Compiler-Guided Data Compression for Reducing Memory Consumption of Embedded Applications *

O. Ozturk, G. Chen, and M. Kandemir

Computer Science and Engineering Department
Pennsylvania State University
e-mail: {ozturk,gchen,kandemir}@cse.psu.edu

I. Kolcu

Computation Department
University of Manchester
e-mail: ikolcu@umist.ac.uk

Abstract— Memory system presents one of the critical challenges on embedded system design and optimization. This is mainly due to ever-increasing code complexity of embedded applications and exponential increase witnessed in the amount of data they manipulate. As a result, reducing memory space occupancy of embedded applications is very important and will be even more important in the next decade. Motivated by this observation, this paper presents and evaluates a compiler-driven approach to data compression for reducing memory space occupancy. Our 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 achieves this by analyzing the source code of the application to be optimized and identifying the order in which the different data blocks are accessed. Based on this analysis, the compiler then automatically inserts compression/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.

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 code complexity of embedded applications and amount of data they process in a typical execution are increasing. 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, took relatively less attention so far.

One of the techniques that can be used to reduce 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 memory space consumption of embedded applications, it needs to be invoked 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 data access pattern of the application.

Our 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 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 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 explain how it operates in practice. In providing 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; that is, the programmer gets the benefits of reducing the memory space consumption without any effort on her part. Second, we want to minimize the potential impact of compression on performance. To do this, our automated approach makes use of data reuse analysis. It needs to be emphasized that in this paper we do not propose a new data compression technique. Instead, we demonstrate how an optimizing compiler for embedded systems can schedule data compressions and decompressions intelligently to minimize memory space occupancy at runtime while also minimizing the associated performance overheads.

The remainder of this paper is structured as follows. In the next section, we give a discussion of the related work on compression. In Section III, we define concept of memory space consumption (occupancy). In Section IV, we present our compiler algorithm. Finally, we conclude the paper in Section V with a summary.

II. Related Work

Compression techniques have been used for both program code and application data to reduce the memory footprint and the energy consumption. RISC processors have been the main focus for code compression techniques. To reduce the size of a program's code segment, [9] uses pattern-matching techniques that identify and coalesce together repeated instruction sequences. A RISC system that can directly execute com-

*This work is supported in part by NSF Career Award #0093082 and by a grant from GSRC.

0-7803-9451-8/06/$20.00

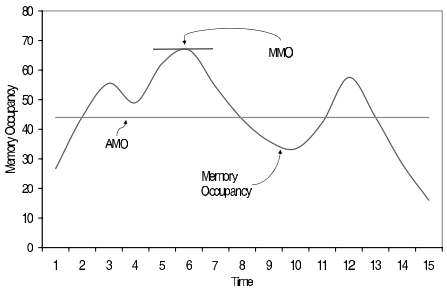

Fig. 1. An example memory space consumption curve and MMO and AMO metrics.

pressed programs is presented in [17]. VLIW (Very Long Instruction Word) processors are now being considered in this area as well. Ros and Sutton [15] present code compression algorithms applied to instruction words. Profile-driven methods that selectively compress code segments in VLIW architectures have also been proposed [18, 10]. Code compression schemes for VLIWs that use Variable-to-fixed (V2F) coding was investigated by [16, 19]. Variable-sized-block method for VLIW code compression has been introduced by Lin et al [12]. Data compression has been used to reduce storage requirements, bus bandwidth, and energy consumption in the past [1, 5, 6]. In [5], a hardware-assisted data compression for on-the-fly data compression and decompression has been proposed. In their work, compression and decompression takes place on the cache-to-memory path. Lee et al [11] use compression in an effort to explore the potential for an on-chip cache compression to reduce cache miss ratio and miss penalty. Abali et al [1] investigate the performance impact of hardware compression. Compression algorithms that would be suitable to use in a compressed cache have been presented in [2]. Apart from memory subsystems, data compression has also been used to reduce the communication volume. For example, data compression has been proposed to reduce the communication latency and energy consumption in sensor networks [7]. Our work is different from all these prior efforts since we give the task of management of the compressed data blocks to the compiler. In deciding the data blocks to compress and decompress, our compiler approach makes use of the data reuse information extracted from the array accesses in the application source code.

