Locality-Conscious Workload Assignment for Array-Based Computations in MPSOC Architectures*

Feihui Li and Mahmut Kandemir
Computer Science and Engineering Department
The Pennsylvania State University
University Park, PA 16802, USA
{feli, kandemir}@cse.psu.edu

ABSTRACT

While the past research discussed several advantages of multiprocessor-system-on-a-chip (MPSOC) architectures from both area utilization and design verification perspectives over complex single core based systems, compilation issues for these architectures have relatively received less attention. Programming MPSOCs can be challenging as several potentially conflicting issues such as data locality, parallelism and load balance across processors should be considered simultaneously. Most of the compilation techniques discussed in the literature for parallel architectures (not necessarily for MPSOCs) are loop based, i.e., they consider each loop nest in isolation. However, one key problem associated with such loop based techniques is that they fail to capture the interactions between the different loop nests in the application. This paper takes a more global approach to the problem and proposes a compiler-driven data locality optimization strategy in the context of embedded MPSOCs. An important characteristic of the proposed approach is that, in deciding the workloads of the processors (i.e., in parallelizing the application) it considers all the loop nests in the application simultaneously. Our experimental evaluation with eight embedded applications shows that the global scheme brings significant power/performance benefits over the conventional loop based scheme.

Categories and Subject Descriptors

D.3.m [**Software**]: Programming Languages—*Miscellaneous*

General Terms

Performance

Keywords

Data Locality, MPSoC

1. INTRODUCTION

MPSOC (multiprocessor-system-on-a-chip) architectures have several key advantages over single core based complex systems as noted by prior studies such as [14, 9, 13] from both area utilization and design verification perspectives. However, MPSOC based systems are also attractive from a programming perspective since processor workload assignment in these architectures can be carried out at a source code level, which is very different from the ILP (instruction level parallelism) style of code parallelization that requires complex dependence check at runtime and costly instruction issue logic. An important problem though is to come up with a suitable workload assignment to each core. This is because there are several issues that need to be accounted for, if one is to find an acceptable workload assignment for a given application:

- *Locality:* Each core should access the same set of data most of the time; this helps to minimize the frequency and volume of interprocessor data communication, and reduce the number of off-chip memory accesses.
- *Parallelism:* The amount of work done between successive synchronization points should be as large as possible; this helps to amortize the cost of interprocessor synchronization, an important concern for embedded MPSOCs.
- *Load Balance:* The amount of work assigned to different cores should be more or less the same; this helps to increase the utilization of cores and reduce overall idle time as well as execution time.

One approach that could be used for addressing these issues is loop-based code parallelization. The idea behind this approach is to consider loop nests of the application one by one and parallelize each loop nest at the coarsest granularity possible (i.e., parallelize the outermost parallelizable loop in each nest). In fact, several proposals (e.g., [1]) originally developed in the context of high-performance code parallelization can be used for this purpose in the MPSOC context. However, one key problem associated with loop based techniques is that they fail to capture the interactions between the different loop nests in the application. Since the parallelization decisions are taken for each loop nest independently of the others, resulting data locality may not be optimal when the entire application is considered. For example, a given processor core can access different data elements in consecutive loop nests (as a result of loop-based parallelization), even though these nests share the same set of arrays. The next section gives an example that illustrates this problem. It needs to be emphasized that failing to ensure the best data locality is of a more serious concern in MPSOCs as compared to high-performance parallel architectures. This is because poor data locality can lead to frequent off-chip references, which may not be tolerable from performance and power perspectives (as an off-chip access is much costlier than an on-chip memory access or interprocessor communication, which is also on-chip). Therefore, every possible effort should be made to minimize the number of off-chip references, by maximizing data reuse.

This paper takes a global approach to the problem and proposes a compiler-driven data locality optimization strategy in the context of embedded MPSOCs. An important characteristic of the proposed approach is that, in deciding the workloads of the processors, it considers all the loop nests in the application simultaneously. It achieves this by starting the optimization process with a data mapping (i.e., logically partitioning data across processors)

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

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

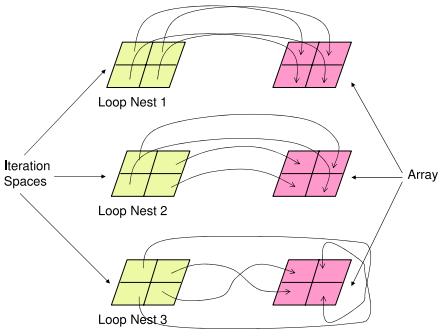

Figure 1: An example data access pattern scenario that involves four processors. In this scenario, three different loop nests access the same array. The arrows show which portion of the array is accessed by each processor core.

and by deriving iteration mapping (workloads) from this data mapping. Our results indicate that this global approach brings significant improvements (in both execution cycles and power consumption) over a conventional loop based code parallelization strategy.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 summarizes loop based code parallelization and points out its problems. Section 4 presents our global approach to code parallelization in embedded MPSOCs. Section 5 presents an experimental evaluation of the global strategy and compares it to the loop based strategy. Section 6 gives our concluding remarks.

