Increasing On-Chip Memory Space Utilization for Embedded Chip Multiprocessors through Data Compression

Ozcan Ozturk, Mahmut Kandemir, and Mary Jane Irwin
The Pennsylvania State University
University Park, PA, USA
ozturk@cse.psu.edu, kandemir@cse.psu.edu, mji@cse.psu.edu

ABSTRACT
Minimizing the number of off-chip memory references is very important in chip multiprocessors from both the performance and power perspectives. To achieve this, the distance between successive reuses of the same data block must be reduced. However, this may not be possible in many cases due to data dependences between computations assigned to different processors. This paper focuses on software-managed on-chip memory space utilization for embedded chip multiprocessors and proposes a compression-based approach to reduce the memory space occupied by data blocks with large inter-processor reuse distances. The proposed approach has two major components: a compiler and an ILP (integer linear programming) solver. The compiler’s job is to analyze the application code and extract information on data access patterns. This access pattern information is then passed to our ILP solver, which determines the data blocks to compress/decompress and the times (the program points) at which to compress/decompress them. We tested the effectiveness of this ILP based approach using access patterns extracted by our compiler from application codes. Our experimental results reveal that the proposed approach is very effective in reducing power consumption. Moreover, it leads to a lower energy consumption than an alternate scheme evaluated in our experiments for all the test cases studied.

Categories and Subject Descriptors
D.3.4 [Programming Languages]: Processors—compilers, memory management, optimization

General Terms
Experimentation, Management, Performance

Keywords
Chip multiprocessors, optimizing compiler, data compression

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

1. INTRODUCTION
Continuously-scaling process technology allows millions of transistors to be packed onto a single die. In practice, this means a single chip represents an increasingly powerful computing platform. One promising instantiation of this powerful platform is a chip multiprocessor where multiple processors are connected by an on-chip communication medium to an on-chip memory space. Past research [6, 11] has discussed the potential benefits of chip multiprocessors over complex single processor based architectures.

One of the important consequences of putting multiple processors on the same die is the reduction in the cost of inter-processor communication (as compared to the case where each processor resides on a separate chip). This is true from both performance and power perspectives. However, this architecture also magnifies the cost of off-chip memory accesses. This is because the main memory accesses in this architecture need to go through the costly off-chip interconnect to access large off-chip memory components. Therefore, it is critical to reduce the number of off-chip memory accesses as much as possible, even if this causes an increase in the volume of on-chip communication activities among the chip multiprocessors. In practice, this requires reusing on-chip data as much as possible. In other words, in the ideal case, when a data block is brought from the off-chip memory to the on-chip memory space, we want all accesses to that data block (potentially coming from different processors) to be made before it is returned to the off-chip memory.

Unfortunately, optimizing data locality for each processor independently of the other processors in the chip multiprocessor may not achieve this desired high reuse of shared data blocks. This is because many previously-proposed approaches to data locality optimization do not really care how shared data is accessed. As a result, the inter-processor reuse distance for a given data block, i.e., the difference in cycles (or loop iterations) between two successive accesses to the same data block by different processors can be very large in practice. Let us consider a simple scenario, for illustrative purposes, where a data block is requested by two different processors. After the access to the data block by the first processor, we can keep the block in the local on-chip memory of that processor until the time it is required by the second processor. This will work fine if the inter-processor reuse distance is short, i.e., the second processor requests the data shortly after the first access completes. On the other hand, if the inter-processor reuse distance is large, we have two options. Either we can send the block to the off-chip memory, which means it will need to be brought back into the on-chip memory space when the second processor requests it. Or, we can continue to keep it in the on-chip memory space until the second request takes place. The drawback of the first option is clear: an extra off-chip access, which we desperately want to avoid. The drawback of the second scheme is that this block occupies space in the on-chip memory without any use until the execution point when the second processor requests it.
This paper proposes a data compression based approach to reduce the negative impact of the second option mentioned above, i.e., keeping the data block in the on-chip memory even if the reuse distance is large. Specifically, we propose to keep such data blocks (with large inter-processor reuse distances) in the on-chip memory space in a compressed form. When the second request comes, we decompress the data block and forward it to the requester (an on-chip transfer). The advantage of this scheme is that since the data block remains compressed between the two accesses it occupies less memory in the on-chip memory space, which makes more space available for new blocks. The drawback is that the block needs to be decompressed before the second request. However, depending on the relative cost of decompression with respect to off-chip memory transfers, this could still be less costly than sending the block to the off-chip memory after the first access and reading it back later for the second access. However, we should not compress the data block if its inter-processor reuse distance is short. Since a typical data-intensive parallel application running on an embedded chip multiprocessor can manipulate multiple data blocks at the same time (each potentially competing for the on-chip memory space), what we need is a global on-chip memory space optimization scheme.