III. Memory Space Occupancy

Memory space occupancy indicates the memory space occupied by an 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. Figure 1 illustrates these two metrics for an example case. Note that, the 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 since 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 multi-programming based embedded environment where multiple applications compete for the same memory space. 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.

IV. Compiler Algorithm

Employing data compression in managing the memory space of the system requires a careful analysis of the data access pattern of the application being considered. This is because using 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. Also, 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. Our goal is to reduce MMO and AMO as much as possible, with as little performance penalty as possible.

A. 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). Figure 2 shows the organization 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.

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 intra-tile subscript vector, which indexes an element within a given tile. For example, Figure 3 shows an array X that has been divided into nine (3×3) tiles, and each tile contains sixteen (4×4) elements. $X[\![2,3]\!]$ 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 $X[\![2,3]\!]$.

Figure 2 shows how we store tiled arrays in the memory. For each compressible array X, our compiler creates a directory, each entry of which corresponds 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. As mentioned above, the directory of each array is stored in the static data area of the memory space.

An array tile can either be 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 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 a constant, 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 Figure 2, we can observe that the slices of the same tile form a link table. Like in the case of the decompression buffer, the compressed area also has a free table keeping all free slices.

Figure 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 the decompression buffer by a *decompressor* 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 emphasize here is that our approach is not tied to any specific compression/decompression algorithm, and the compressor and decompressor can be implemented either in software or hardware. In our current implementation, however, we use only software 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 data tiling. Our compiler automatically (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 a tile subscript vector and an intra-tile subscript vector, respectively. That is, we map $X[\vec{I}]$ into $X[\![p(\vec{I})]\!][q(\vec{I})]$. In this paper, given a tile

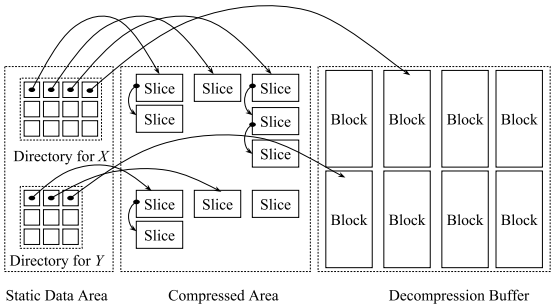

Fig. 2. Memory organization.

size T, we use the following mapping functions:

$$p((i_1, i_2, ..., i_n)) = (\lfloor i_1/N_1 \rfloor, \lfloor i_2/N_2 \rfloor, ..., \lfloor i_n/N_n \rfloor);$$
$$q((i_1, i_2, ..., i_n)) = (i_1 \mod N_1, i_2 \mod N_2, ..., i_n \mod N_n);$$

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

B. Loop Tiling

While data tiling transforms memory layout of each compressible array, loop tiling (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. Figure 5 gives such an example. Figure 5(a) is the original code of a loop nest, which accesses a 600×600 array X. We apply data tiling to array X such that the size of each tile is 100×100. Figure 5(b) shows the code after data tiling. For illustration purposes, let us assume that the decompression buffer can contain up to three tiles, and that we use an LRU 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 decompressor 100 times for each tile. Hence, the decompressor is invoked $100 \times 36 = 3600$ times. By applying loop tiling to the loop nest shown in Figure 5(b), we obtain the tiled loop nest in Figure 5(c). In this tiled code, loop iterators i and j are the *inter-tile iterators* and the loop nest formed by them is referred to as the *inter-tile loop nest.* Similarly, the iterators ii and jj are the *intra-tile iterators* and the loop nest formed by them is referred to as the *intra-tile loop nest.* During the execution of this loop nest, the decompressor is invoked only 36 times, once for each i, j combination. Loop tiling has been widely studied in the literature (e.g., [20]). Due to the space limitation we have, we do not discuss the details of loop tiling. 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. It is to be mentioned however that our compiler uses loop tiling for a different purpose than most of the commercial and academic compilers.

