuse in the SPM as much as possible. Clearly, the available SPM space plays an important role here since if the total size of the data blocks with high reuse is larger than the available SPM capacity, some critical data blocks have to be left out. Obviously, this can have a devastating impact on both performance and energy consumption, as it leads to frequent off-chip accesses for data.

It is possible to increase effective SPM capacity by making use of data block compression since, at any given time, not all the data blocks in the SPM have the same criticality. For example, while some of them might be in active use, others will not be accessed until some time in the future. Therefore, it might be possible to keep the latter type of data blocks in a compressed form to increase the available space for new data blocks. The drawback of compression is that if such a (compressed) block needs to be accessed later, it first needs to be decompressed, which typically takes time and consumes energy. Because of this, the number of decompressions should be minimized as much as possible. In fact, for such a SPM management strategy to be successful, the cost of decompressing a data block within the SPM must be lower than accessing it (in an uncompressed form) from the off-chip memory. As an example, consider a 200MHz embedded processor and a decompression rate of 20MB/sec (similar to the LZO algorithm [14]). Decompressing a 1KB compressed data block (to 2KB) within the SPM takes about 20,000 cycles. Instead, assuming a 100 cycle off-chip memory latency, bringing a 2KB block from the off-chip memory consumes 50,000 cycles (assuming a 1-word wide bus). In other words, if we find the requested data block in the SPM in the compressed form, it will be 2.5 times faster than accessing the same block from the off-chip memory. Moreover, our compiler algorithm tries to keep the hot data blocks uncompressed as much as possible, and only compresses the data blocks that will not be needed for some time. Also, the off-chip memory accesses keep getting more and more expensive in terms of processor cycles (as a result of increased clock frequencies) and energy. Therefore, one might expect a compression-based SPM management scheme to be even more attractive in the future.

In managing a SPM based on data compression, one needs to make several important decisions: (1) When a new data block is created, should it be compressed, or be left uncompressed? (2) Where should a newly-created data block be stored – in the SPM,
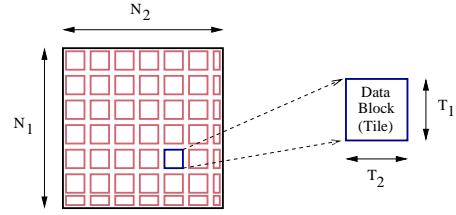

Figure 2: *Dividing a two-dimensional array into data blocks (tiles). All the data blocks have the same size except possibly at the boundaries of the array.*

or in the off-chip memory? (3) When the SPM is full and a new data block needs to be brought in (from the off-chip memory), what should be done? Should we kick out the unused (SPM) blocks to the off-chip memory, or should we compress them within the SPM itself? (4) When the current use of a data block is over, should we compress it or not? In the rest of this paper, we present a compiler-based SPM management heuristic based on data compression. In our strategy, the decisions mentioned above are made by an *optimizing compiler* based on a thorough analysis of the application at hand. Our experiments with several applications show that data compression can be very useful in increasing the benefits coming from a SPM.

3. DETAILS OF OUR APPROACH

3.1 Preliminaries

We focus on programs constructed using loop nests (with compile-time known bounds) and array accesses (with affine subscript expressions). Figure 1 shows an example loop nest that accesses an array X through two references with affine subscript expressions ($X[j_1, j_2]$ and $X[j_1 - 1, j_2 + 1]$). In order to make use of a SPM, an array is (logically) divided into *data blocks* (*data tiles*), which are the *unit of transfer* between the SPM and the off-chip memory (see Figure 2). When an array element that is not in the SPM is required by the current computation, the corresponding data block is brought into the SPM. The transformed code in Figure 1 gives the blocked (tiled) version of the original loop nest. In this code, loops i_1 and i_2 iterate over the data blocks, and are called the *inter-tile iterators* (or *block iterators*). In comparison, ii_1 and ii_2 are referred to as the *intra-tile iterators,* and iterate over the elements of a given data block (indexed by i_1 and i_2). The explicit data block transfers between the SPM and the off-chip memory occur only across the iterations of i_1 and i_2.