2. RELATED WORK

We discuss the related work in two categories: single processor based data locality optimizations and MPSOC related efforts. In the single processor domain, several strategies were proposed to improve cache performance, including prefetching [12], data copying between different portions of the address space [18], and locality optimizations for array-based codes [19, 5]. Dynamic techniques such as access re-ordering on-the-fly between the processor and memory (using a specialized hardware unit) have also been used [11]. The IMEC group [4] investigated the problem of constructing customized memory hierarchies for low power, and pioneered the work on applying loop transformations to minimize power dissipation in data dominated embedded applications. Memory optimizations for embedded systems were addressed, among others, by Shiue and Chakrabarti [17]. Kodukula [8] proposed a data space oriented code restructuring scheme for cache based single processor systems. The work described in this paper is different from these prior efforts as we target an MPSOC architecture.

In the MPSOC domain, most of the efforts focused on architecture design and circuit related issues [14, 9]. [6] focused on the problem of determining the optimal number of processors for each loop nest in a given application. and [10] argued that a processor can be put into sleep mode when it reaches the barrier early. Another study [16] pointed out that a chip multiprocessor can be an energy-efficient alternative for exploiting future billion transistor designs, and also mentioned that voltage scaling can further complement this architecture. They showed around 9% to 15% power savings in multimedia applications that use independent threads. Our work is different from these studies as well in that we achieve energy savings via data space oriented application parallelization.

3. LOOP BASED CODE PARALLELIZATION

The loop based code parallelization focuses on a single loop nest at a time, and parallelizes it using the data dependence information extracted by the compiler. While several proposals exist in the literature, the main goal behind all these techniques is to rewrite a given loop nest in a form that allows parallel execution of independent loop iterations. To minimize synchronization costs, it is also important that we obtain coarse grain parallelism, as opposed to fine grain parallelism. In terms of parallel execution, this means parallelizing the outermost (parallelizable) loop as much as possible.

Each execution of a nest body can be represented by an iteration vector, each entry of which corresponds to a loop, starting from the top. When there is no confusion, we use the terms "loop iteration" and "iteration vector" interchangeably. Note that, an iteration vector represents the executions of all the statements in the loop body (under the specified values of the iterators in the vector).

If a loop iteration $\vec{q_2}$ depends on an iteration $\vec{q_1}$ (where $\vec{q_2} > \vec{q_1}$, the difference between them, $\vec{q_2} - \vec{q_1}$ is called the data dependence vector [3].[1] We are mostly interested in cases where all the entries of a dependence vector are constants, in which case it is also referred to as the distance vector [3]. The distance vectors extracted from a loop nest collectively define a distance matrix, whose rows are made of distance vectors. Let us focus on an arbitrary distance vector $\vec{d}$ extracted from a nest with n loops:

$$\vec{d} = (d_1 \quad d_2 \quad d_3 \quad \cdots \quad d_{n-1} \quad d_n)^T.$$

Considering (only) this distance vector, the kth loop can be parallelized if at least one of the two conditions below are satisfied [3]:

- $d_k = 0$, or
- $(d_1 \quad d_2 \quad \cdots \quad d_{k-1})^T$ is lexicographically positive.[2]

In obtaining the coarsest grain parallelism, we parallelize only the kth loop such that this loop parallelizable and none of the loops from top down to the $(k-1)$th loop is parallelizable. If there are multiple distance vectors in the nest, a loop is parallelizable if and only if it is parallelizable according to all these distance vectors. The different approaches to coarse grain loop based parallelization differ mostly in their capability of extracting the highest level of parallelism in a given loop nest.

One of the serious drawbacks of loop based parallelization is that it does not capture the data sharings between the different nests, as far as data locality is concerned. This is illustrated in Figure 1, which depicts how an example application with three separate nests accesses a two-dimensional array. The left part of the figure shows the iteration spaces of the nests. Each iteration space is assumed to be divided into 4 parts as a result of parallelization over 4 processor cores. That is, each core is set to execute one fourth of the original iteration space of each nest. The array (data space) manipulated by these nests (note that all three cores manipulate the same array) is also shown as divided into four regions (on the right of the figure). The arrows from the iteration spaces to the array space indicate which array region each part of the iteration space accesses. Since the loop based parallelization does not capture the data sharings between the different loop nests, it can assign to a core from each nest the (iteration space) part in the same position. As a result, a given core can have the access pattern shown in Figure 2(a). Let us focus on the portions of the iteration spaces marked "*", which are assigned to the same processor core under the loop based scheme. The problem with this access pattern is that, at each nest, the core in question accesses a different data region of the array. Therefore, one would not expect a good data reuse (data cache performance) from this access pattern. The objective of the global parallelization scheme discussed in the next section is to address this problem. Notice that, improving data cache performance normally brings both energy reduction and performance benefits.