C. Compression-Based Space Management

Our compiler inserts buffer management code at each loop nest that uses the decompression buffer. For ease of discussion,

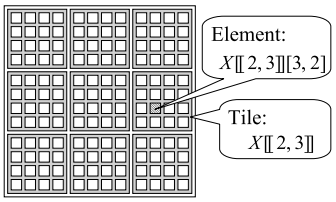

Fig. 3. Data tiling for array X.

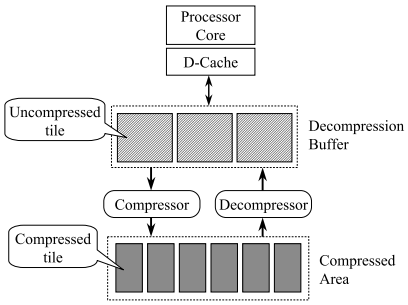

Fig. 4. Architecture supporting memory compression.

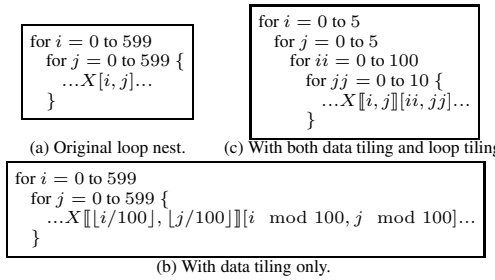

Fig. 5. Code transformation for data tiling and loop tiling.

we use the following abstract form to represent a tiled loop nest:

$$\begin{array}{l}\text{for } \vec{I} = \vec{L} \text{ to } \vec{U} \ \{\\ \quad T_1(R_1(\vec{I}), W_1(\vec{I}));\\ \quad T_2(R_2(\vec{I}), W_2(\vec{I}));\\ \quad \ldots\ \ldots\\ \quad T_n(R_n(\vec{I}), W_n(\vec{I}));\\ \}\end{array}$$

where $\vec{I}$ is the iteration vector, $\vec{L}$ and $\vec{U}$ are the lower and upper bound vectors for the loop nest. T_i $(i = 1..n)$ represents an intra-tile loop nest. Since we focus on the access pattern of each array at a tile level, we treat each intra-tile loop nests 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 intra-tile loop nest T_i at the inter-tile iteration $\vec{I}$. $R_i(\vec{I})$ and $W_i(\vec{I})$ can be computed as follows:

$$\begin{aligned} R_i(\vec{I}) &= \{X_k[\![f_j^{(i)}(\vec{I})]\!] \mid \text{"... = ...}X_k[\![f_j^{(i)}(\vec{I})]\!][...]\text{..." appears in } T_i\},\\ W_i(\vec{I}) &= \{X_k[\![f_j^{(i)}(\vec{I})]\!] \mid \text{"}X_k[\![f_j^{(i)}(\vec{I})]\!][...] = ...\text{" appears in } T_i\},\end{aligned}$$

where X_k is an array accessed by T_i, and $f_j^{(i)}(\vec{I})$ is a mapping function that maps inter-tile 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 intra-tile loop nest T_i, let us assume:

$$\begin{aligned} &W_i(\vec{I}) = \{X_{k_1}[\![f_1^{(i)}(\vec{I})]\!], X_{k_2}[\![f_2^{(i)}(\vec{I})]\!], X_{k_j}[\![f_j^{(i)}(\vec{I})]\!]\}\\ &R_i(\vec{I}) - W_i(\vec{I}) = \{X_{k_{j+1}}[\![f_{j+1}^{(i)}(\vec{I})]\!], X_{k_{j+2}}[\![f_{j+2}^{(i)}(\vec{I})]\!], X_{k_m}[\![f_m^{(i)}(\vec{I})]\!]\}.\end{aligned}$$

Figure 6 shows a transformed loop nest augmented with the decompression buffer management code. In the transformed code, we use counter c to count the number of intra-tile 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 intra-tile 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 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 number of intra-tile loop nests that have been executed, and $d_i(t)$ is the *reuse distance* of tile t at intra-tile loop nest T_i. The reuse distance of tile is the number of intra-tile loop nests executed between the current and the next accesses to this tile.