Self temporal reuse is said to exist when an array reference in a loop nest accesses the same data in different loop iterations. Similarly, if a reference accesses nearby data in different iterations, we say that there exists *self spatial reuse* [21]. It should be emphasized that the most useful forms of data reuse (temporal or spatial) are those exhibited by the innermost loop. Therefore, a good SPM management strategy is the one that converts temporal and spatial data reuses into data locality. That is, the available SPM space should be managed in such a way that the vast majority of the data requests from the CPU should be satisfied from the SPM.

3.2 Reuse Vectors at the Data Tile Level

In this subsection, we give three crucial mathematical definitions that help us formulate the problem of utilizing the available SPM space in the most efficient way using a compression/decompression based scheme.

Definition: A *block iteration vector* (*BIV*) is a vector each entry of which has a value from the block iterator. For example, in the blocked loop nest shown on the right of Figure 1, $\vec{I} = (4 \quad 2)^T$ represents a BIV, and corresponds to the execution of the blocked loop nest with $i_1 = 4$ and $i_2 = 2$.

Definition: A *block-level reuse vector* (*BRV*) is defined to be the difference between two BIVs that access the same data block. Specifically, if $\vec{I_1}$ and $\vec{I_2}$ are two BIVs that access the same data block and $\vec{I_2} > \vec{I_1}$, then $\vec{r} = \vec{I_2} - \vec{I_1}$ is the corresponding BRV. As

an example, consider the blocked loop nest on the right of Figure 1. The data block whose first element is $X[a, b]$ is first accessed by the block iteration vector $\vec{I_1} = (k_1 \quad l_1)^T$ such that $k_1 = a$ and $l_1 = b$. The reuse of the same data block occurs when the block iteration vector $\vec{I_2} = (k_2 \quad l_2)^T$ is executed, where $k_2 - 1 = a$ and $l_2 + 1 = b$. Since $k_2 = a + 1$ and $l_2 = b - 1$, the BRV between them can be calculated as $\vec{r} = \vec{I_2} - \vec{I_1} = (a+1 \quad b-1)^T - (a \quad b)^T = (1 \quad -1)^T$. That is, the data block has a reuse distance of $(1 \quad -1)^T$. Note that, in this example, the first use of the data block in question occurs via the first reference in the loop nest, whereas the second use (reuse) occurs via the second reference.

Thus, a BRV captures the distance between the successive accesses to the same data block. Therefore, it can be used to guide compression/decompression decisions. For example, if $\vec{r}$ is very (lexicographically) large, it indicates that it may not be a good idea to keep the corresponding data block in the SPM in an uncompressed form (since otherwise it would be occupying SPM space without much use). In contrast, a small $\vec{r}$ value indicates a nearby data reuse. However, while BRV captures the data reuse from the viewpoint of a single data block, since the SPM is a shared space (among potentially all data blocks), we need a modified reuse vector definition that can enable us rank different data blocks according to their relative importance (criticality) at any given point during the execution. This observation leads us to our next definition.

Definition: Let $\vec{t}$ represent the current point in execution in terms of block iterators. The *next reuse vector* (*NRV*) of a data block is the difference between the next use of the block (specified in terms of a block iteration vector) and $\vec{t}$. That is, if the data block in question is next used by iteration $\vec{I}$, then the corresponding NRV can be calculated as $\vec{n} = \vec{I} - \vec{t}$. Note that $\vec{I} > \vec{t}$. As an example, suppose that $\vec{t} = (2 \quad 3)^T$ and, for a given data block, its next reuse occurs when $\vec{I} = (5 \quad 3)^T$. Based on this, we can calculate the next reuse vector as $\vec{n} = (5 \quad 3)^T - (2 \quad 3)^T = (3 \quad 0)^T$.

