Exploiting Shared Scratch Pad Memory Space in Embedded Multiprocessor Systems

Mahmut Kandemir
Microsystems Design Lab
Pennsylvania State University
University Park, PA 16802
kandemir@cse.psu.edu

J. Ramanujam
Dept. Elec. & Comp. Engr.
Louisiana State University
Baton Rouge, LA 70803
jxr@ece.lsu.edu

A. Choudhary
Dept. Elec. & Comp. Engr.
Northwestern University
Evanston, IL 61801
choudhar@ece.nwu.edu

ABSTRACT

In this paper, we present a compiler strategy to optimize data accesses in regular array-intensive applications running on embedded multiprocessor environments. Specifically, we propose an optimization algorithm that targets the reduction of extra off-chip memory accesses caused by inter-processor communication. This is achieved by increasing the application-wide reuse of data that resides in the scratch-pad memories of processors. Our experimental results obtained on four array-intensive image processing applications indicate that exploiting inter-processor data sharing can reduce the energy-delay product by as much as 33.8% (and 24.3% on average) on a four-processor embedded system. The results also show that the proposed strategy is robust in the sense that it gives consistently good results over a wide range of several architectural parameters.

Categories and Subject Descriptors B.3 [**Hardware**] Memory Structures; D.3.4 [**Software**] Programming Languages: Processors [Compilers]

Terms Algorithms, management, performance.

Keywords Embedded multiprocessors, energy consumption, scratch pad memories, access patterns, compiler optimizations, data tiles.

1. INTRODUCTION

The mobile/embedded computing device market is projected to grow to 16.8 million units in 2004, representing an average annual growth rate of 28% over the five year forecast period [10]. This brings up the issue of optimizations in application design, system software and circuit/architectural levels for resource-constrained devices, where battery energy and limited storage capabilities are brought into spotlight. While there have been significant strides made in optimizing energy and performance using circuit and architectural level optimizations, not many studies have looked at the problem of software level optimizations for mobile/embedded systems. Consequently, any effort in this direction will help us to realize the goal of achieving high performance and low energy consumption in mobile/embedded devices.

As microprocessors grow more and more powerful, designers are building larger and ever more sophisticated systems to solve complex problems. In particular, in the area of digital signal and video processing, specialized microprocessors are getting more powerful, making them attractive for a wide range of embedded applications. As a result of this, embedded system designers are now using multiple processors in a single system (either in the form of SoC or in the form of a multiprocessor board) to address the computational requirements of a wide variety of applications. In future, we can expect that more and more embedded systems will be based on multiprocessor architectures.

Many applications from image and video processing domains have significant data storage requirements in addition to their pressing computational requirements. Previous studies [5, 13] show that high off-chip memory latencies and energy consumptions are likely to be the limiting factor for future embedded systems. Therefore, compiler-based techniques for optimizing memory access patterns are extremely important. An optimizing compiler can be particularly useful in embedded image and video processing applications as many of these applications exhibit regular (compile time analyzable and optimizable) data access patterns. Previous work [2, 9] shows that regular data access patterns found in array-dominated applications can be better captured if we employ a scratch pad memory (a software-managed SRAM), instead of a more traditional data cache.

In this paper, we present a compiler-based strategy for optimizing energy consumption and memory access latency of array dominated applications in a multiprocessor based embedded system. Specifically, we show how a compiler can increase data sharing opportunities when multiple processors (each one is equipped with a scratch pad memory) operate on a set of arrays in parallel. *This is in contrast with previous work on scratch pad memories that exclusively focused on single processor architectures.* In this paper, we make the following major contributions:

- We show that inter-processor communication requirements in a multiprocessor embedded system can lead to extra off-chip memory requests and present a compiler strategy that eliminates these extra off-chip memory requests. Our strategy is implemented using an experimental compiler infrastructure [1] and targets array-dominated embedded applications.
- We report performance and energy consumption data showing the effectiveness of our optimization strategy. The results show that exploiting inter-processor data sharing can reduce energy-delay product by as much as 33.8% (and 24.3% on average) on a four-processor embedded system. The results also show that the proposed strategy is robust in the sense that it gives consistently good results when several architectural parameters are varied over a wide range.