[1] We are mainly interested in data dependences here. This is because the application codes we focus on are loop nest intensive and they do not contain conditional flow of executions.

[2] We say that vector $\vec{d} = (d_1 d_2 \cdots d_n)$ is lexicographically less than (shown as $<$) vector $\vec{d'} = (d'_1 d'_2 \cdots d'_n)$ if there is a c such that $1 \leq c \leq n$ and $d_i = d'_i$ for all $i < c$ and $d_c < d'_c$. A vector is said to be lexicographically positive (negative) if it is greater than (less than) the zero vector.

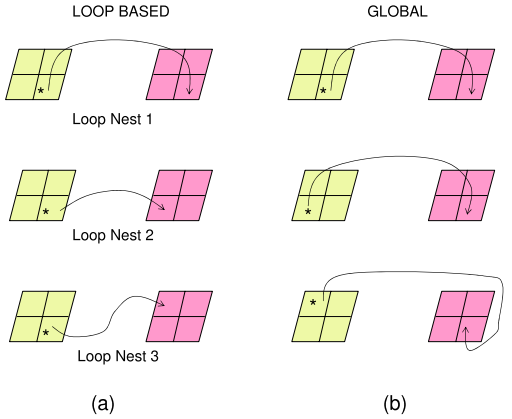

Figure 2: Loop based parallelization (a) and global parallelization (b) from the perspective of a given processor core. The iteration space portion marked using "*" indicate the part assigned to a particular processor core.

4. GLOBAL PARALLELIZATION

4.1 MPSOC Architecture

Our focus is on a shared memory based MPSOC architecture. In this architecture, we have multiple cores on the same die (typically between 4 and 16). Each core has private L1 instruction and data caches, and thus ensuring data locality is very important. While we do not evaluate in this paper, this architecture can also accommodate a shared on-chip L2 cache. We assume a bus based on-chip interconnect and cache coherence is maintained using a MESI-like consistency mechanism, the choice of which is orthogonal to our approach. We also assume the existence of a shared off-chip memory. Our objective is to minimize the number of off-chip memory references.

4.2 Overview

Figure 2(b) illustrates, through an example, our global approach to code parallelization. The figure shows the assigned parts from each loop nest to a particular processor core. The portions marked "*" indicate the iterations assigned to the same core. Note that this assignment differs from the one shown in Figure 2(a) in two ways. First, the same core is assigned to different parts in the different iteration spaces (i.e., not the corresponding parts). Second, and more importantly, the core accesses the same array region in each nest, which means that one can expect a very good data locality (L1 performance). The main goal of the global strategy is to achieve the highest data reuse possible. Clearly, this ideal scenario (as depicted in Figure 2(b)) may not be achieved in all cases, due to data dependences between the different loop iterations. However, our approach tries to exploit the maximum possible data reuse allowed by data dependences. The next subsection explains the mathematical engine behind this global code parallelization scheme.

4.3 Mathematical Details

Our approach to the workload determination problem is data space oriented, meaning that it decides the set of iterations to be assigned to each processor core considering the arrays accessed by the application. We use $\mathcal{Q}_k$ (where $1 \leq k \leq n$) to denote the set of iterations that will be executed by loop nest k. Let us focus on an array Z_j (where $1 \leq j \leq m$) manipulated by the application. We use $\mathcal{Z}_j$ to represent the set of data items in Z_j. Note that $\mathcal{Z}_j$ defines a rectilinear polyhedron. We assume that there are p processor cores in the MPSOC over which the application is to be parallelized.

In the first step of our approach, we logically divide the array into p regions and each region is assigned to a processor core. We now focus on processor s (where $1 \leq s \leq p$) and loop nest k (where $1 \leq k \leq n$). Let $\mathcal{Z}_{s,j}$ be the data elements from array Z_j that are assigned to core s (we will discuss shortly how this data assignment is actually made). We use $\mathcal{Q}_{s,j,k}$ to represent the set of loop iterations from loop nest k that touch the elements in $\mathcal{Z}_{s,j}$. The global parallelization strategy assigns the iterations in $\mathcal{Q}_{s,j,k}$ to core s. In other words, each core executes the iterations that access the array region it is assigned to. Note that this iteration assignment can be repeated for each loop nest. In other words, core s is assigned iterations $\mathcal{Q}_{s,j,1}$, $\mathcal{Q}_{s,j,2}$, ..., $\mathcal{Q}_{s,j,n}$. The common characteristic of these sets is that all the iterations in them access $\mathcal{Z}_{s,j}$. Consequently, when all the iteration assignments (for all loop nests) are complete, we have the situation shown in Figure 2(b) for the example in Figure 1.