3.3 Data Block Ranking Based on NRVs

The NRVs of different data blocks can be used to rank them according to their relative criticality. To illustrate this, let us assume that there are K different data blocks (in the compressed form or not) in the SPM at execution point $\vec{t}$ (again, defined in terms of block iterators), and that a new data block needs to be brought in. Assuming that there is not enough space in the SPM to accommodate the incoming block, we need to create space for it. One way of solving this problem is to select a *victim data block* and either compress it within the SPM or send it to the off-chip memory. If we can calculate NRVs $\vec{n_1}$, $\vec{n_2}$, ..., $\vec{n_K}$ for data blocks 1, 2, ..., K that are currently in the SPM and compare them with each other lexicographically, we can identify the largest one and select the associated data block as the victim. Put another way, the victim is selected to be data block j, where $\vec{n_j} > \vec{n_k}$ for all $k \neq j$ and $1 \leq k, j \leq K$. Obviously, repeating such a calculation/comparison sequence at each execution step $\vec{t}$ at runtime could be intolerable from a latency point of view. Therefore, in the proposed strategy, the compiler *precomputes* these decisions at compile-time and comes up with a *schedule*. This schedule (which can also be considered as a lookup table) is represented using a data structure (to be described shortly) and can be executed at runtime without much overhead. In a sense, the workload in our approach is divided between the compiler and the runtime system. The compiler is responsible for generating a suitable schedule at compile-time, and the runtime system is responsible for executing it (i.e., moving data blocks between the SPM and off-chip memory and compressing/decompressing them as deemed necessary).

Ideally, at any given point during the execution, our compiler algorithm tries to achieve the situation depicted in Figure 3. In this figure, the data blocks, ordered from left to right according to non-decreasing NRVs, are divided into three groups. The first group (the one on the left), that consists of blocks that will be referenced soon (since their NRVs are lexicographically small), are in the SPM in an uncompressed form. Note that keeping such blocks compressed would cost lots of execution cycles to be spent in decom-
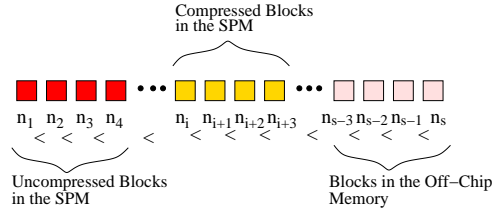

Figure 3: *A snapshot during execution illustrating how and where s data blocks are kept based on their NRVs. Note that the blocks are ordered from left to right according to (lexicographically) non-decreasing NRVs.*

pression. The second group (in the middle) holds the data blocks whose NRVs are not as small as those in the first group but not very large either. These blocks are good candidates to be kept in the SPM in the compressed form. The last group of blocks (on the right) are the ones with large NRVs that should reside in the off-chip memory since storing them in the SPM would waste SPM space without much benefits. Our current implementation is reluctant in sending data blocks to the off-chip memory; rather, if compressing a data block with large NRV will create enough space to accommodate the newly-requested/created block, it employs compression.

4. COMPILER ALGORITHM

Our implementation operates under an assumption: a data block is operated on only from the SPM; that is, if the execution needs a data block that is currently residing in the off-chip memory, the block first needs to be brought into the SPM. However, with appropriate modifications to the algorithm, we can drop this assumption.

Our compiler algorithm makes an important decision for three different (but similar) types of activities: (1) when creating a new data block; (2) when bringing a data block from the off-chip memory to the SPM; and (3) when decompressing a data block within the SPM. In all these cases, the compiler first computes the NRVs for all the current residents of the SPM and determines the one with the (lexicographically) largest NRV. After that, the compiler checks whether the victim selected has already been compressed. If this is the case, it is sent to the off-chip memory. If this is not the case, the compiler checks whether compressing the selected victim block in the SPM would create sufficient space for the incoming data block. If so, the victim is compressed within the SPM and the new block is brought in. If compression cannot create sufficient free space, then the compiler can potentially consider two alternatives: (1) it checks whether sending the selected victim to the off-chip memory would create sufficient space for the incoming data block, and (2) it identifies a second victim (i.e., the one with the second largest NRV among the SPM residents) and calculates the potential SPM space saving if this second victim is also compressed (along with the first victim) within the SPM. Our current implementation favors (2) since it is oriented to use the SPM space more aggressively, and our initial assessment has indicated that it generally generates better results than (1). If the victim block (when compressed within the SPM or sent to the off-chip memory) is not sufficient to create SPM space needed, the compiler selects another victim, and repeats the same process. In other words, as long as there is not enough SPM space, we keep selecting a new victim block, and either send it to the off-chip memory or compress it within the SPM. However, if a (victim) block is compressed, we count this as an access to that block, and this delays its transfer to the off-chip memory. Note also that this algorithms first tries to compress as many SPM-resident blocks as possible, before starting to send blocks to the off-chip memory. In other words, it utilizes the SPM space as much as possible. An additional optimization we employ is that the selected victim is directly sent to the off-chip memory (after the compression) if it has no next reuse in the rest of the execution.