The rest of this paper is organized as follows. In the next section, we discuss our multiprocessor architecture and execution model. In Section 3, we present details of our optimization strategy. In Section 4, we introduce our experimental setup and report experi-

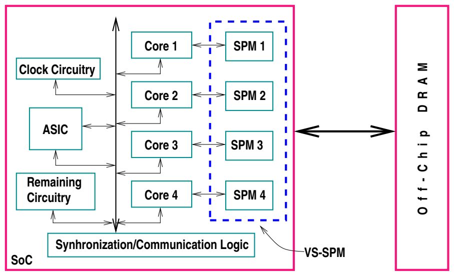

Figure 1: A VS-SPM based system architecture.

mental results. In Section 5, we offer our conclusions and give an outline of the planned future work on this topic.

2. ARCHITECTURE, EXECUTION MODEL

A scratch pad memory (SPM) is a fast SRAM that is managed by software (the application and/or compiler) [11, 9, 2, 14]. It can be used for optimizing both data and instruction accesses. In applications that exhibit regular data access patterns (which are compile-time analyzable and optimizable), an SPM can outperform a conventional data cache memory as software can lead to a better flow of data to/from SPM as compared to the conventional cache, whose management of the flow of data is at a fine-grain level (cache line granularity), controlled by hardware, and mostly independent of the application [9].

A *virtually shared scratch pad memory* (VS-SPM), on the other hand, is a shared SRAM space made up by individual SPMs of multiple processors. In this paper, we focus on a multiprocessor on-a-chip architecture, as shown in Figure 1. In this architecture, we have a system-on-a-chip (SoC) and an off-chip DRAM that can hold data as well as instructions. The SoC holds multiple processor cores (with their local SPMs), inter-processor communication/synchronization mechanism, clock circuitry, and some ASIC. The SPMs of individual processors make up a VS-SPM. Each processor has fast access to its own SPM as well as to SPMs of other processors. With respect to a specific processor, the SPMs of other processors are referred to as remote SPMs. Accessing remote SPMs is possible using fast on-chip communication links between processors. Accessing off-chip DRAM, however, is very costly in terms of both latency and energy. Since per access energy and latency of VS-SPM are much lower than the corresponding values of DRAM, it is important to make sure that as many data requests (made by processors) as possible are satisfied from the VS-SPM. Obviously, this can be achieved by maximizing the reuse of data in the VS-SPM. While an application programmer can achieve this by carefully chorographing data flow to/from VS-SPM, it is clear that this would be overwhelming for her/him. Instead, we show in this paper that automatic compiler support can achieve very good results by optimizing the reuse of data in the VS-SPM. It should be noted that each processor in the SoC can also contain an instruction cache and/or a loop cache. Their management, though, is orthogonal to the VS-SPM management.

An example system that is similar to the architecture considered in this paper is the VME/C6420 from Blue Wave Systems [3]. This system uses a 9-port crossbar on the board. Four of these ports connect to identical processors with their local SRAM modules, four to high-bandwidth I/O connections, and the ninth to an off-board memory. The ports on the crossbar enable high data movement rates (typically, 264 Mbytes/sec). Using crossbar links increases the net bandwidth provided by the system. Consequently, accessing local SRAM module and remote SRAM modules are much faster and much less energy consuming than accessing the off-chip DRAM. The local SRAM modules in Blue Wave correspond to local SPMs in our system; consequently, the compilation techniques for both the systems should be very similar (despite the fact that one of these is board-based and the other is SoC based).

The execution model in a VS-SPM based architecture is as follows. The system takes as input a *loop-level* parallelized application. In this model, each loop nest is parallelized as much as possible. In processing a parallel loop, all processors in the system participate computation and each executes a subset of loop iterations. When the execution of a parallel loop has completed, the processors synchronize using a special construct called *barrier* before starting the next loop. The synchronization and communication between processors is maintained using fast on-chip communication links. Based on the parallelization strategy, each processor works on a portion of each array in the code. Since its local SPM space is typically much smaller than the portion of the array it is currently operating on, it divides its portion into chunks (called *data tiles*) and operates on one chunk at a time. When a data tile has been used, it is either discarded or written back into off-chip memory (if modified).

In order to improve the reuse of data in the VS-SPM, one can consider *intra-processor data reuse* and *inter-processor data reuse.* Intra-processor data reuse corresponds to optimizing data reuse when considering the access pattern of each processor in isolation. Previous work presented in [11] and [9] addresses this problem. It should be noted, however, that exploiting intra-processor reuse only may not be very effective in a VS-SPM based environment. This is because intra-processor data reuse has a local (*processor-centric*) perspective and does not take inter-processor data sharing effects into account. Such effects are particularly important in applications where data regions touched by different processors overlap. This is very common in many array-intensive image processing applications. Inter-processor data reuse, on the other hand, focuses on the problem of optimizing data accesses considering access patterns of all processors in the system. In other words, it has an *application-centric* view of data accesses.