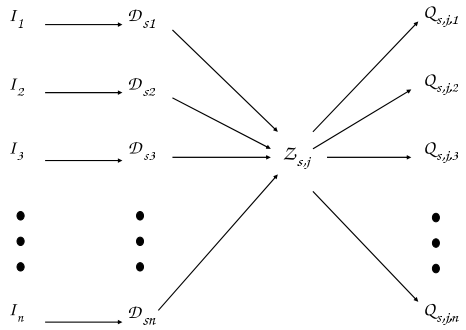

Figure 3: The process of determining the set of iterations (from all the nests) that will be executed by processor s.

At this point, there are three important issues that need to be addressed. The first issue is the problem of (logically) dividing the array elements across the processors. The second one is regarding the fact that not all the loop nests access all the elements of a given array. Consequently, we need a strategy to handle the case when one or more loop nests access only a portion of the array. The third issue is that normally an application processes multiple arrays and our iteration mapping (workload assignment) must be carried out considering all the arrays. Otherwise (i.e., if the workload assignment considers only one array), the resulting parallelized code may not be able to exploit data reuse for the arrays not considered during the workload assignment. In the following paragraphs, we elaborate on these three issues.

Dividing array elements across processors is very important as it determines the data access pattern and thus influences parallelism and cache locality. Our approach to this problem can be explained as follows. For each nest, we extract the maximum parallelism using a previously published approach in the literature [1]. This approach implements a method for deriving an optimal hyper-parallelepiped tiling of iteration space for minimal communication in multiprocessors with caches. It uses the notion of uniformly intersecting references to capture locality in array references and estimates interprocessor data communication traffic based on a data footprint concept. After parallelizing the code using the approach in [1], for each nest, our approach determines the set of data items (array elements) accessed by each core s. In determining this set of elements, we build the following set, assuming that $\mathcal{I}_s$ is the set of iterations assigned to processor core s (by the loop parallelization used) and $\mathcal{D}_s$ is the set of array elements we want to determine:

$$\mathcal{D}_s = \{\vec{d} \mid \exists \vec{I} \in \mathcal{I}_s, \quad \exists R \in \mathcal{R}_s \text{ such that } R(\vec{I}) = \vec{d}\}.$$

In this formulation, $\mathcal{R}_s$ is the set of references to the array and R represents a reference in the loop nest (i.e., a mapping from the iteration space to the data space). Since the access pattern imposed by each nest (on the array) can be different from the other nests, we next employ a unification step that comes up with a globally acceptable array partitioning (data mapping). This data mapping is then used for distributing the iterations across the processor cores. While it is possible to implement different unification schemes, the scheme used in this study is a simple one that selects the most frequently requested data mapping (when all the loop nests in the application are considered). It is to be mentioned that since this strategy is oriented toward enhancing data reuse, it may not necessarily be optimal from parallelism perspective (when each nest

is considered in isolation). This is because we try to determine a globally acceptable data mapping, which may not be the best one for each and every loop nests. However, as will be demonstrated by our experiments, the gains coming from enhanced data locality are usually far more than the losses in loop level parallelism. As an example of our array partitioning approach, let us assume that an array is accessed by three different nests. The first and the third nests require the array elements to be assigned to the processors in a row-block fashion (i.e., each processor is given a consecutive set of rows), whereas the second nest demand a column-block distribution across the processors. Let us use $\mathcal{D}_{s_1}$, $\mathcal{D}_{s_2}$ and $\mathcal{D}_{s_3}$ denote the distributions demanded by the nests, i.e., row-block, column-block, and row-block in that order, from the perspective of processor s. Considering these, our approach selects the row-block distribution ($\mathcal{D}_{s_1}$), as it is requested by a larger number of processors; i.e., we set $\mathcal{Z}_{s,j}$ to $\mathcal{D}_{s_1}$.

Figure 3 summarizes the process of determining the loop iterations that will be executed by processor s. Basically, using the approach in [1], we first determine the sets $\mathcal{D}_{s_1}$, $\mathcal{D}_{s_2}$, ..., $\mathcal{D}_{s_n}$. We next obtain $\mathcal{Z}_{s,j}$ using the strategy explained in the previous paragraph, and then determine $\mathcal{Q}_{s,j,1}$, $\mathcal{Q}_{s,j,2}$, ..., $\mathcal{Q}_{s,j,n}$, i.e., the iterations from the different nests that are assigned to processor s.

For the second issue, let us consider two loop nests, k and l, that access the same array (Z_j). We use $\mathcal{M}_{j,k}$ and $\mathcal{M}_{j,l}$ to denote the set of elements (of Z_j) accessed by nests k and l, respectively (note that $\mathcal{M}_{j,k} \subseteq \mathcal{Z}_j$ and $\mathcal{M}_{j,l} \subseteq \mathcal{Z}_j$). We can express the problem as follows: Divide (logically) the iterations in $\mathcal{Q}_k$ and $\mathcal{Q}_l$ across the p processor cores such that the parts assigned to core s from these two nests, namely $\mathcal{Q}_{s,j,k}$ and $\mathcal{Q}_{s,j,l}$, access the same set of elements as much as possible. To illustrate this pictorially, consider the sample scenario shown in Figure 4(a). This figure shows an array (say Z_j) accessed by two different nests (say k and l). The set of array elements accessed by each nest are also shown (as dark rectangular regions) within the array space. Note that neither loop accesses the entire array, and there is only a partial overlap between the sets of data elements accessed by nests k and l. Our approach in this case divides the array as shown in Figure 4(b), assuming that $p = 4$. Note that, as indicated by the two arcs shown in the figure, the two of the processors reuse their data (i.e., each of them accesses the same set of elements in both the nests), whereas the other two access different elements in different nests. Our approach determines $\mathcal{Q}_{s,j,k}$ and $\mathcal{Q}_{s,j,l}$ such that the access pattern shown in Figure 4(b) is obtained. While our approach does not achieve data reuse for all four processors in this particular case, it is still much better than randomly assigning iterations to processors (which would most probably exploit much less data reuse across the nests).