We use a compiler-based approach to compute $d_i(t)$–the reuse distance of tile t at intra-tile loop nest T_i. In the following discussion, we explain how $d_i(t)$ can be computed at compilation time. The tiles in $R_i(\vec{I}) \cup W_i(\vec{I})$, the set of tiles used by the i^{th} intra-tile loop nest at inter-tile loop iteration $\vec{I}$, can be divided into three types: (1) the tiles that will be reused by another intra-tile loop nest at the same inter-tile iteration $\vec{I}$, (2) the tiles that will be reused by some intra-tile loop nest at another inter-tile iteration $\vec{I}'$ $(\vec{I} \prec \vec{I}')$, and (3) the tiles that will never be reused. The set of tiles belonging to the first type, i.e., the tiles that will be reused at the same inter-tile loop iteration, can be computed as follows:

$$U_i(\vec{I}) = (R_i(\vec{I}) \cup W_i(\vec{I})) \cap \bigcup_{j=i+1}^{n} (R_j(\vec{I}) \cup W_j(\vec{I})),$$

where n is the number of intra-tile loop nests in the body of the inter-tile loop nest. For each tile $t \in U_i(\vec{I})$, the reuse distance can be computed as:

$$d_i(t) = j - i,$$

where j is the minimum integer greater than i such that $t \in R_j(\vec{I}) \cup W_j(\vec{I})$.

For the tiles in the set $V_i(\vec{I}) = (R_i(\vec{I}) \cup W_i(\vec{I})) - U_i(\vec{I})$, i.e., the tiles of types (2) and (3), our compiler computes their reuse distances by executing the loop nest below (this loop nest is generated by the compiler using Omega library [14],[1] and it is executed only once at the compilation time):

$$\begin{array}{l} V = V_i(\vec{I}_0);\ \ c = 0;\\ \text{for } \vec{I} = \vec{I}_0 + (0, 0, ..., 0, 1)^{\mathrm{T}} \text{ to } \vec{I}_N \ \{\\ \quad c = c + n;\\ \quad \text{for } i = 1 \text{ to } n \ \{\\ \quad\quad \text{for each } t \in V \cap (R_j(\vec{I}) \cup W_j(\vec{I})) \ \{\ d_j(t) = c + j - 1;\ \}\\ \quad\quad V = V - (R_j(\vec{I}) \cup W_j(\vec{I}));\\ \quad \}\\ \}\\ \text{for each } t \in V \ \{\ d_i(t) = \infty;\ \}\end{array}$$

[1]The Omega Library is a tool that provides functions for manipulating sets and relations that are defined using Presburger formulas. Presburger formulas are logical formulas that are built using affine expressions and universal/existential quantifiers.

```
for I⃗ = L⃗ to U⃗ {
  … …
  T_i(R_i(I⃗), W_i(I⃗));
  … …
}
```