Focusing on this problem under a software-managed memory hierarchy, this paper proposes an ILP (integer linear programming) based solution which determines the best set of data blocks to compress/decompress and the times at which to compress/decompress them. The proposed approach has two major components (as depicted in Figure 1): a compiler and an ILP solver. The compiler’s job is to analyze the application code and extract information on data access patterns. This access pattern information is then passed to our ILP solver which determines the data blocks to compress/decompress and the times (the program points) at which to compress/decompress them. After that, the application code is modified to reflect these compression/decompression decisions. We tested the effectiveness of this ILP based approach using several data access patterns extracted from applications by our compiler. Our experimental results reveal that the proposed approach is very effective in improving power consumption. Moreover, it leads to a lower energy consumption than an alternate scheme evaluated in our experiments for all the test cases studied.

The rest of this paper is organized as follows. The next section discusses related work. Section 3 explains the chip multiprocessor abstraction used. Section 4 discusses the approach used by our compiler for extracting the data block access pattern information from a given application code. Section 5 presents the details of our ILP formulation. Section 6 shows an example application of our approach. Section 7 presents and discusses the results from our experimental analysis. Finally, we conclude the paper in Section 8.

3. ARCHITECTURE ABSTRACTION

The architectural abstraction our approach works with is given in Figure 2. In this chip multiprocessor architecture, each processor has a local on-chip memory and the local memories of all processors collectively define the local memory space of the chip multiprocessor. There is also an off-chip memory space which is assumed to be much larger in capacity compared to the on-chip memory space. All memory components (on-chip and off-chip) are under software control; that is, the software is responsible for moving the data elements across the memory components (i.e., they are not structured as caches). Note that software-managed memories allow accurate worst-case execution time (WCET) analysis and is, thus, very useful in real-time embedded computing. To minimize the overheads and exploit spatial locality in our multiprocessor architecture, the data transfers between the memory components are managed at a data block granularity. In other words, when a processor issues a request for a data item, the entire data block that contains that item is brought from the off-chip memory to the on-chip memory.

From a processor’s viewpoint, in this architecture a data block can be in three possible locations: its local memory, local memory of another processor (also called remote memory in this paper), and off-chip memory. For a processor, accessing data from its local memory is less expensive than accessing it from the off-chip memory. Therefore, exploiting data locality by keeping as many data blocks as possible in the on-chip memory is very important. This paper uses data compression to increase the number of data blocks that can be kept in the on-chip memory. Figure 2 also shows potential data block transfers between the memory components.

prior proposals to the problem are based on loop transformations [7, 8, 9, 19], i.e., modifying the order of loop iterations to make data access pattern more cache friendly. Data reuse and data lifetime information has been exploited in different directions that include both code space and data space optimizations [5, 17, 18]. To address the memory space optimization problem [3, 12] proposed an application-specific memory synthesis for system-on-a-chip architectures. While such specialized architectures generate very good results from both power and performance angles, the resulting design may not be very flexible as the data access pattern of the entire application should be captured in a single memory configuration. Data compression has also been used to reduce memory footprint and energy consumption in the past [1, 2, 10, 21, 22]. In [4] data compression is used to reduce the energy consumption in DRAMs by increasing the effectiveness of low-power operating modes. Data compression has also been used in the context of scratch-pad memories (SPMs) [16]. Our goal in this work is to show that data compression can be used to reduce the number of off-chip memory accesses in chip multiprocessors.

2. RELATED WORK

An important advantage of chip multiprocessors is that it reduces the cost of interprocessor data communication from both performance and power perspectives as this communication does not need to go over off-chip buses. The prior work [6, 11, 13, 15] discusses several advantages of these architectures over complex single-processor based designs. A potential drawback is the increased cost of off-chip accesses as compared to interprocessor communication. In particular, multiple processors can try to use the same set of buses/pins to go off-chip, and this can put a high pressure on memory subsystem. Consequently, cutting the number of off-chip memory accesses becomes extremely important. Prior research exploit data locality as a potential solution. Most of the
4. COMPIlER ANALYSIS FOR DETECTING DATA BLOCK ACCESS PATTERN

Our ILP solver, described in Section 5, determines optimum data transfers and data compressions/decompressions and takes as input the data block access pattern. This access pattern is basically a trace that gives the order at which the different data blocks are requested by each processor. There are two primary ways of extracting this access pattern information. The first option is profiling, i.e., executing/simulating the application code with some representative input and recording the data block access pattern. While this may be viable in some cases, this is not a very reliable approach in general as it can be input dependent. The second option, which is the one adopted in this work, uses static compiler analysis to extract this information. The main advantage of this approach is that it does not rely on any particular input. Its drawback, on the other hand, is the fact that it needs to be conservative in its analysis of the application code.