In mathematical terms, our approach proceeds as follows. We first determine the set of common elements between $\mathcal{M}_{j,k}$ and $\mathcal{M}_{j,l}$, i.e., $M_{j,common} = \mathcal{M}_{j,k} \cap \mathcal{M}_{j,l}$. Then, we assign the first $|\mathcal{M}_{j,k}|/p$ of these ($|M_{j,common}|$) elements to the first processor, the next $|\mathcal{M}_{j,k}|/p$ to the second processor, and so on. At the point where we have assigned all ($|M_{j,common}|$) elements, the remaining elements (i.e., $|\mathcal{M}_{j,k}| - |M_{j,common}|$) are assigned to the remaining processors. A similar process is repeated for the second nest (l) as well. However, in processing this nest, we are careful in assigning the same set of (common) elements to the same processor core as in the previous nest. Then, based on these data assignments, we perform the iteration assignment as explained earlier in this section.

Our approach to the third issue – multiple arrays accessed by the nests – can be explained as follows. We first identify affinity among the elements of the different arrays. Two data items are said to have affinity if they are accessed by the same loop iteration. As an example, consider the following loop nest and the three array references that appear in it:

$$\begin{array}{l} do\ t_1 = 1, M-2 \\ \quad do\ t_2 = 4, N \\ \quad\quad ...Z_1[t_1][t_2]...Z_2[t_2][t_1]...Z_3[t_1+2][t_2-3]... \end{array}$$

We note that, for a given loop iteration (a, b) in this nest, i.e., when $t_1 = a$ and $t_2 = b$, the loop accesses array elements $Z_1[a][b]$, $Z_2[b][a]$, and $Z_3[a+2][b-3]$, and thus, these three array elements have affinity. Then, instead of dividing an array into regions (as in the single array case), we divide data elements into affinity classes. Each affinity class contains data elements that exhibit affinity among them. Then, the iteration mapping (assignment) is carried out based on these affinity classes. As an example, Figure 5 depicts an example set of elements that belong to the same affinity class for the example loop nest shown above. Note that, in this figure, the regions of the different arrays marked as dark rectangles constitute the affinity class that will be assigned to a single core. Other processor cores are assigned their affinity classes in a similar fashion.

In mathematical terms, let $\mathcal{A}_{s,1,k}$, $\mathcal{A}_{s,2,k}$,$\mathcal{A}_{s,3,k}$, ..., $\mathcal{A}_{s,m,k}$ be the array regions accessed by processor s from arrays Z_1, Z_2, Z_3, ..., Z_m, respectively, in loop nest k (i.e., they form an affinity class for processor s). In computing $\mathcal{Z}_{s,j}$, our approach uses these regions. After computing $\mathcal{Z}_{s,j}$, the sets $\mathcal{Q}_{s,j,1}$, $\mathcal{Q}_{s,j,2}$, ..., $\mathcal{Q}_{s,j,n}$, i.e., the iterations from the different nests that are assigned to pro-

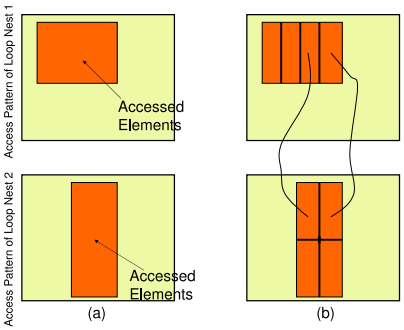

Figure 4: An example data access pattern and partitioning scenario. (a) The parts of the array accessed by two different nests. (b) The array partitioning across four processor cores.

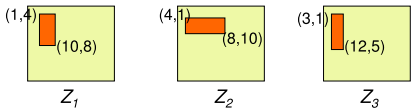

Figure 5: An example access pattern scenario on three arrays. The three array regions accessed collectively define an affinity class (i.e., the set of elements assigned to a single core). The top-left and bottom-right points of the data region accessed from each array are marked.