In order to see the difference between application-centric and processor-centric views, let us consider the code fragment in Figure 2 which performs Jacobi iteration over two $N \times N$ square arrays U_1 and U_2. In this code fragment, $f(.)$ is a linear function and $parfor$ indicates a parallel for-loop whose iterations are to be distributed (evenly) across processors available in the system. Note that since this loop does not have any data dependences, both the for-loops are parallel. Since our approach works on an already parallelized program, we do not concern ourselves with the question of how the code has been parallelized (i.e., by application programmer or by an optimizing compiler). Assuming that we have four processors (P_1, P_2, P_3, and P_4) as shown in Figure 1, the portion of array U_2 accessed by each processor is shown in Figure 3(b). Each processor is responsible from updating an $(N/2) \times (N/2)$ sub-array of U_2. The portions accessed by processor 1 from array U_1 are shown in Figure 3(a). This portion is very similar (in shape) to its portion from array U_2 except that it also includes some elements *shared* (accessed) by other processors. In the remainder of this paper, such elements are called *non-local elements* (or border elements). Assuming that the arrays U_1 and U_2 initially reside in off-chip DRAM, each processor brings a data tile of its U_1 sub-array and a data tile of its U_2 sub-array from DRAM to VS-SPM, updates the corresponding elements, and stores the U_2 data tile back in the DRAM. Figure 3(c) shows seven data tiles (from U_2) that belong to processor 4. This processor accesses these tiles starting from tile 1 and ending with tile 7.

Let us first see how each processor can make effective use of its local SPM space (processor-centric optimization). We focus on processor 1, but our discussion applies to any processor. Processor 1 first brings a data tile from U_2 and a data tile from U_1 from off-chip memory to its SPM. After computing the new values for its U_2 data tile, it stores this data tile back in off-chip memory and

```
parfor (i=2; i <= N-1; i++)
 parfor (j=2; j <= N-1; j++)
  U2[i][j] += f(U1[i][j-1]+U1[i-1][j]+U1[i][j+1]+U1[i+1][j])
```

Figure 2: Jacobi iteration.

proceeds by bringing new data tiles from U_1 and U_2. However, to exploit reuse, it keeps the last row of the previous U_1 data tile in SPM. This is because this row is also needed when computing the elements of the new U_2 tile. This optimization is referred to as local buffering or processor-centric data reuse.

While such a tiling strategy makes effective use of the SPM space as far as intra-processor reuse is concerned, it fails to capture inter-processor data reuse. Let us consider how our four processors execute this nest in parallel. Figure 3(d) illustrates the scenario where four processors are working on their first data tiles from array U_1 (assuming row-block data tiles). Let us focus on processor 3; similar discussion applies to other processors as well. This processor, while working on its first data tile, needs data from processors 1, 2, and 4. More specifically, it needs an entire row from processor 1 (that is, the last row of processor 1's last tile), a single element from processor 2, and two elements from processor 4. These non-local elements are also shown in Figure 3(d). It should be noted that processor 4 can supply these elements immediately from its local SPM. This is because these two array elements are part of the data tile it is currently working on. However, processors 1 and 2 need to perform off-chip memory accesses for the data required by processor 3 as these data are not currently in their local SPMs. Obviously, these *extra off-chip memory requests* (that is, memory requests performed due to inter-processor communication requirements only) will be very costly. A similar scenario occurs when we consider a different data tile shape. Figure 3(e) shows processor access patterns (to array U_1) when a square data tile is used. It also shows the non-local elements required by processor 4 when it is operating on its first data tile. We can easily see that none of these elements are in any SPM (as other processors are also working on their first data tiles). Consequently, in order for processor 4 to complete its computation on its first tile, other processors need to perform extra off-chip memory accesses. It should also be noted that it is not a good idea to try to bring the non-local elements to the local SPM and keep them there until they are requested by other processors. This is because such a strategy would lead to keeping data in the SPM without much reuse and decrease overall SPM space utilization.