Our compiler analysis for determining data block access patterns works with a given data block size which we assume, for simplicity of implementation, is the same for all arrays in the application code. The ground input to the analysis is a parallelized application code. The exact mechanism used in parallelizing the code is orthogonal to the focus of this paper. What we mean by parallelization in this context is distributing loop iterations across the processors; each processor is typically assigned a subset of loop iterations. Note that, each loop is parallelized independently, and this approach exploits data parallelism (i.e., each processor executes the same code, at a given time, using different data). The compiler analyzes the parallelized input code and determines, for each processor, the order of accesses for data blocks (based on the given data block size). As an example, let us consider the code fragment shown in Figure 3(a). This fragment contains two separate loop nests executed one after another. Assuming the data block partitioning shown in Figure 3(b), the data block access pattern for this code fragment is 60, 61, 62, 63, 64, 65, 66, 67, assuming a single processor. This is because first the data elements in data block 60 are accessed, and after they are all accessed, the elements in data block 61 are accessed, and so on. However, if the block partitioning is as shown in Figure 3(c), the access pattern would be 60, 61, 62, 63, (64, 65, 66, 67)N, where sN means N repetitions of pattern s. Note that, since it is difficult and costly to keep track of the block accesses in terms of individual loop iteration granularity, we keep track of them in terms of execution steps, or simply steps. A step in this work corresponds to a set of loop iterations that are executed successively one after another. For example, if only half of a data block could be accessed by a certain step, then the block access pattern for the first loop iteration in Figure 3(b) would be 60, 61, 62, 63, 64, 65, 66, 67, 68. While we also employ several approaches to condense a given access pattern, the detailed discussion of these techniques is outside the scope of this paper. Now, let us suppose that the first loop nest in Figure 3(a) is parallelized across two processors by distributing the iterations of loop k1 across the processors; i.e., the first processor is assigned iterations 0...N/2 − 1, and the second processor is assigned iterations N/2...N − 1. Then, the block access pattern for the first processor (under the block partitioning Figure 3(b)) in this nest is 60, 61, and that for the second nest is 62, 63, assuming that in a step we can access all the elements in a data block. In this paper, the block partitions used are dictated by the parallelization strategy used for the most expensive (in terms of energy consumption) loop nest in the application code and the number of processors. The step sizes, on the other hand, are selected such that, as we move from one step to another, the set of data blocks accessed changes significantly.

5. ILP FORMULATION

ILP provides a set of techniques that solve those optimization problems in which both the objective function and constraints are linear functions and the solution variables are restricted to be integers. The 0-1 ILP is an ILP problem in which each (solution) variable is restricted to be either 0 or 1 [14]. It is used in this paper to determine the data blocks to compress/decompress and the times at which to compress/decompress them. Table 1 gives the constant terms used in our ILP formulation. Note that the entries of the access matrix, Ap,s, are filled by the compiler analysis explained in Section 4. We used Xpress MP [20], a commercial tool, to formulate and solve our ILP problem.

Our objective is to find the location of each data block and its form (i.e., compressed versus uncompressed) for each step for minimum energy consumption. Based on a given chip multiprocessor architecture, we determine the location of each data block using the data access pattern. We define 0-1 variables for each data block and for every step. By using these 0-1 variables, we determine in which memory component the data block should reside. Also, compression/decompression activities can be captured by these variables. Although, we assume that the on-chip memory sizes of all processors are equal (i.e., all of them have the same local memory size), this can easily be relaxed, if desired, to model a heterogeneous system.