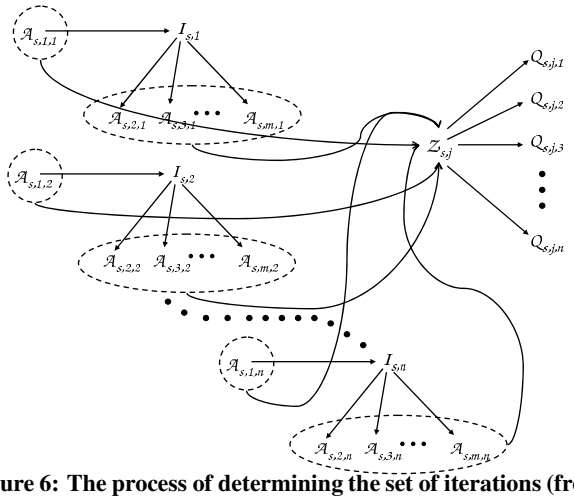

Figure 6: The process of determining the set of iterations (from all the nests) that will be executed by processor s (assuming that we have n nests).

Benchmark Name	Number of C Lines	Number of Arrays	Data Size	Cache Misses	Memory Energy
Compress	127	6	705.3KB	72758	183.18mj
Conv-Spa	231	8	542.2KB	59018	109.66mj
Filter	270	11	496.6KB	51254	115.71mj
Laplace	438	10	882.5KB	95332	301.09mj
LU-Decomp	86	2	707.7KB	70663	280.34mj
Minkowski	455	8	904.3KB	88496	602.20mj
Seg-02	622	10	830.0KB	50762	584.78mj
Text	1018	14	912.0KB	98198	813.91mj

Figure 7: Benchmarks used in this study. The values in the last two columns are for the loop based scheme.

cessor s, can be computed as has been discussed earlier. Figure 6 gives an illustration of this process. Note that, this calculation is just for processor s and needs to be carried out for each processor separately.

It must be noted that the global scheme, as it is explained so far, is data oriented; i.e., its main goal is to optimize cache behavior. However, optimizing cache behavior alone may not guarantee load balance across the processor cores. To illustrate this potential problem, let us consider a simplistic scenario where two processors access an array in different nests. Our approach guarantees that each core accesses the same array region in different nests; however, this does not mean that the cores execute the same number of loop iterations. In fact, it is possible that the number of iterations executed by one of the cores can be many more than the other (simply because, for example, there are more references to one part of the array). In such cases, load imbalance can dominate the locality behavior and the overall performance can suffer. To remedy this situation when it happens, we augment the global code parallelization scheme with a *load balancing step.* The objective of this step is to transfer some iterations from one (or more) core(s) to others in an attempt to balance the workload across the cores. To do this, it first identifies the loop nests with load imbalance and then measures the severity of the imbalance. If the load imbalance is severe, it determines the set of processors with large load than the others and redistributes some of their iterations. The concept of "severity" is measured using a parameter, which captures the largest load imbalance. Specifically, in our current implementation, if the load of any core is more than twice of that of any other core, we assume that the load imbalance is severe.

Before giving an example, we also want to emphasize that the proposed approach is different from loop fusion and similar techniques that operate neighboring loop nests in the code. In contrast to these techniques, the approach proposed in this paper considers *all* the nests at the same time, and in general, the output (transformed) code generated by our approach cannot be obtained by simple loop fusioning.

4.4 Example

Consider the example code fragment shown below, assuming that the data dependence analysis reveals that all the loops of both the nests can be executed in parallel, i.e., there is no cross-loop data dependences, except for t_2 in the first nest:

$$
\begin{array}{l}
arrays \quad Z_1[N][N], Z_2[N][N], Z_3[N][N]; \\
\\
do\ t_1 = 1, N \\
\quad do\ t_2 = 1, N \\
\quad\quad ... = Z_1[t_1][t_2] + Z_2[t_2][t_1] + ... \\
do\ t_1 = 1, N \\
\quad do\ t_2 = 1, N \\
\quad\quad ... = Z_1[t_2][t_1] + Z_3[t_2][t_1] + ...
\end{array}
$$

Since we try to achieve coarse-grain parallelism, we parallelize only a single loop from each nest. Let us also assume that the data dependences prevent any loop transformation on the first nest, whereas the second loop can be restructured. Considering the first loop nest, one can see that if we parallelize t_1, this means each processor accesses a block of consecutive rows from array Z_1, a block of consecutive columns from array Z_2. Focusing on the second nest however indicates that, we can have different array partitionings depending on whether t_1 or t_2 is parallelized. Since the first loop nest is more restricted, our approach divides the arrays in such a way that a processor takes a consecutive set of rows from arrays Z_1 and Z_3, and a consecutive set of columns from array Z_2. Consequently, the iteration assignment for the first processor is determined as follows. In the first loop nest, it executes iterations (t_1, t_2) such that $1 \leq t_1 \leq N/4$ and $1 \leq t_2 \leq N$. On the other hand, from the second loop nest, it executes iterations (t_1, t_2) such that $1 \leq t_1 \leq N$ and $1 \leq t_2 \leq N/4$. Note that, with this assignment, this processor uses the same set of elements from array Z_1 in both the nests. The iteration assignments for the other three processor cores can be derived similarly. Notice that, if we have just used a loop based parallelization, we could have assigned a different set of iterations to the first processor from the second nest, e.g., (t_1, t_2) where $1 \leq t_1 \leq N/4$ and $1 \leq t_2 \leq N$. But, as can be seen, this assignment does not exploit inter-nest data reuse from a processor's perspective.