However, since the code fragment in Figure 2 does not exhibit any data dependence, the processors do not have to stick to the same tile processing order; that is, they do not have to process their data tiles in the same order. In particular, they can bring (and process) their data tiles in such a way that whenever one processor needs a non-local array element (to perform some computation), the needed data can be found in some remote SPM. If this can be achieved for all non-local accesses, we can eliminate all the extra off-chip memory accesses due to inter-processor communication. Figures 3(f) and (g) show how extra off-chip memory requests can be eliminated when row-block and square data tiles are used, respectively. Note that in these scenarios each processor has a different tile access pattern. The following section present an automatic compiler-directed tile processing strategy to achieve this and Section 4 measures potential benefits of doing so.

3. COMPILER SUPPORT

There are at least two sub-problems in compiling array-dominated applications for a VS-SPM based environment:

– *Data Tile Shape/Size Selection:* The first step in compilation is to determine the shape and sizes of data tiles. The important parameters in this process are the available SPM space and data access pattern of the application. While this problem is important, it is beyond the scope of this paper. In this paper, we assume rectilinear tile shapes and that all processors have the same SPM capacity and operate with identical data tiles (whose size is determined by the local SPM size).

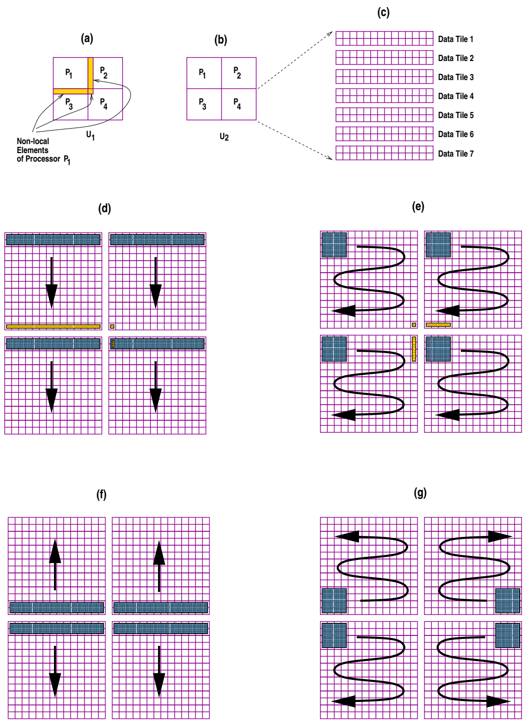

Figure 3: (a-b) Local array portions of processors for the arrays accessed in Figure 2. (c) Data tiles to be processed by a processor. (d-g) Different tile access patterns.

– *Tile Access Pattern Detection:* In this step, which we also call *scheduling*, given a data tile shape/size, we want to determine a data tile access pattern (for all processors) such that extra off-chip memory accesses (due to inter-processor communication) are eliminated. In the rest of this section, we present a compiler technique that addresses this problem.

The degree of freedom [4] of a given data tile indicates the its movement capability on data space. For instance, the degree of freedom for the data tile given in Figure 3(d) is one as it can move only in one direction (in vertical direction), whereas the degree of freedom for the data tile shown in Figure 3(e) is two as it can move in both horizontal and vertical directions.

We define a *tile access pattern matrix* (*scheduling matrix*), denoted $\mathcal{H}$, which determines the order in which the data tiles are accessed. The dimensions of a scheduling matrix are decided based on the degree of freedom. Let us focus on two-dimensional arrays; our results extend to higher-dimensional arrays as well. In the two-dimensional case, if the degree of freedom is one, the scheduling matrix is 2×1; if the degree of freedom is two, the scheduling matrix is 2×2. For nested loops that do not contain flow-dependences, the tile access patterns and their corresponding scheduling matrices for a degree of freedom of 1 and 2 are given in Figures 4(a) and (b), respectively. Each column of H corresponds to an axis and the value of the vector in a column denotes the direction of the access along the corresponding axis. In addition to the elements of each column, the ordering of the columns of H is extremely important. The first column denotesAs an example,