(a) Original loop nest.

```
B — the set of tiles in the buffer;
D — the size of buffer;
t — the tile to be loaded;
⇒  procedure load(t) {
⇒    if(t ∉ B) {
⇒      if(|B| = D) {
⇒        for each v ∈ B such that v.r < c
⇒          v.r = ∞; // the next use time of v has been mispredicted
⇒        select v ∈ B such that
⇒          the directory for v is not marked and v.r is maximized;
⇒        if(v.w = 1) compress(v);
⇒        evict(v); B = B − {v};
⇒      }
⇒      decompress(t); B = B + {t};
⇒    }
⇒    t.r = c + d_i(t);
⇒  }

for I⃗ = L⃗ to U⃗ {
     … …
⇒    c = c + 1;
⇒    // mark the tiles in W_i(I⃗) ∪ R_i(I⃗),
⇒    // preventing these tiles from being evicted.
⇒    mark_directory_entry(X_{k_1}[[f_1^(i)(I⃗)]]);
⇒    mark_directory_entry(X_{k_2}[[f_2^(i)(I⃗)]]);
⇒    … …
⇒    mark_directory_entry(X_{k_m}[[f_m^(i)(I⃗)]]);

⇒    // load tiles in W_i(I⃗)
⇒    load(X_{k_1}[[f_1^(i)(I⃗)]]); X_{k_1}[[f_1^(i)(I⃗)]].w = 1;
⇒    load(X_{k_2}[[f_2^(i)(I⃗)]]); X_{k_2}[[f_2^(i)(I⃗)]].w = 1;
⇒    … …
⇒    load(X_{k_j}[[f_j^(i)(I⃗)]]); X_{k_j}[[f_j^(i)(I⃗)]].w = 1;

⇒    // load tiles in R_i(I⃗)
⇒    load(X_{k_{j+1}}[[f_{j+1}^(i)(I⃗)]]);
⇒    load(X_{k_{j+2}}[[f_{j+2}^(i)(I⃗)]]);
⇒    … …
⇒    load(X_{k_m}[[f_m^(i)(I⃗)]]);

⇒    // unmark the tiles in W_i(I⃗) ∪ R_i(I⃗),
⇒    // allowing these tiles to be evicted.
⇒    unmark_directory_entry(X_{k_1}[[f_1^(i)(I⃗)]]);
⇒    unmark_directory_entry(X_{k_2}[[f_2^(i)(I⃗)]]);
⇒    … …
⇒    unmark_directory_entry(X_{k_m}[[f_m^(i)(I⃗)]]);
     T_i(R_i(I⃗), W_i(I⃗));
     … …
}
```

(b) The transformed loop nest augmented with the decompression buffer management code. The lines marked with "⇒" are inserted by our compiler.

Fig. 6. Code transformation employed by our compiler.

In the above compiler-generated code, $\vec{I}_0$ and $\vec{I}_N$ are two vectors such that $|\vec{I}_N - \vec{I}_0| = N$ and $\vec{L} \preceq \vec{I}_0 \prec \vec{I}_n \preceq \vec{U}$, where $|\vec{I}_N - \vec{I}_0|$ denotes the number of loop iterations between $\vec{I}_0$ and $\vec{I}_N$, and $\vec{L}$ and $\vec{U}$ are, respectively, the lower and upper bound vectors for the target inter-tile loop nest for which we compute the reuse distances. $\vec{I}_0$ can be any vector between $\vec{L}$ and $\vec{U}$. Note also that integer N is a threshold, and n is the number of intra-tile loop nests in the body of the inter-tile loop nest. The reuse distances larger than nN are treated as infinity (∞). Note that an inaccuracy in computing the reuse distance may lead to performance penalties when the program is executed; however, it does not cause any error in the program execution (i.e., it is not a correctness issue), since we are conservative in computing the set of of tiles that are used in each intra-tile loop nest.

D. Exploiting Extra Resources

While the compiler approach presented above schedules compressions and decompressions such that memory space consumption is reduced without excessively increasing the original execution time, we can still incur performance penalties. This is because the decompression activities can occur on the critical path of execution and this in a sense symbolizes the tradeoff between memory savings and performance overheads due to data compression. In this section, we show how we can make use of extra resources available in our approach.