Table 2 lists the 0-1 variables used in our ILP formulation. Assuming that P denotes the number of processors in the architecture, S the number of steps, B the number of data blocks, we can formulate our problem as follows. A data block can be in three different states: compressed on-chip (Cfb,p,s), uncompressed on-chip (Dfb,p,s) and off-chip (Ofb,s). Note that, we do not allow a data block to reside in the compressed form within the off-chip memory since we assume the off-chip memory is sufficiently large to keep all the data blocks. Since a data block must be in one of these three states, the following constraint must hold:

\[ \text{Osb} = \sum_{i=1}^{P} (\text{Cfb},i,s + \text{Dfb},i,s) = 1, \quad \forall\text{b},s. \]  

(1)

In the above expression, i iterates over the processors in the sys-

![Image 317x466 to 568x589]

Figure 3: (a) An example code fragment consisting of two loop nests. (b-c) Two alternate data block partitions.
term. An access to a data block can only be performed if the data block in question is not compressed. This is captured by the following constraint:

$$\sum_{i=1}^{P} CF_{b,i,s} = 0, \forall b, s \text{ such that } \exists p A_{p,s} = b. \quad (2)$$

The available on-chip memory space, equally divided among the processors, is limited. Hence, the data blocks stored in each of these on-chip memory components have to be in accordance with the available space. We can express this constraint as follows:

$$\sum_{i=1}^{P} \left( CF_{i,p,s} / \text{CompRatio} + DF_{i,p,s} \right) \times DS \leq MS, \forall p, s. \quad (3)$$

Note that, CompRatio is the compression ratio for the compression algorithm being used. $MS$ is the on-chip memory space allocated for each processor. As mentioned earlier, although the available on-chip memory space is divided among the processors equally, it is possible to modify our formulation to reflect a non-uniform distribution of the available memory space, and solve the problem for such a heterogeneous system.

If a data block is moved from the on-chip memory space to the off-chip memory space or vice-versa, some amount of energy is expended. In order to capture these movements, we use the MoveOff binary variable. This variable indicates whether data block $b$ is moved from on-chip/off-chip memory to off-chip/on-chip memory at step $s$. Specifically, we have:

$$\begin{align*}
\text{MoveOn}_{b,s} & \geq \text{Off}_{b,s} - \text{Off}_{b,s-1} \\
\text{MoveOff}_{b,s} & \geq \text{Off}_{b,s-1} - \text{Off}_{b,s} \\
& \forall b, s \text{ such that } s \geq 2. \quad (4)
\end{align*}$$

The first expression above indicates whether data block in question ($b$) is moved from on-chip memory to off-chip memory at step $s$. On the other hand, the second expression will hold true only if the data block is moved from off-chip memory to on-chip memory at step $s$. It is important to note that, MoveOff$_{b,s}$ cannot have a negative value since it is a 0-1 variable.

The data block movements within the on-chip memory space are also allowed (see Figure 2), and needs to be captured as well. We use $\text{MoveOn}_{b,s}$ to serve for this purpose. A data block is moved from the local memory of $p_1$ to the local memory of $p_2$ if the following expression holds:

$$\begin{align*}
\text{MoveOn}_{b,s} & \geq (CF_{b,p_2,s} + DF_{b,p_2,s}) \\
& \quad \left. + (CF_{b,p_1,s-1} + DF_{b,p_1,s-1}) - 1, \right. \\
& \forall b, p_1, p_2, s \text{ such that } s \geq 2, \ p_1 \neq p_2. \quad (5)
\end{align*}$$

Compressing a data block would incur some energy penalty\(^1\) which can be captured by using the 0-1 variables $CF_{b,p,s}$ and $DF_{b,p,s}$ for each data block accessed. A data block is compressed if it is in the uncompressed form at a certain step ($s-1$) and in the compressed form during the next step ($s$). The following expression captures this constraint:

$$\begin{align*}
\text{Compress}_{b,s} & \geq CF_{b,p_1,s} + DF_{b,p_2,s-1} - 1, \\
& \forall b, p_1, p_2, s \text{ such that } s \geq 2. \quad (6)
\end{align*}$$

Notice that, we are assuming here a data block is not moved from on-chip memory to off-chip memory if it is in the compressed form. Even though it is possible to drop this assumption in our formulation, it does not seem to be meaningful to keep a data block in the compressed form in the on-chip memory if it is not going to be used and will be kicked out from the on-chip memory space. Similarly, decompressing a data block would also incur some energy penalty,\(^1\) which can be captured as follows:

$$\begin{align*}
\text{DeCompress}_{b,s} \geq & \ DF_{b,p_1,s} + CF_{b,p_2,s-1} - 1, \\
& \forall b, p_1, p_2, s \text{ such that } s \geq 2. \quad (7)
\end{align*}$$

Having specified the necessary constraints to be satisfied, we next give our objective function. Recall that our goal is to minimize memory energy consumption. In our execution model, there are three components of the total memory energy consumption: data access energy, data transfer (movement) energy, and data compression/decompression energy. In the following paragraphs, we explain each of these three components in detail.