The data structure constructed by the compiler is illustrated in Figure 4. This is an array of records; each of the three entries of a record instance is the head of a linked list. The first of these linked lists keeps the ids of the data blocks that need to be brought from the
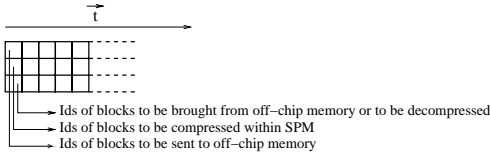
Ids of blocks to be brought from off–chip memory or to be decompressed
Ids of blocks to be compressed within SPM
Ids of blocks to be sent to off–chip memory

Figure 4: *The schedule data structure constructed by the compiler. This data structure is built at compile time but the schedule it contains is executed at runtime.*

```
for each nest ∇ᵢ:
    tile ∇ᵢ;
    identify data blocks δᵢ₁, δᵢ₂, ..., δᵢfᵢ
endfor
for each tiled nest ∇ᵢ:
    identify block iterator bounds b⃗ᵢₗ and b⃗ᵢᵤ
    for each t⃗ᵢ from b⃗ᵢₗ to b⃗ᵢᵤ − 1
        determine required data blocks μᵢ₁, μᵢ₂, ..., μᵢgᵢ by t⃗ᵢ
        t⃗ᵢ.required = ids of μᵢ₁, μᵢ₂, ..., μᵢgᵢ
        for each μᵢₖ, where 1 ≤ k ≤ gᵢ:
            repeat
                select a victim block v
                if v is compressed:
                    t⃗ᵢ.to-be-sent += v
                else
                    t⃗ᵢ.compressed += v
                endif
            until SPM space is available
        endfor
    endfor
endfor
```

Figure 5: *Algorithm for determining a schedule for data transfers and compressions/decompressions. "+=" indicates "adding a new block id to the linked list in question."*

off-chip memory or to be decompressed within the SPM (i.e., the blocks requested at that step during the execution). Note that since the ids of the blocks to be decompressed are kept in the schedule, at runtime, we can easily check whether a SPM access entails decompression. The second list keeps the ids of the blocks that need to be compressed within the SPM. And, finally, the third list keeps the ids of the blocks that need to be sent to the off-chip memory. The compiler fills this data structure conservatively at compile-time, and restructures the code accordingly. It needs to be conservative as it may not have complete information about the possible branch outcomes. One might argue that, given ever increasing array sizes and loop iteration counts in array-intensive embedded applications, this data structure can be very large. However, this is not the case, mainly because $\vec{t}$ iterates over the data blocks (i.e., not on individual array elements), and given that data blocks can be large, the number of different vectors taken on by $\vec{t}$ (which are determined by the block iterator bounds) is not very large.

The overall compiler algorithm is given in Figure 5. The algorithm iterates over all the nests in the application code and over all potential $\vec{t}$ vectors for a given nest. In this algorithm, $\vec{t_i}$.required, $\vec{t_i}$.compressed, and $\vec{t_i}$.to-be-sent correspond to the three linked lists explained in the previous paragraph. Note that we need to create a space in the SPM for each of the data blocks whose id is in the $\vec{t_i}$.required list, and this is achieved by the algorithm within the second for loop in the algorithm code. The algorithm terminates when all values that can be taken on by $\vec{t_i}$ are iterated. The asymptotic complexity of this algorithm is $\mathcal{O}(\mathcal{N}\mathcal{B}^2)$, where $\mathcal{N}$ is the number of nests in the application and $\mathcal{B}$ is the maximum number of data blocks. As will be discussed later in the experimental results section, in practice, this algorithm increases the original compilation time by 47.3% in the worst case, and is much faster than an ILP-based solution that derives a slightly better schedule.