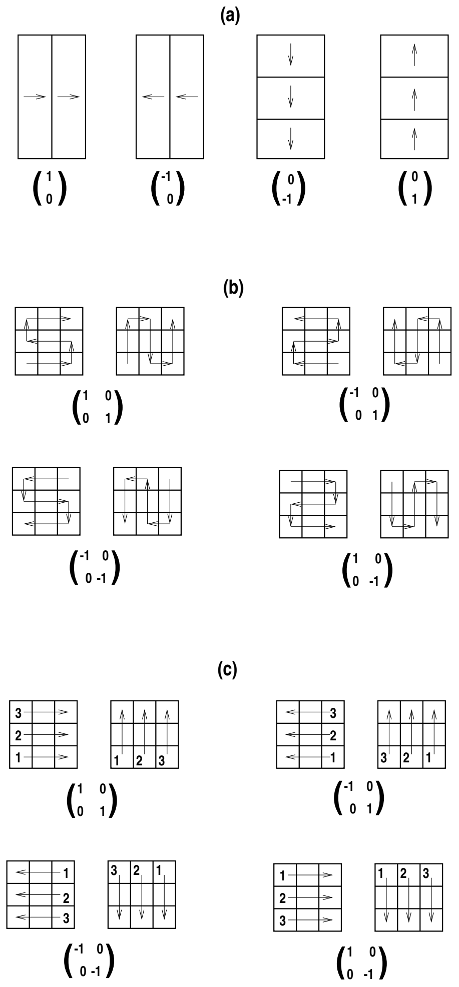

Figure 4: Tile access patterns and scheduling matrices for two-dimensional arrays accessed in a nested loop.

the matrix $\mathcal{H} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$ corresponds to positive direction of movement (at the beginning of the pattern) in the x-axis and negative direction of movement in the y-axis.

We also define *direction vectors* for each processor with respect to its neighbors. For example, the direction vector of processor 1 with respect to processor 4 (see Figure 3(a)) is: $\vec{v}_{1,4} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$. The direction vector of processor i with respect to processor j is denoted $\vec{v}_{i,j}$. Essentially, this is a vector whose elements indicate the signs (positive denoted by $+1$, negative denoted by -1 and zero by 0) of the vector distance from the co-ordinates of processor i to the co-ordinates of processor j.

The objective of our tile access pattern strategy should be minimizing the extra off-chip memory accesses. This can be achieved by making sure that whenever a processor needs a non-local array element, some other processor can supply it from its SPM. To achieve this, our scheduling strategy uses the following result (proof omitted for lack of space):

Lemma 1. Consider two processors i and j with their scheduling matrices $\mathcal{H}_i$ and $\mathcal{H}_j$, respectively. Schedules denoted by these matrices eliminate extra off-chip memory accesses (when considering only these two processors) if and only if they satisfy the following equality called the *scheduling equality*: $\mathcal{H}_i^T \vec{v}_{i,j} = \mathcal{H}_j^T \vec{v}_{j,i}$.

Our scheduling algorithm consists of three steps:

(1) Assign a *symbolic* scheduling matrix to each processor. The rank of the scheduling matrix will be equal to the degree of freedom of data tiles.

(2) Construct scheduling equalities for each processor pair using direction vectors and scheduling matrices. These equalities collectively represent the constraints that need to be satisfied for eliminating all extra DRAM accesses due to inter-processor communication.

(3) Initialize the scheduling matrix of a processor with an arbitrary schedule (e.g., using one of the matrices in Figures 4(a) or (b)) and compute the corresponding scheduling matrices (tile access patterns) of the remaining processors by solving the scheduling equalities.

As an example application of this algorithm, consider the nest in Figure 2 assuming four processors and a degree of freedom of 2 (Figure 3(e)). In the first step, we assign symbolic scheduling matrices for processors. Let $\mathcal{H}_i = \begin{pmatrix} {h^i}_{1,1} & {h^i}_{1,2} \\ {h^i}_{2,1} & {h^i}_{2,2} \end{pmatrix}$ be the scheduling matrix for processor i. In the second step of our scheduling algorithm, we compute scheduling equalities. For example, scheduling equalities for processor 2 are:

$$\begin{pmatrix} {h^2}_{1,1} & {h^2}_{2,1} \\ {h^2}_{1,2} & {h^2}_{2,2} \end{pmatrix} \begin{pmatrix} -1 \\ -1 \end{pmatrix} = \begin{pmatrix} {h^3}_{1,1} & {h^3}_{2,1} \\ {h^3}_{1,2} & {h^3}_{2,2} \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} {h^2}_{1,1} & {h^2}_{2,1} \\ {h^2}_{1,2} & {h^2}_{2,2} \end{pmatrix} \begin{pmatrix} 0 \\ -1 \end{pmatrix} = \begin{pmatrix} {h^4}_{1,1} & {h^4}_{2,1} \\ {h^4}_{1,2} & {h^4}_{2,2} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} {h^2}_{1,1} & {h^2}_{2,1} \\ {h^2}_{1,2} & {h^2}_{2,2} \end{pmatrix} \begin{pmatrix} -1 \\ 0 \end{pmatrix} = \begin{pmatrix} {h^1}_{1,1} & {h^1}_{2,1} \\ {h^1}_{1,2} & {h^1}_{2,2} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Equalities for other processors are constructed in a similar fashion. In the last step, we determine individual scheduling matrices. To do so, we initialize the scheduling matrix of a processor to a schedule and find the scheduling matrices of the remaining processors. For example, if we set: $\mathcal{H}_1 = \begin{pmatrix} {h^1}_{1,1} & {h^1}_{1,2} \\ {h^1}_{2,1} & {h^1}_{2,2} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, we find $\mathcal{H}_2 = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$; $\mathcal{H}_3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$; $\mathcal{H}_4 = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$. These scheduling matrices give us the tile access patterns shown in Figure 3(g). Note that these access patterns do not incur any extra off-chip memory accesses originating from inter-processor communication.