In a multiprocessor environment, we can reduce the performance overheads incurred by data compression by overlapping the execution of compression and decompression procedures with that of the computing loop nest. Specifically, for each inter-tile loop nest, our compiler generates two threads: the *computing thread* and the *buffer management thread*. In a multiprocessor based environment, these two threads can be executed in parallel. Figure 7 shows the code our compiler generates. Figure 7(b) gives the code for the buffer management thread. For each intra-tile iteration T_i at each inter-tile iteration $\vec{I}$, the buffer management thread decompresses each tile t in the set $R_i(\vec{I}) \cup W_i(\vec{I})$ if t is not in the buffer. The management thread increases $t.c$, the *reference counter* associated with tile t, by one for each $t \in R_i(\vec{I}) \cup W_i(\vec{I})$. The reference counter associated with each tile $t \in R_i(\vec{I}) \cup W_i(\vec{I})$ will be decreased by the computing thread after the execution of intra-tile T_i. A non-zero value in the reference counter $t.c$ indicates that tile t is being used or will be used by the computing thread, and consequently, t cannot be evicted from the buffer. On the other hand, when the buffer management thread needs to evict a tile from the buffer to make space for a new tile, it can evict any tile whose reference counter is zero (nevertheless, for better performance, as discussed in Section C, we also require $v.r$ be maximized). After increasing the reference counter for each tile in $R_i(\vec{I}) \cup W_i(\vec{I})$, the management thread performs a V operation on a counting semaphore named "Iteration". The value of this semaphore (Iteration.v) indicates the number of intra-tile loop nests that the computing thread can execute without being blocked. If the value of this semaphore is zero, the computing thread cannot continue with its execution due to the fact that some tiles required by the computing thread are not yet ready in the buffer. After the V operation on semaphore "Iteration", the management thread starts to decompress the tiles that will be used by the next intra-tile loop nest, without further synchronization with the computing thread.

Figure 7(c) gives the code (with necessary instructions inserted by our compiler) for the computing thread. Before executing each intra-tile loop nest T_i, the computing thread performs a P operation on the semaphore "Iteration". This P operation blocks the computing thread if some tiles that will be used by T_i are not ready in the buffer. In this case, the computing thread has to wait until the management thread decompresses all the required data tiles. After executing intra-tile loop nest T_i, the computing thread decreases the reference counter for each tile used by T_i. As discussed above, if the

```
// definition of semaphore operations
Struct Semaphore {
  int v;
  ThreadQueue q;
}

P(s) {
  s.v = s.v − 1;
  if(s.v < 0)
    block in s.q;
}

V(s) {
  s.v = s.v + 1;
  if(s.v ≤ 0)
    unblock one thread in s.q;
}


// semaphores for synchronizing the
// computing and management threads
Iteration – counting semaphore,
            Iteration.v is initialized to 0;
Counters – binary semaphore,
            Counters.v is initialized to 1;
```

(a) Semaphores and their operations.

```
for I⃗ = L⃗ to U⃗ {
  for i = 1 to n {
    c = c + 1;
    for each t ∈ R_i(I⃗) ∪ W_i(I⃗) {
      if(t ∉ B) {
        if(|B| = D) {
          repeat
            S = {v | v ∈ B ∧ v.c = 0}
          until S ≠ φ;
          select v ∈ S such that
            v.r is maximized
          if(v.w = 1) compress(v);
          evict(v); B = B − {v};
        }
        decompress(t);
        t.c = 0; B = B ∪ {v};
      }
      t.r = c + d_i(t);
      if(t ∈ W_i(I⃗)) t.w = 1;
      P(Counters); t.c = t.c + 1;
      V(Counters);
    }
    V(Iteration);
  }
}
```

(b) The buffer management thread.

```
for I⃗ = L⃗ to U⃗ {
  P(Iteration);
  T_1(R_1(I⃗), W_1(I⃗));
  P(Counters);
  for each t ∈ R_1(I) ∪ W_1(I)
    t.c = t.c − 1;
  V(Counters);
  … …
  … …
  P(Iteration);
  T_i(R_i(I⃗), W_i(I⃗));
  P(Counters);
  for each t ∈ R_i(I⃗) ∪ W_i(I⃗)
    t.c = t.c − 1;
  V(Counters);
  … …
  … …
  P(Iteration);
  T_n(R_n(I⃗), W_n(I⃗));
  P(Counters);
  for each t ∈ R_n(I) ∪ W_n(I)
    t.c = t.c − 1;
  V(Counters);
}
```

(c) The computing thread.