5. EVALUATION

5.1 Setup

For our experiments, we modeled a processor architecture similar to that of Intel StrongARM SA-1100 clocked at 200MHz by enhancing the ARM-ISA extended [1] SimpleScalar simulator [2]. The modeled architecture has a SPM and an off-chip memory. The default capacity of the SPM (with 1-cycle access latency) is 4KB, and the off-chip memory is assumed to have a 100-cycle access latency. In order to compare this SPM-based architecture to a conventional data cache-based system, we also simulated a cache based version of the same architecture. The simulated data cache (with 1-cycle access latency) is assumed to be 4-way set-associative with a line size of 32 bytes. In order to make a fair comparison, we kept the cache and SPM sizes equal. Also, for both the architectures, we modeled a 256-entry fully-associative TLB with a 100-cycle miss latency. In the rest of this paper, the data cache-based architecture/execution is referred to as the *base architecture/execution*. We use an LZO compression/decompression algorithm [14] to handle compressions and decompressions; the decompression rate of this algorithm is around 20 MB/sec. It is to be emphasized that while, in this particular implementation, we chose a software-based compression/decompression, our approach can also accommodate a hardware-based compressor/decompressor (e.g., similar to that proposed in [6]). In such a case, we could even perform decompressions before the block is actually needed, thereby taking the decompression cost out of the critical path. The data block size used in our experiments is 512 bytes.

To test the effectiveness of our compression based approach, we used a benchmark suite that consists of six array-based embedded applications. The binaries for these codes are built on a NetWinder Developer ARM workstation from Rebel Systems (www.rebel.com) using GNU GCC version 2.95.1 with optimizations enabled (-O), and compiled to use the Linux/ARM system calls. The important characteristics of these codes are given in Figure 6. The third and fourth columns in this table give, respectively, the number of execution cycles and the number of off-chip data references under the *base execution*. All execution cycle results presented in Section 5.2 are given as a fraction of the values shown in the third column.

We performed experiments with three different SPM-based versions of each application. The *static* scheme decides what should be stored in the SPM by statically analyzing the program and determining the most frequently accessed data blocks execution wide. Once these data blocks are identified, they are brought into the SPM and remain there until the end of the execution. In determining the most frequently accessed blocks, we used profile data. In the *dynamic* scheme, the contents of the SPM are determined dynamically during the course of the execution. The specific strategy used here is very similar to that presented in [11]. Neither of these two strategies employs data compression. Finally, *compressed* is used to denote the heuristic described in this paper. All the code modifications necessary to implement these three schemes have been fully automated within the SUIF compiler [3]. Also, before the codes are modified for the SPM, they have been optimized for data locality using a set of loop and data transformations (this also helps improve the behavior of the base execution significantly).

5.2 Results

We start by comparing three different approaches to SPM management. The first and second bars in the graph given in Figure 7 represent the normalized execution cycles obtained through the static and dynamic schemes, respectively. The third bar in the graph corresponds to the results obtained by the approach presented in this paper. These results clearly show that the compression-based strategy generates much better results than the other two. Specifically, the average execution cycle improvements due to static, dynamic, and compression-based strategies are 22.22%, 36.81%, and 49.15%. In fact, we see that the compression-based strategy generates the best results for all six embedded benchmarks. The last bar for each benchmark will be discussed later.

In order to create space in the SPM for an incoming (or decompressed) data block, our approach uses compression and (if this is not successful) off-chip storage (i.e., it sends the victim block to the off-chip memory). A classical dynamic approach to SPM management that does not use compression/decompression performs only off-chip stores to create SPM space. To better illustrate why our compression-based approach outperforms the classi-

Benchmark Name	Brief Description	Number of Cycles	Number of Off-Chip References
des	DES crypto algorithm	1,709.44	316.26
disc	Speech/music discriminator	286.67	97.85
hier	Motion estimation algorithm	1,522.17	258.07
jpeg	Lossy compression for still images	31.61	38.73
pgp	IDEA/RSA public-key encryption	422.48	57.40
rasta	Speech recognition	205.33	38.15