5. EXPERIMENTAL EVALUATION

5.1 Setup and Implementation

We implemented the global code optimization approach discussed in this paper within SUIF, an experimental compiler from Stanford University [2]. SUIF is structured as a series of passes, each of which operates on the same intermediate format (IR). We implemented our approach as a separate pass within SUIF. Our implementation takes as input an application code written in C and the number of processors (p) over which the code is to be parallelized, and generates as output a parallelized code based on data reuse as explained in the text. However, to generate the code that executes only the iterations assigned to a core, we also employ a polyhedral tool [15]. The increase in compilation times (over the loop based approach) due to the global parallelization strategy was about 60% when averaged over all the benchmarks we tested (the bulk of this time was spent within the polyhedral tool in generating codes for individual cores).

We obtain our results through simulation. Unless otherwise stated, we simulate an MPSOC architecture where each processor core has an 8KB L1 with an access latency of 2 cycles. We also assume a main memory access latency of 80 cycles.

Figure 7 gives the eight benchmark codes used in this study. We collected this particular set of benchmarks since we believe that they represent a good mix of array intensive embedded applications that could be run on MPSOCs. The third column of this table gives the number of C lines for each benchmark and the next column shows the number of arrays accessed by each benchmark. The fifth column gives the amount of total data accessed and the sixth column shows the number of data cache (L1) misses accumulated across all processors when the loop based parallelization is used with 8 processors. Finally, the last column gives the memory energy consumption due to data accesses (obtained using the CACTI tool under the 70nm process technology). What we mean by "memory energy" here is sum of the energies consumed in the data cache and the off-chip memory. While we observed that our global approach reduces both memory energy consumption and execution cycles, in this paper we present mostly energy numbers as performance improvements are similar (we present only a single set of performance numbers). The values given in the last two columns of this table are for the loop based scheme.

5.2 Results

Figure 8 gives the memory energy consumption of the global scheme, as a fraction of the memory energy consumption of the loop based scheme (see the last column of Figure 7). Each point on the x-axis corresponds to a particular number of processors used in parallel execution (from 2 to 12). An important observation from these curves is that the effectiveness of the global scheme generally (but not always) increases with increasing number of processors. This is due to the fact that, when the number of cores is increased

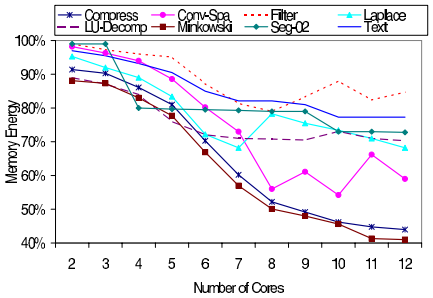

Figure 8: Memory energy consumption.

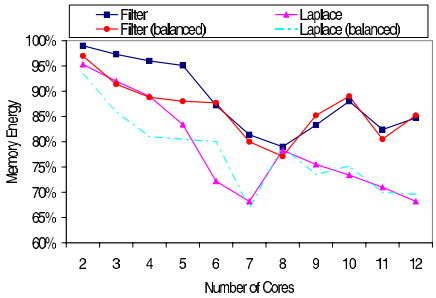

Figure 9: Impact of workload balancing.

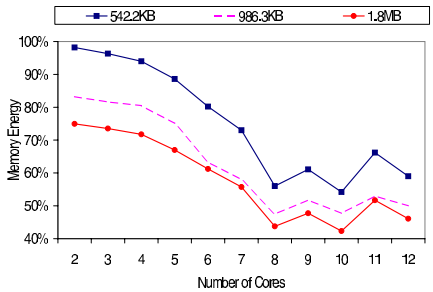

Figure 10: Impact of data size.

(under the same data size), the data accessed by each processor is usually fragmented more in the memory space and this reduces the chances for data sharing across the different nests when the loop based strategy is employed.

We next quantify the benefits of workload balancing (following the application of the global scheme) as explained in Section 4.3. We focus on two of our benchmarks, namely, Filter and Laplace, as these are the two codes with the largest workload imbalance across processors. The graph in Figure 9 gives the curves for these two codes when load balancing is used. The curves without load balancing are also reproduced here for ease of comparison. One can see from these curves that the impact of load balancing is more pronounced with the small number of cores. This is because when the number of cores is small, the load imbalance across them is more significant. In contrast, with a larger number of cores, trying to balance the workloads may not be as effective.