- **Data Access Energy** is the energy consumed when a data block is accessed. It is composed of the energies spent in local accesses (on-chip), remote accesses (on-chip), and off-chip accesses. The local access energy can be calculated by considering the accesses made by a processor to its local on-chip memory space ($A_{p,s}$). We express this cost as follows:

$$E_{A-local} = \sum_{s=1}^{P} \sum_{p=1}^{B} (DF_{p,s,p} \times C_{onchip}). \quad (8)$$

Similarly, the non-local (remote) on-chip access energy cost can be captured by using the following expression:

$$E_{A-remote} = \sum_{b=1}^{B} \sum_{s=1}^{P} \sum_{p=1}^{B} (DF_{b,p_1,s} \times C_{remote}), \quad (9)$$

Finally, the energy consumed due to off-chip data accesses can be expressed as:

$$E_{A-offchip} = \sum_{s=1}^{S} \sum_{b=1}^{B} (Off_{b,s} \times C_{offchip}), \quad (10)$$

As a result, the overall data access energy can be calculated using the following expression:

$$E_A = E_{A-local} + E_{A-remote} + E_{A-offchip}. \quad (11)$$

- **Data Transfer Energy** is the energy consumed when a data block is transferred (moved) from one memory component to another. The data blocks can be transferred within the on-chip memory space, i.e., across the local memories of the processors, and they can also be transferred between the on-chip and off-chip memories (see Figure 2). Consequently, the total data block transfer cost can be obtained by adding up these two components:

$$E_T = \sum_{b=1}^{B} \sum_{s=1}^{S} (MoveOn_{b,s} \times C_{moveon} + MoveOff_{b,s} \times C_{moveoff}). \quad (12)$$

- **Compression/Decompression Energy** is the energy consumed

\(^1\)This energy does not include the energy of accessing the data block in memory (on-chip local and on-chip remote). It is solely due to the compression/decompression activity. The memory access energies are captured by the constraints defined earlier.
Figure 4: The data block placement generated with compression and without compression. Note that both the schemes generate optimal results; the difference is that one of them does not use compression.

when a data block is compressed/decompressed. We use the following expression to specify this energy cost:

\[ E_{\text{C}} = \sum_{b=1}^{B} \sum_{s=1}^{S} (C_{\text{comp}}b,s \times C_{\text{comp}} + D_{\text{decomp}}b,s \times C_{\text{decomp}}). \]  

Based on these individual energy costs, we can now express the total memory energy consumption \( E \) as follows:

\[ E = E_A + E_T + E_C. \]  

In this formulation, \( E_A, E_T, \) and \( E_C \) correspond to access energy, transfer energy, and compression energy, respectively. Based on the discussion above, our 0-1 ILP problem can be defined as one of “minimizing \( E \) under constraints (1) through (13).”

6. EXAMPLE

In this section, we give an example showing the effectiveness of our approach. An example data access pattern for a case with 4 processors, 6 steps and 12 data blocks is given in Table 3. We assume, for the sake of explanation, that each processor has an on-chip local memory of size 2K and the data blocks are of size 1K. We further assume, for simplicity, that the compression ratio is 2 (i.e., when a data block is compressed it is half of the original size).

In Figure 4 the data block placements generated by our approach as well as those generated by an alternate approach that does not use compression are shown. Our compression-based approach is shown on the left, whereas the optimum placement for the no-compression case is shown on the right. Note that, in no-compression scheme, it is possible to transfer the data blocks within the on-chip memory and between the on-chip memory and the off-chip memory between execution steps. For this example, we see that the optimum energy savings with the no-compression scheme is obtained without any data block transfers across the steps. The most important point in this example is that, in our approach, all of the data blocks are kept within the on-chip memory by using compression. On the other hand, 8 data blocks are accessed from the off-chip memory if compression is not used. As it can be seen from the figure, data blocks 1, 2, 3 and 4 are kept compressed until they are needed by other processors. Hence, they do not require any off-chip accesses. The number of total operations needed for both compression and no-compression cases are given in Table 4. Even if we conservatively assume that off-chip access is 100 times costlier than an on-chip local access (in terms of energy), compression/decompression is 25 times costlier than an on-chip local access and on-chip remote access is 5 times costlier than an on-chip local access, our approach reduces the energy consumption by 22%.

