Tiling based Compression for Reducing Memory Occupancy of Embedded Applications

Dhanashree Kulkarni

Abstract—This paper aims to increase the memory system utilization of embedded applications. An approach is presented that inserts compression and decompression calls in the application code to reduce maximum and average memory space consumption. Memory space occupancy indicates the memory space occupied by application data at each point during the course of execution. There are two metrics associated with memory space occupancy namely Maximum memory occupancy (MMO) and Average memory occupancy (AMO). The goal behind the approach presented in this paper is to reduce both MMO and AMO for array/loop intensive embedded applications.

Keywords—tiling, data compression, Maximum memory occupancy, average memory occupancy

I. INTRODUCTION

Many parallel embedded systems execute multiple applications simultaneously, which puts additional pressure on the on-chip memory capacity. For example, much commercial architecture 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.

The experimental results show that the proposed approach reduces both maximum and average memory occupancies significantly. In addition, it shows that it can save more memory space than different alternative techniques present today.

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].

III. DATA COMPRESSION 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.

1) Data Tiling and Memory Compression: This scheme compresses only arrays that can benefit from data compression. These arrays are referred to as compressible arrays.

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 in the compressed area. The memory in this area is divided into equal-sized slices. As in the case of the decompression buffer, the compressed area also has a free table keeping all free slices. Fig.1 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 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.

![Fig 1. Architecture supporting memory compression](image)

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. Consider an example. Suppose original code of a loop nest accesses a 600 × 600 array X. We apply data tiling to array X such that the size of each tile is 100 × 100. 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 decompressor 100 times for each tile. Hence, the decompressor is invoked 100 × 36 = 3600 times. By applying loop tiling to the loop nest we can obtain a tiled loop nest by introducing, loop iterators i and j as intertile iterators, and the loop nest formed by them is referred to as the intertile loop nest. Similarly, we introduce iterators ii and jj as the intratile iterators, and the loop nest formed by them is referred to as the intratile loop nest. By using tiled loop nest, the decompressor is invoked only 36 times, once for each i, j combination.

B. Experiments

1) Platform: The compression-based approach has been simulated using MATLAB. As the compression/decompression method, the Lempel–Ziv–Welch (LZW) algorithm is used. LZW 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. The compression based tiling approach is designed such that it can work with any compression/decompression algorithm and is not tied to a particular one in any way.

To test the effectiveness of the approach in reducing memory-space occupancy, it is applied to some array-based benchmarks. For each benchmark code experiments are performed with five different versions, which can be summarized as follows.

**BASE:**

The base execution does not employ any data compression or decompression, but only uses iteration space tiling. The memory saving results presented are all normalized with respect to this base version.

**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).

**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.

**CD or CDT:**
This is the proposed compression–directed tiling scheme. It uses compression and decompression based on the data-tiling information.

**CD+LF:**
This is a scheme that combines the compression-based approach with dead block recycling. In principle, this version should generate the best memory occupancy savings.

The three different benchmarks used for experimentation are Jacobi benchmark, LU decomposition benchmark and JPEG encoding benchmark. In numerical linear algebra, the Jacobi method is an algorithm for determining the solutions of a system of linear equations with largest absolute values in each row and column dominated by the diagonal element. The process is to solve for each diagonal element, and plug in an approximate value. Iteration is done until it converges. In linear algebra, the LU decomposition is a matrix decomposition which writes a matrix as the product of a lower triangular matrix and an upper triangular matrix. The product sometimes includes a permutation matrix. The decomposition is used in numerical analysis mainly to solve systems of linear equations or to calculate the determinant.

The name "JPEG" stands for Joint Photographic Experts Group, based on the name of the committee that created the JPEG standard. The JPEG discovered that besides removing the most of the variations in luminance, most of the slight changes in color (from pixel to pixel) can be removed and still end up with a very good representation of the image. JPEG uses Discrete Cosine Transform (DCT). The first step in JPEG encoding is to break the matrix into 8 × 8 blocks. Once this has happened the 8 × 8 'mini-matrix' blocks are put through the DCT, where the lower frequencies, or gradual changes in luminosity are pushed toward the upper-left of the 8 × 8 matrix. The next step is Quantization and reordering. The Process of quantization occurs when we divide every number in the 8 × 8 DCT matrix by a corresponding number in the Luminance (light intensity) Quantization table. Once 8 × 8 has been divided, each element is rounded to the nearest integer, this step turns all those elements between to zero, which makes the 8 × 8 matrix mostly sparse. Once the image has been quantized there are many runs of zeros throughout the matrix so, the 8 × 8 blocks are reordered as single 64-element columns and in a way that gives long runs of zeros.

2) Results:
First, as expected, the memory occupancy trend of the BASE version continuously increases. In comparison, the CD version has much better memory occupancy behavior. 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. From the behavior of the compression directed tiling approach (CD), it is seen 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, it is seen that the best space savings are achieved with the CD+LF version since it combines the advantages of both data compression and dead block recycling.

To calculate the memory occupancy different inputs are given for the same function and memory occupancy is calculated for each input. Then an average of the memory occupancy is calculated for all the 5 version and it is seen that the best average memory occupancy is shown by CD+LF version. The benchmarks that have been selected are very different from each other and hence it can be said that the proposed approach can work in different fields and different aspects. We can make a comparison of the results obtained from all the 3 benchmarks and then draw a conclusion regarding the success of the proposed compression based tiling approach. The memory saving obtained by using the compression based tiling approach are compared with respect to the base version.

The results of the experimentation clearly show that the compression approach is able to reduce memory-space occupancy significantly But memory consumption is only one
part of the whole picture. There are some other factors that have to be considered. The two important factors are impact on execution cycles i.e. performance and energy consumption. The performance and energy overheads incurred by this approach depend largely on the underlying execution model.

Consider the impact on energy consumption. Assume that a 1-MB SRAM memory is available, and the on-chip memory space is divided into banks, each being 64 KB. Since data compression reduces the total memory demand, the optimized programs operate with fewer banks on average, which, in turn, reduces the energy consumption. So in a way, this approach will save energy over the BASE case. The use of this approach shows some performance degradation but is very less and can be neglected in cases where memory has a greater priority.

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. Further, in case of Multiprocessor systems, preserving 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 in time. Thus the compression directed approach can be used in multiprocessor environments which will not only reduce the memory consumption significantly but will also improve performance and provide a savings in energy consumption.

IV. CONCLUSION AND FUTURE WORK

The memory capacity of embedded systems is continuously increasing and the increase in the application complexity and data-set sizes are far greater. To reduce memory consumption an approach to Data compression is presented that inserts compression and decompression calls in the application code based on Tiling information. The proposed approach reduces memory consumption (both Maximum Memory Occupancy and Average Memory Occupancy) significantly for embedded applications. As future work, how data compression can be used in MPSoC-based architectures to reduce the number of off-chip accesses is of great interest. 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. Also frequent off-chip memory accesses can increase overall power consumption dramatically. An on-chip memory management scheme for MPSoCs based on data compression can be considered for future work. A critical issue is to schedule compressions and decompressions intelligently so that they do not conflict with ongoing execution of application. Also, one needs to decide which processors should participate in the compression and decompression activities at any given point during the course of execution.

REFERENCES