Compiler-Directed Scratch Pad Memory Optimization for Embedded Multiprocessors

Mahmut Kandemir, *Member, IEEE*, Ismail Kadayif, *Member, IEEE*, Alok Choudhary, *Senior Member, IEEE*, J. Ramanujam, *Member, IEEE*, and Ibrahim Kolcu

***Abstract*—This paper presents 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 at reducing extra off-chip memory accesses caused by interprocessor communication. This is achieved by increasing the application-wide reuse of data that resides in scratch-pad memories of processors. Our results obtained using four array-intensive image processing applications indicate that exploiting interprocessor data sharing can reduce energy-delay product significantly on a four-processor embedded system.**

***Index Terms*—Embedded systems, loop-dominated applications, scratch-pad memories (SPM).**

I. Introduction

As MICROPROCESSORS grow more and more powerful, designers are building larger and ever more sophisticated systems to solve complex problems. As a result, embedded system designers are now using multiple processors in a single system (either in the form of system-on-a-chip (SoC) or in the form of a multiprocessor board) to address the computational requirements of a wide variety of applications. In this paper, we present a compiler-based strategy for optimizing energy consumption 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 data arrays in parallel. This is in contrast with previous work on scratch-pad memories (SPM), that exclusively focused on single-processor architectures. In this paper, we make the following contributions.

1) We show that interprocessor communication requirements in an embedded multiprocessor system can lead to extra off-chip memory requests, and present a compiler strategy that eliminates these extra off-chip memory requests. Our optimization strategy is fully implemented using an experimental compiler infrastructure and targets array-dominated embedded applications.
2) We report experimental data showing the effectiveness of our optimization strategy. The results show that exploiting interprocessor data sharing can reduce energy-delay product by as much as 33.8% (and 24.3% on average) on a four-processor embedded system.

Manuscript received March 7, 2003; revised June 25, 2003. This work was supported in part by National Science Foundation CAREER Award 0093082.

M. Kandemir and I. Kadayif are with The Pennsylvania State University, University Park, PA 16802 USA (e-mail: kandemir@cse.psu.edu).

A. Choudhary is with Northwestern University, Evanston, IL 60208-2300 USA.

J. Ramanujam is with Louisiana State University, Baton Rouge, LA 70803-5901 USA.

I. Kolcu is with the University of Manchester Institute of Science and Technology, Manchester M60 1QD, U.K.

Digital Object Identifier 10.1109/TVLSI.2004.824299

II. Architecture and Execution Model

A virtually shared scratch pad memory (VS-SPM), 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 Fig. 1. In this architecture, we have a SoC and an off-chip DRAM (which can hold data as well as instructions). SoC has multiple processor cores (with their local SPMs), an interprocessor communication/synchronization mechanism, a clock circuitry, and some ASIC. The SPMs of individual processors make up a VS-SPM. Each processor has fast access to its own SPM (called the local SPM) as well as to the 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 made possible using fast on-chip communication links between processors. Accessing off-chip DRAM, however, is very costly in terms of both latency and energy consumption. 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.

References [1], [2], and [14] discuss dynamic-data reuse techniques for software-controlled cache hierarchies. Steinke *et al.* [13] focus on a strategy for placing program and data objects into SPM for saving energy. Panda *et al.* [11] present a powerful static data-partitioning scheme for efficient utilization of SPM. Their approach is oriented toward eliminating the potential conflict misses due to limited associativity of on-chip cache. This approach benefits applications with a number of small (and highly reused) arrays that can fit in the SPM. Benini *et al.* [5] discuss an elegant memory-management scheme that is based on keeping the most frequently used data items in a software-managed memory (instead of a conventional cache). Kandemir *et al.* [8] propose a dynamic SPM management scheme for data accesses. Their framework uses both loop and data transformations to maximize the reuse of data elements stored in the SPM. They enhance their approach in [9]. Hallnor and Reinhardt [6] propose a new software-managed cache architecture and a new data replacement algorithm. In contrast to these studies, our work focuses on improving the energy behavior on a multiprocessor environment. Therefore, our SPM management strategy is entirely different from prior work.