Figure 6: *Benchmarks used in our experiments and their important characteristics.*
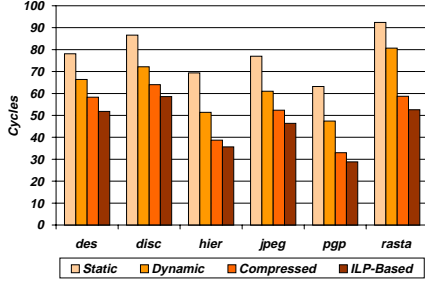
Figure 7: *Normalized execution cycles with different SPM management schemes.*

cal dynamic approach, we present in Figure 8 the number of compressions and off-chip stores performed by our approach over the time for one of our benchmarks (disc). The x-axis in this figure represents the time divided into epochs. The y-axis gives cumulative dynamic compressions and off-chip stores. We can see that our approach uses compression very heavily, and since decompression is cheaper than reading data from the off-chip memory, this brings large performance benefits at runtime. In a dynamic approach that does not use compression, all compressions would most probably be off-chip stores, explaining the performance difference between the approaches. The success of our compression-based SPM management strategy depends critically on the number of accesses made to the compressed data blocks. If this number is high, it can potentially offset the benefits that could come from compression. This is reason why our approach tries to keep the hot data blocks in an uncompressed form, whereas the non-hot SPM-resident blocks are kept in the compressed form.

6. COMPARISON WITH AN ILP-BASED APPROACH

Our ILP solution takes as input the compiler-derived data block access pattern information (the same information used by the compiler to construct the schedule). Note that this access pattern information indicates the data blocks that need to be read/written at each step. The output of the ILP solution is an optimal schedule. It should be mentioned that the ILP formulation to be presented below is more general than necessary in that it also allows operating on data blocks while they are in the off-chip memory. However, for our experiments, for a fair comparison with the heuristic approach, we modified this formulation so that the execution operates only on the SPM-resident data blocks.

Let s be the number of scheduled steps in the execution, d the number of different arrays, and b_i the number of data blocks for array i. Our goal is to use 0-1 integer variables to determine the optimal points in execution for transferring data blocks between the off-chip memory and SPM and for scheduling compressions and decompressions. In this particular problem, we define our 0-1 variables as follows:

• $US_{j,k,b}$: indicates whether at step j block b of array k is not compressed and in the SPM.

• $UM_{j,k,b}$: indicates whether at step j block b of array k is in the off-chip memory.

• $CS_{j,k,b}$: indicates whether at step j block b of array k is compressed form and in the SPM.
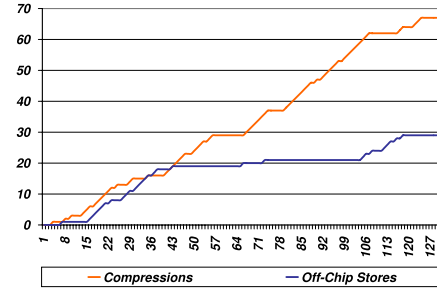
Figure 8: *Compressions and off-chip stores for disc. Each point on either curve denotes a cumulative number over the execution time.*

Having defined our 0-1 variables, we now explain our ILP formulation. Since a data block of an array can be in one of two locations (the SPM or memory) and in one of two forms (compressed or uncompressed) at a given time, the following equation should be satisfied:

$$US_{j,k,b} + UM_{j,k,b} + CS_{j,k,b} = 1 \tag{1}$$

However, a modification is necessary whenever a data block is accessed. Since read/write can only be performed when the block is not in the compressed form, if, at any given step j' where $0 \le j' \le s$, block b of array k is accessed, we have:

$$US_{j',k,b} + UM_{j',k,b} = 1 \tag{2}$$

Another important constraint that has to be satisfied is regarding the SPM size. More specifically, we cannot assign more blocks to the SPM than it can hold. Assuming that the size of the SPM is L, we can write the following equation for every step, where $csize_k$ is used to represent the compressed size of a block of array k, and $size_k$ is used to denote the uncompressed size of a block of array k:

$$\sum_{k=0}^{d-1} \sum_{b=0}^{b_k-1} (CS_{j,k,b} \times csize_k + US_{j,k,b} \times size_k) \le L, \forall j \tag{3}$$