Fig. 7. The buffer management thread and the computing thread generated by our compiler.

value of the reference counter of tile t is reduced to zero, we allow the buffer management thread to reuse the memory space occupied by t.

The management thread is blocked when it needs to decompress a new tile but the value of the reference counter for each tile in the buffer is greater than 0 (that is, none of the tiles have been used yet). In this case, the management thread has to wait for the computing thread to release some tiles by reducing their reference counters. If the computing thread is also blocked at the P operation on semaphore "Iteration", the system is deadlocked. Fortunately, this deadlock cannot happen as long as the following condition is satisfied:

$$D \geq \max_{\forall i, \vec{I}} |R_i(\vec{I}) \cup W_i(\vec{I})|, \qquad (1)$$

where D is size of the buffer (in terms of the number of tiles). Note that, if this condition is not satisfied, the single-threaded approach discussed in Section C cannot work properly, either. It is important to note that, in our approach, the condition expressed by 1 is always satisfied.

V. Concluding Remarks

This paper presents 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. Our preliminary results are encouraging and motivate further research on compiler-directed data compression.

References

[1] 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. on Computers,* Nov 2001.

[2] E. Ahn, S. Yoo, S. Mo, and S. Kang. Effective algorithms for cache-level compression. In Proc. The 11th Great Lakes symposium on VLSI, 2001.

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

[4] T. M. Austin and D. Burger. The SimpleScalar architectural research tool set. http://www.cs.wisc.edu/∼mscalar/simplescalar.html

[5] L. Benini, D. Bruni, A. Macii, and E. Macii Hardware-Assisted Data Compression for Energy Minimization in Systems with Embedded Processors. In Proc. *Conf. on Design, Automation and Test in Europe,* 2002.

[6] C. D. Benveniste, P. A. Franaszek, and J. T. Robinson. Cache-Memory Interfaces in Compressed Memory Systems. In Proc. *Workshop on Solving the Memory Wall Problem,* June 2000.

[7] M. Chen and M. L. Fowler. The importance of data compression for energy efficiency in sensor networks. In Proc. *2003 Conference on Information Sciences and Systems,* The Johns Hopkins Univ., March 2003.

[8] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In Proc. *the SIGPLAN Conference on Programming Language Design and Implementation,* La Jolla, CA, June 1995.

[9] K. D. Cooper, and N. McIntosh. Enhanced code compression for embedded RISC processors. In Proc. *Conference on Programming Language Design and Implementation,* 1999.

[10] S. Debray, and W. Evans. Profile-Guided Code Compression. In Proc. *Conf. on Programming Language Design and Implementation,* 2002.

[11] J. S. Lee, W. K. Hong, and S. D. Kim. Design and Evaluation of a Selective Compressed Memory System. In Proc. *The 1999 IEEE International Conference on Computer Design,* 1999.

[12] C. H. Lin, W. Wolf, and Y. Xie. LZW-based code compression for embedded systems. In Proc. *Conf. on Design, Automation and Test in Europe,* 2004.

[13] LZO Algorithm. http://gnuwin32.sourceforge.net/packages/lzo.htm

[14] The Omega Project. http://www.cs.umd.edu/projects/omega/

[15] M. Ros, and P. Sutton Code Compression Based on Operand-Factorization for VLIW Processors. In Proc. *The Conference on Data Compression,* 2004.

[16] B. P. Tunstall. Synthesis of Noiseless Compression Codes. *PhD Thesis, Georgia Institute of Technology*, Sept. 1967.

[17] A. Wolfe, and A. Chanin. Executing compressed programs on an embedded RISC architecture. In Proc. *The 25th International Symposium on Microarchitecture,* 1992.

[18] Y. Xie, W. Wolfe, and H. Lekatsas. Profile-Driven Selective Code Compression. In Proc. *DATE,* 2003.

[19] Y. Xie, W. Wolfe, and H. Lekatsas. Code compression for VLIW using variable-to-fixed coding. In Proc. *The 15th International Symposium on System Synthesis,* 2002.

[20] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, CA, 1996.