The scheduling algorithm for a nested loop that may contain flow-dependences is the same as that of a loop that does not contain any flow-dependences except that the scheduling matrix selected should respect all data dependences. Particularly, when flow-dependences [15] are involved, we cannot have snake-like tile access patterns shown in Figure 4(b). Instead, for the cases with a degree of freedom of 2, we can employ the tile access patterns and the associated scheduling matrices shown in Figure 4(c). The numbers attached to arrows in this figure indicate the order of accesses. Our approach has two steps. First, we run only the first two steps of the algorithm given for the non-flow dependence case, and obtain all scheduling equalities. Then, instead of just initializing one of the scheduling matrix to an arbitrary form and determining the others (as in the previous case), we try all possible scheduling matrices for one of the processors and, for each case, we determine the corresponding scheduling matrices for other processors. In other words, instead of just one solution, we come up with multiple solutions. Among these solutions, we select the one (if any) that does not violate any data dependences. In case we have no such a matrix, we employ a default scheduling scheme that does not break any dependences. Obviously, the default strategy will not necessarily eliminate extra DRAM accesses.

As an example, let us consider the successive-over-relaxation (SOR) loop shown in Figure 5. In this code fragment, if we ap-

```
for (i=2; i <= N-1; i++)
 parfor(j=1; j <= N-1; j++)
  U_1[i][j] += g(U_1[i-1][j]+U_1[i][j+1]+U_1[i+1][j])
```

Figure 5: SOR iteration.

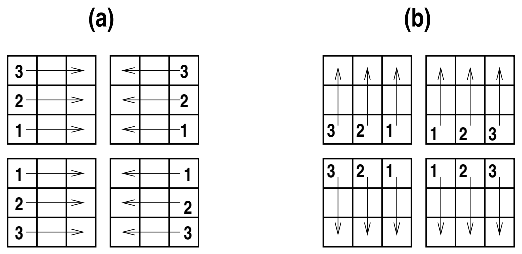

Figure 6: Illegal (a) and legal (b) access patterns for the code fragment in Figure 5.

ply our three-step strategy, one possible schedule would have the following scheduling matrices (assuming four processors): $\mathcal{H}_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}; \mathcal{H}_2 = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}; \mathcal{H}_3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}; \mathcal{H}_4 = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$. This is shown in Figure 6(a). Since the schedules for processors 2 and 4 violate data dependences, they are not acceptable. Another schedule has these scheduling matrices: $\mathcal{H}_1 = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}; \mathcal{H}_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}; \mathcal{H}_3 = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}; \mathcal{H}_4 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$. The tile access pattern corresponding to this schedule is given in Figure 6(b). We note that this is a legal schedule. In should be noted however that in the case of flow-dependences, it may not always be possible to eliminate all extra off-chip memory accesses.

4. EXPERIMENTS

4.1 Experimental Setup

Our experimental setup consists of a compiler environment and an in-house simulator. Our optimization algorithm is implemented using the SUIF (Stanford University Intermediate Format) experimental compiler infrastructure [1]. SUIF consists of a small, clearly documented kernel and a toolkit of compiler passes built on top of the kernel. The kernel defines the intermediate representation, provides functions to access and manipulate the intermediate representation, and structures the interface between compiler passes. The toolkit currently includes C and Fortran front-ends, a loop-level parallelism and locality optimizer, an optimizing MIPS back-end, a set of compiler development tools, and support for instructional use. The output of SUIF is a C code which can be compiled using the native compiler of the platform in question.