After specifying the constraints that need to be satisfied for any valid schedule, we next formulate the *cost expression*. Basically, there are five components of the cost: read cost, write cost, transfer cost (between the SPM and off-chip memory), compression cost, and decompression cost. The read cost is incurred when the block is read from the SPM or off-chip memory, and can thus be expressed as:

$$P_R = \sum_{j=0}^{s-1} \sum_{k=0}^{d-1} \sum_{b=0}^{b_k-1} R_{j,k,b} \times (US_{j,k,b} \times CRUS + UM_{j,k,b} \times CRUM) \tag{4}$$

In this equation, $R_{j,k,b}$ is used to indicate that, at step j, the CPU reads data block b of array k. Note that $R_{j,k,b}$ (that can be either 1 or 0) is extracted from the block access pattern information passed to the ILP solver. On the other hand, CRUS and CRUM give the cost of reading an uncompressed block from the SPM and off-chip memory, respectively. Similarly, we can express the write cost as follows:

$$P_W = \sum_{j=0}^{s-1} \sum_{k=0}^{d-1} \sum_{b=0}^{b_k-1} W_{j,k,b} \times (US_{j,k,b} \times CWUS + UM_{j,k,b} \times CWUM) \tag{5}$$

Here, $W_{j,k,b}$, CWUS, and CWUM are similar to $R_{j,k,b}$, CRUS and CRUM, except that they are defined for the write operation. The data transfer can be from the off-chip memory to the SPM, or vice versa. In addition, a data block can reside in the SPM in the compressed or uncompressed form. Consequently, we can formu-

late the transfer cost as:[1]

$$P_M = \sum_{j=1}^{s-1}\sum_{k=0}^{d-1}\sum_{b=0}^{b_k-1}(((US_{j-1,k,b}||CS_{j-1,k,b})\&\&UM_{j,k,b})|| \quad (6)$$

$$((US_{j,k,b}||CS_{j,k,b})\&\&UM_{j-1,k,b})) \times CM \quad (7)$$

In this last formulation, CM is used to denote the transfer cost (for a given data block). The last two components of the overall cost are the compression and decompression costs (denoted P_C and P_D, respectively), and can be written as follows:

$$P_C = \sum_{j=1}^{s-1}\sum_{k=0}^{d-1}\sum_{b=0}^{b_k-1}((US_{j-1,k,b}||UM_{j-1,k,b})\&\&CS_{j,k,b}) \times CC \quad (8)$$

$$P_D = \sum_{j=1}^{s-1}\sum_{k=0}^{d-1}\sum_{b=0}^{b_k-1}(CS_{j-1,k,b}\&\&(US_{j,k,b}||UM_{j,k,b})) \times CD \quad (9)$$

In these expressions, CC and CD indicate the cost of compressing and of decompressing one block, respectively. Based on the cost components described above, one can express the overall cost as:

$$P_{total} = P_R + P_W + P_M + P_C + P_D$$

Therefore, our ILP problem can be defined formally as one of minimizing P_{total} under the constrains stated in (1), (2), and (3).

Note that our ILP-based formulation is different from the prior work [17] as it focuses on scheduling not only data transfers but also compressions and decompressions. The last bars in Figure 7 represent the normalized execution cycles for the ILP-based solution described above. We see that the average execution cycle reduction due to the ILP-based scheme is around 54.37%, that is not excessively larger than the corresponding saving with the compression-based heuristic (that is 49.15%). The reason that the heuristic approach is not quite as good as the ILP-based one is that in some cases trying to keep the data in the SPM in the compressed form aggressively (instead of just forwarding it to the off-chip memory) hurts performance. However, the compiler-based algorithm reaches a solution in a much shorter time as compared to the ILP-based scheme. Specifically, the average percentage increases in compilation time (over the base scheme that does not use compression) due to the heuristic strategy were 47.3%, 31.1%, 36.2%, 28.8%, 44.0%, and 26.9% for des, disc, hier, jpeg, pgp and rasta, respectively. The corresponding percentage increases for the ILP-based scheme were 487.1%, 392.6%, 886.9%, 721.0%, 1,024.2%, and 563.3%. In other words, the compiler-based scheme achieves comparable results with the ILP-based scheme while spending much less time in compilation.