7. EXPERIMENTAL EVALUATION

We test the effectiveness of our ILP-based approach in reducing energy consumption and execution cycles of loop-intensive applications with eight benchmark codes.

7.1 Setup

Table 5 lists the benchmark codes used in this study and their important characteristics (taken from the Spec and Perfect Club benchmark suites). The second column of this table gives the total size of the data processed by the benchmark and the third column gives the data block size. The last column shows the number of steps used for each benchmark. The default simulation parameters used in our experiments are given in Table 6. Note that the energy latency values are given as multiples of the on-chip (local) access energy latency value. For example, remote on-chip access (i.e., an access made by a processor to the local, on-chip memory of another processor) is 5 times as costly as an on-chip access to local memory, and that an off-chip access is 150 times as costly as an on-chip access. We performed experiments with two different versions of each benchmark code in our suite:

- **No-Compression**: This is the classical on-chip memory management strategy that does not use any compression or decompression. The idea is to keep the hot data blocks in the on-chip memory space as much as possible. In our implementation, we use ILP to decide the optimal location of data blocks during the course of execution. Note that this scheme achieves optimal memory space utilization and data block movements as long as we do not consider compression.

- **Compression**: This is the integer linear programming based strategy discussed in this paper, wherein a data block is compressed/decompressed if doing so is more energy efficient. As mentioned earlier, the rationale behind this scheme is to keep as many data blocks as possible in the on-chip memory space, thereby cutting the number of off-chip memory accesses.

In the experimental results to be presented shortly, the term “execution cycles” denotes the total number of execution cycles taken by the application execution, which includes the cycles spent during compression and decompression as well. Also, the term “energy consumption” refers to the energy expended in the on-chip and off-chip memories (during data transfers and data accesses) as

<table>
<thead>
<tr>
<th>Scheme</th>
<th>On-Chip Local</th>
<th>On-Chip Remote</th>
<th>Off-Chip</th>
<th>Comp.</th>
<th>Decomp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compression</td>
<td>20</td>
<td>4</td>
<td>0</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>No-Compression</td>
<td>16</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4: The number of different types of activities for our example.
9. CONCLUSIONS

Recent research indicates that packing multiple processors on the same die is an effective way of utilizing ever-increasing number of transistors. Placing multiple processors into a single die reduces communication costs between the processors that are incurred in conventional high-performance parallel architectures. However, on the negative side, this tighter integration exerts an even higher pressure on off-chip accesses to the memory system, which eventually makes effective utilization of on-chip memory space very critical. This paper focuses on-chip memory utilization problem for an embedded chip multiprocessor and proposes a solution that makes use of both a compiler and an ILP solver. This approach uses compression for data blocks whose inter-processor reuse distances are large. We evaluated this approach using real access patterns extracted by the compiler from embedded applications. Our experiments show that this ILP based approach is useful from both performance and power angles, and it outperforms an alternate scheme attempting the same objective without employing any data compression.

Table 6: The default simulation parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Processors</td>
<td>4</td>
</tr>
<tr>
<td>Compression Ratio</td>
<td>2</td>
</tr>
<tr>
<td>On-Chip Local Access Energy</td>
<td>1</td>
</tr>
<tr>
<td>On-Chip Remote Access Energy</td>
<td>5</td>
</tr>
<tr>
<td>Off-Chip Access Energy</td>
<td>150</td>
</tr>
<tr>
<td>Compression Energy</td>
<td>25</td>
</tr>
<tr>
<td>Decompression Energy</td>
<td>25</td>
</tr>
<tr>
<td>On-Chip Local Access Latency</td>
<td>1</td>
</tr>
<tr>
<td>On-Chip Remote Access Latency</td>
<td>5</td>
</tr>
<tr>
<td>Off-Chip Access Latency</td>
<td>150</td>
</tr>
<tr>
<td>Compression Latency</td>
<td>25</td>
</tr>
<tr>
<td>Decompression Latency</td>
<td>25</td>
</tr>
</tbody>
</table>

Figure 5: (a) Percentage energy reductions. (b) Percentage execution cycle reductions.

8. CONCLUSIONS

well as the energy consumed during compressions and decompressions.

7.2 Results

Figure 5 gives the results. The bar charts show the energy/performance improvements brought by our approach over the No-Compression scheme. Note that all the numbers are in terms of the values given in Table 6. One can make several observations from these results. First, we see that the average improvements in energy consumption and performance are 43.3% and 33.3%, respectively, showing the effectiveness of our approach. Second our approach generates better results than the No-Compression case for all the benchmarks tested.

9. REFERENCES