The next set of results is on the impact of dataset size on the behavior of the global approach. We present in Figure 10 the results for only Conv-Spa with three different data sizes (one of them being the original one used in the experiments so far); the rest of the benchmarks showed similar trends, and we do not present them here in detail. An important observation from Figure 10 is that the global scheme brings more savings over the loop based scheme when the data size is increased. This is due to the fact that increasing data sizes makes optimizing for data locality more important (as cache behavior becomes more critical).

Our final set of results concentrate on performance improvements. The normalized execution cycles of the benchmarks under our global approach are given in Figure 11 for processor counts 2, 6, and 12. Each bar is normalized with respect to the loop based scheme that uses the same number of processors. We see from these results that our approach brings performance benefits as well (especially with increasing number of processors). The reason that the performance savings are not as high as energy savings is that some of the performance penalty incurred by the loop based scheme can be hidden in parallel execution. However, the energy overheads it incurs cannot be hidden. Since our (performance and energy) savings are given with respect to the loop based scheme, we witness higher energy savings than performance savings.

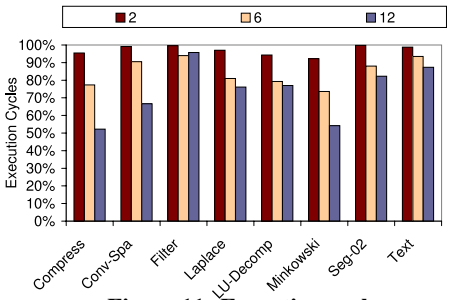

Figure 11: Execution cycles.

6. CONCLUSIONS

Memory behavior of embedded MPSOC applications is the main factor that determines their behavior from both performance and power perspectives. Memory behavior in turn is largely shaped by how the application code is parallelized over multiple processor cores and by how the processors access data and communicate/synchronize with each other. While conventional loop based code parallelization can be successful in several applications, since such techniques do not capture the interactions (e.g., data sharings) across the different nests of a given application, they perform poorly for other applications. The work presented in this paper proposes, fully implements, and experimentally evaluates a global (application wide) parallelization strategy based on reusing data across different loop nests. The idea is to capture the data reuse across the loop nests and assign iterations to processor cores in such a way that a given processor accesses the same set of data elements in different loop nests as much as possible. We implemented this global parallelization strategy within an optimizing compiler and tested its effectiveness using a set of eight embedded benchmark codes. Our experiments indicate that this global strategy brings significant improvements – in terms of both energy consumption and execution cycles perspectives – over the conventional loop based code parallelization strategy.

7. REFERENCES

[1] A. Agarwal, D. Kranz, and V. Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared memory multiprocessors. In Proc. *International Conference on Parallel Processing,* 1993.

[2] 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.

[3] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[4] F. Catthoor et al. Custom memory management methodology – exploration of memory organization for embedded multimedia system design. *Kluwer Academic Publishers,* 1998.

[5] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. In Proc. *Conference on Programming Language Design and Impl.*, 1995.

[6] I. Kadayif, M. Kandemir, and M. Karakoy. An energy saving strategy based on adaptive loop parallelization. In Proc. *DAC,* 2002.

[7] I. Kadayif, I. Kolcu, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Exploiting processor workload heterogeneity for reducing energy consumption in chip multiprocessor. In Proc. *DATE,* 2004.

[8] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multilevel blocking. In Proc. *ACM Conf. on Programming Language Design and Impl.*, 1997.

[9] V. Krishnan and J. Torrellas. A chip multiprocessor architecture with speculative multi-threading. *IEEE Transactions on Computers, Special Issue on Multi-threaded Architecture,* 1999.

[10] J. Li, J. Martinez, and M. Huang. The thrifty barrier: Energy-efficient synchronization in shared-memory multiprocessors. In Proc. *High Performance Computer Arch.,*, 2004.

[11] S. A. McKee and W. A. Wulf. Access ordering and memory-conscious cache utilization. In Proc. *Symposium on High-Performance Computer Arch.,* 1995.

[12] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of compiler algorithm for prefetching. In Proc. *International Conference on Architectural Support for Programming Languages and Operating Systems,* 1992.

[13] MP98: a mobile processor. http://www.labs.nec.co.jp/MP98/top-e.htm.

[14] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single chip multiprocessor. In Proc. *Conference on Architectural Support for Programming Languages and Operating Systems,* 1996.

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

[16] R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The energy efficiency of CMP vs. SMT for multimedia workloads. In Proc. *the 18th Annual International Conference on Supercomputing*, 2004.

[17] W.-T. Shiue and C. Chakrabarti. Memory exploration for low-power embedded systems. In Proc. *Design Automation Conference,* 1999.

[18] O. Temam, E. D. Granston, and W. Jalby. To copy or not copy: A compile-technique for assessing when data copying should be used to eliminate cache conflicts. In Proc. *Supercomputing'93,* 1993.

[19] M. Wolf and M. Lam. A data locality optimizing algorithm. In Proc. *Conference on Programming Language Design and Impl.*, 1991.