7. CONCLUSIONS

This paper demonstrates that data compression can be a viable technique for increasing the effectiveness of SPMs. Our compression-based heuristic brings around 25% reduction in execution cycles over a previously-proposed dynamic SPM management scheme, and is only 9.2% worse than an ILP-based optimal scheme. It achieves this by keeping as many data blocks as possible in the SPM but ensuring that frequently accessed data blocks are not frequently compressed or decompressed. The scheme proposed in this paper is different from the prior studies on SPM management in that it uses compression/ decompression to better utilize the available SPM space.

8. REFERENCES

[1] T. M. Austin. The SimpleScalar/ARM Toolset. http://www.eecs.umich.edu/~taustin/simplescalar

[2] T. M. Austin and D. Burger. The SimpleScalar Architectural Research Tool Set. http://www.cs.wisc.edu/~mscalar/simplescalar.html

[3] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng The SUIF compiler for scalable parallel machines. In Proc. *the 7th SIAM Conference on Parallel Processing for Scientific Computing,* February, 1995.

[4] N. Bellas, I. N. Hajj, C. Polychronopoulos, and G. Stamoulis. Energy and performance improvements in microprocessor design using a loop cache. In Proc. *International Conference on Computer Design,* 1999, pp. 378–383.

[5] L. Benini, A. Macii, E. Macii, and M. Poncino. Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation. *IEEE Design & Test of Computers,* pages 74–85, April-June, 2000.

[6] L. Benini, D. Bruni, A. Macii, and E. Macii. Hardware-assisted data compression for energy minimization in systems with embedded processors. In Proc. *DATE'02,* Paris, France, March 2002.

[7] K. D. Cooper and T. J. Harvey. Compiler-controlled memory. In Proc. *the International Conference on Architectural Support for Prog. Lang. and Operating Systems,* CA, November 1998.

[8] CPU12 Reference Manual. Motorola Corporation, 2000. http://motorola.com/brdata/PDFDB/ MICROCON-TROLLERS/16_BIT/68HC12_FAMILY/REF_MAT/CPU12RM.pdf.

[9] E. G. Hallnor and S. K. Reinhardt. A fully-associative software-managed cache design. In Proc. *International Conference on Computer Architecture,* pp. 107–116, Vancouver, British Columbia, Canada, 2000.

[10] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In Proc. *the 39th Design Automation Conference,* 2002.

[11] M. Kandemir, J. Ramanujam, M. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In Proc. *the 38th Design Automation Conference,* June 2001.

[12] M. Kandemir, J. Ramanujam, and A. Choudhary. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In Proc. *the 39th Design Automation Conference,* June 2002.

[13] L. H. Lee, B. Moyer and J. Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In Proc. *International Symposium on Low Power Electronic Design,* San Diego, CA, August, 1999.

[14] http://www.oberhumer.com/opensource/lzo/

[15] M-CORE – MMC2001 Reference Manual. Motorola Corporation, 1998. http://www.motorola.com/SPS/MCORE/info_documentation.htm.

[16] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad-memory in embedded processor applications. In Proc. *European Design and Test Conference,* Paris, March 1997.

[17] S. Steinke et al. Reducing energy consumption by dynamic copying of instructions onto on-chip memory. In Proc. *ISSS'02,*, Kyoto, Japan, October 2002.

[18] TMS370Cx7x 8-bit Microcontroller. Texas Instruments, Revised February 1997. http://www-s.ti.com/sc/psheets/spns034c/spns034c.pdf.

[19] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In Proc. *the ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems,* 2003.

[20] L. Wang, W. Tembe, and S. Pande. Optimizing on-chip memory usage through loop restructuring for embedded processors. In Proc. *9th International Conference on Compiler Construction,* March 30–31 2000, pp. 141–156, Berlin, Germany.

[21] M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison Wesley, 1996.

---

[1] Actually, in our ILP formulation, the boolean expressions "and" ($\&\&$) and "or" ($||$) are expressed using extra 0-1 variables. Considering expression (7) for example, there is the term $CS_{j-1,k,b}\&\&UM_{j,k,b}$. This means that data block b of array k is compressed and in the SPM at step $j-1$, and it is moved to the off-chip memory at the next step. This is expressed in our ILP formulation with 0-1 variable $CSUM_{j,k,b}$ as follows: $CSUM_{j,k,b} \geq CS_{j-1,k,b} + UM_{j,k,b} - 1$.