To test the effectiveness of our strategy, we used four array-dominated applications (written in C) from the image processing domain: `3D`, `dfe`, `splat`, and `wave`. `3D` is an image-based modeling application that simplifies the task of building 3D models and scenes. `dfe` is a digital image filtering and enhancement code. `splat` is a volume rendering application which is used in multi-resolution volume visualization through hierarchical wavelet splatting. It is used primarily in the area of morphological image processing. And finally, `wave` is a wavelet compression code that targets specifically medical applications. This code has a characteristic that it can reduce image data to an extremely small fraction of its original size without compromising image quality significantly. These C programs are written so that they can operate on images of different sizes. The total input sizes used in our experiments

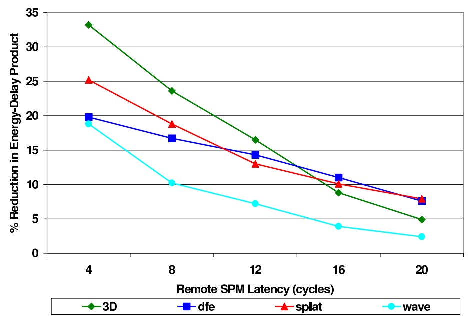

Figure 7: % savings in energy-delay product (base configuration).

were 305KB, 286KB, 635KB and 628KB for `3D`, `dfe`, `splat` and `wave`, respectively.

Our simulator takes a parallel code written in C as input and simulates a multiprocessor architecture. Each simulated processor is a 100MHz MIPS 4Kp core with a five-stage pipeline that supports four execution units (integer, multiply-divide, branch control, and processor control). Each core has thirty-two, 32-bit general-purpose registers. We assume a local SPM (SRAM) attached to each core with an access latency of 2 cycles. We also simulate inter-processor communication. To test the robustness of strategy, we experimented with different remote SPM access latencies (from 4 cycles to 16 cycles). We also assumed an extra 1 cycle latency for each inter-processor synchronization operation. The simulated off-chip DRAM is 4MB with an access latency of 80 cycles. The simulator outputs the number of accesses to each SPM, number off off-chip accesses, number and volume of inter-processor communications, and overall execution time of the application. To parallelize the applications, we adopted an aggressive strategy in which all the loops in a given nest (except the innermost one) are parallelized (provided that data dependences allow that). The energy model used for SPMs is very similar to that of a cache memory [12] except that it assumes full associativity and does not consider a tag array. The energy model used by our simulator for interconnects is transition-sensitive and is very similar to that presented in [16]. Since we focus only on data memory performance and energy, we do not report data on instruction accesses and datapath activity. In computing the executing cycles, however, all the cycles spent in datapath (including the stall cycles) are accounted for.

4.2 Results

Due to space concerns, we present data for only energy-delay product. Energy-delay product is a suitable metric that allows us to evaluate the impact of an optimization on both energy and performance (execution cycles) [6, 8]. In our context, the energy component of the energy-delay product corresponds to the memory system energy due to data accesses. This includes the energy consumed in SPMs, interconnect between processors, off-chip DRAM accesses, and in the interconnect between SoC and off-chip DRAM. The delay is the overall parallel execution time of the application. In our experiments, we compare two different versions of each benchmark. The first version is the code with only local SPM optimizations; it does *not* try to eliminate extra DRAM accesses due to inte-processor communication. The second version is the result of our VS-SPM based optimization strategy. All results presented in this subsection are *percentage improvements* brought by the second version over the first version.

Figure 7 gives the energy-delay product improvements for our base configuration with different values of remote SPM access latency. In the base configuration, the access latencies for local SPM and off-chip DRAM are 2 and 80 cycles, respectively. Also, the

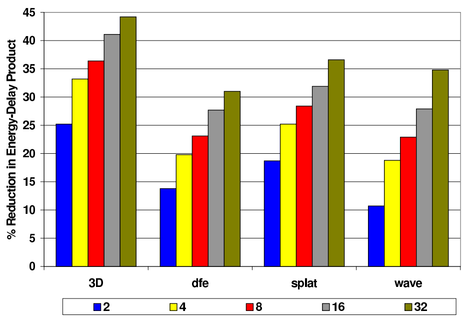

Figure 8: % savings in energy-delay product (sensitivity to the number of processors).

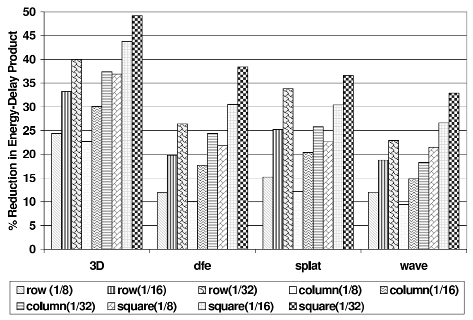

Figure 9: % savings in energy-delay product (sensitivity to the shape of the tile and size of the available SPM space).

number of processors is four, the row-block data tiles are used, and the available local SPM space is 1/8th of the local portion of the processor (this ratio is called the *slab ratio*). Note that the slab ratio can be changed by changing the SPM size or array sizes; in our experiments, we changed the SPM size. It should be mentioned that when the remote SPM latency is increased, the corresponding per access energy consumption (for remote SPM) is also proportionally increased. We observe from these results that our approach generates best results with the smallest remote SPM latencies (as expected). The average improvements when remote SPM latency is 4 and 20 are 24.3% and 5.7%, respectively.

To measure the sensitivity of our approach to the number of processors, we performed another set of experiments. We used processor sizes of 2, 8, 16, and 32. All other parameters are the same as in the base configuration. The results shown in Figure 8 (which also include our default processor size of 4) indicate that the effectivess of our strategy increases when the number of processors is increased. This is because an increase in the number of processors leads to an increase in inter-processor communication volume, and consequently, the version without VS-SPM optimization performs very frequent extra DRAM accesses. Therefore, we observe an increase in percentage improvements.

The last set of experiments gauge the sensitivity of our strategy to the shape of the tile and size of the available SPM space. Each bar in Figure 9 indicates the shape of the tile (row-block, column-block, or square) and the slab ratio (within parantheses). It can be observed that our approach is more effective with smaller slab ratios. Note that a smaller slab ratio represents more pressure on data memory. Considering the fact that applications are processing larger and larger data sets, this result is encouraging. We also observe that we save more when square tiles are used. This is due to the fact that the performance (and energy behavior) of the original codes is extremely poor when square tiles are used.

5. CONCLUSIONS AND FUTURE WORK

Efficient compilation techniques for complex programs are particularly important for array-dominated embedded applications. As microprocessors grow more and more powerful, designers are building more sophisticated embedded/mobile systems. An important trend in such systems is the increased use of multiple processors. This paper presents a compiler-directed optimization strategy for exploiting software-controlled, shared SRAM space in a multiprocessor embedded system. Our approach is oriented towards eliminating extra off-chip DRAM accesses caused by inter-processor communication. The results obtained using four array-intensive applications show that significant reductions in energy-delay product are possible using this approach. This work can be extended in several ways. First, we plan to perform experiments with multiple levels of SPMs and hybrid multiprocessor environments that consist of both SPMs and data caches. Second, we would like to investigate different compiler optimizations for optimizing inter-processor communication and study the impact of these optimizations on off-chip memory accesses. Third, we want to explore compiler optimizations that exploit array sharing between different loop nests in an embedded multiprocessor environment. Finally, we would like to extend this work to heterogeneous multiprocessor architectures.

Acknowledgments We acknowledge the support of the National Science Foundation through grants 0093082, 0073800, and 0121706.

6. REFERENCES

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

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

[3] Blue Wave Systems. http://www.bluews.com/

[4] R. Bordawekar, A. Choudhary, and J. Ramanujam. Automatic optimization of communication in compiling out-of-core stencil codes. In Proc. *10th ACM International Conference on Supercomputing*, 1996.

[5] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom memory management methodology.* Kluwer, 1998.

[6] A. Chandrakasan, W. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*. IEEE, 2001.

[7] Dinero IV Trace-Driven Uniprocessor Cache Simulator. URL: http://www.cs.wisc.edu/∼markhill/DineroIV/

[8] R. Gonzales and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, pages 1277–1284, Sep. 1996

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

[10] *Mobile Computing Devices: A New Era in Personal Computing*, August 2000. Computer Market Dynamics.

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

[12] W-T. Shiue and C. Chakrabarti. Memory exploration for low power, embedded systems. In Proc. *36th Design Automation Conference (DAC'99)*, 1999.

[13] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. Y. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In Proc. *Int. Symp. Computer Architecture*, 2000.

[14] L. Wang, W. Tembe, and S. Pande. Optimizing on-chip memory usage through loop restructuring for embedded processors. In Proc. *9th International Conference on Compiler Construction*, 2000.

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

[16] Y. Zhang, Y. Chen, W. Ye, and M. J. Irwin. System-level interconnect power modeling. In Proc. *the 11th International ASIC Conference*, 1998.