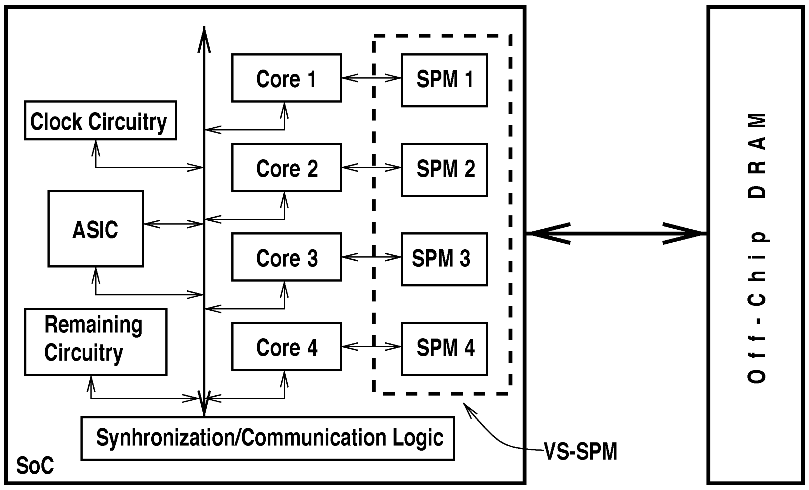

Fig. 1. VS-SPM based system architecture. Each processor has an SPM, and a processor can access its local SPM and the SPM of another processor (i.e., a remote SPM). As compared to off-chip accesses, the SPM accesses (whether local or remote) are much less expensive from an energy consumption perspective.

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 are 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 (also called data tiles) and operates on one chunk at a time. When a data tile has been processed, it is either discarded or written back into off-chip memory (if modified).

III. Problem Description and Proposed Solution

A. Problem Description

In order to improve the reuse of data in the VS-SPM, one can consider intraprocessor data reuse and interprocessor data reuse. Intraprocessor data reuse corresponds to optimizing data reuse when considering access pattern of each processor in isolation. Previous work presented in [8] and [11] addresses this problem. It should be noted, however, that exploiting intraprocessor reuse only may not be very effective in a VS-SPM based environment. This is because intraprocessor data reuse has a local (processor-centric) perspective and does not take interprocessor data sharing (communication) 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. Interprocessor 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.

To illustrate the difference between application-centric and processor-centric views, let us consider the code fragment in Fig. 2, which performs Jacobi iteration over two $N \times N$ square

```
$parfor(i = 2; i \leq (N-1); i++)$
  $parfor(j = 2; j \leq (N-1); j++)$
    $U_2[i][j] += f(U_1[i][j-1] + U_1[i-1][j] + U_1[i][j+1] + U_1[i+1][j]);$
```

Fig. 2. Jacobi iteration. This loop is fully parallel, thanks to lack of data dependences between loop iterations. Consequently, each on-chip processor can execute its iterations in any order. Our strategy exploits this observation by reordering data tile fetches, which, in turn, reduces the total number of off-chip requests due to interprocessor data sharing.

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. Since this loop does not have any data dependences, both the for-loops are parallel. Also, since our approach works on an already parallelized program, we do not concern ourselves with the question of how the code has been parallelized. Assuming that we have four processors (P_1, P_2, P_3, and P_4) as shown in Fig. 1, the portion of array U_2 accessed by each processor is shown in Fig. 3(b). Each processor is responsible from updating an $(N/2) \times (N/2)$ subarray of U_2. The portions accessed by processor 1 from array U_1 are shown in Fig. 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. These elements are called nonlocal 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 subarray and a data tile of its U_2 subarray from DRAM to VS-SPM, updates the corresponding elements, and stores the U_2 data tile back in the DRAM. Fig. 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. Other processors also operate on similar data tiles.

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 other processors as well. This processor first brings a data tile from U_2 and a data tile from U_1 from off-chip memory to its local SPM. After computing the new values for its U_2 data tile, it stores this data tile back in off-chip memory and proceeds by bringing new data tiles from U_1 and U_2. However, to exploit data reuse, it keeps the last row of the previous U_1 data tile in SPM. This is because

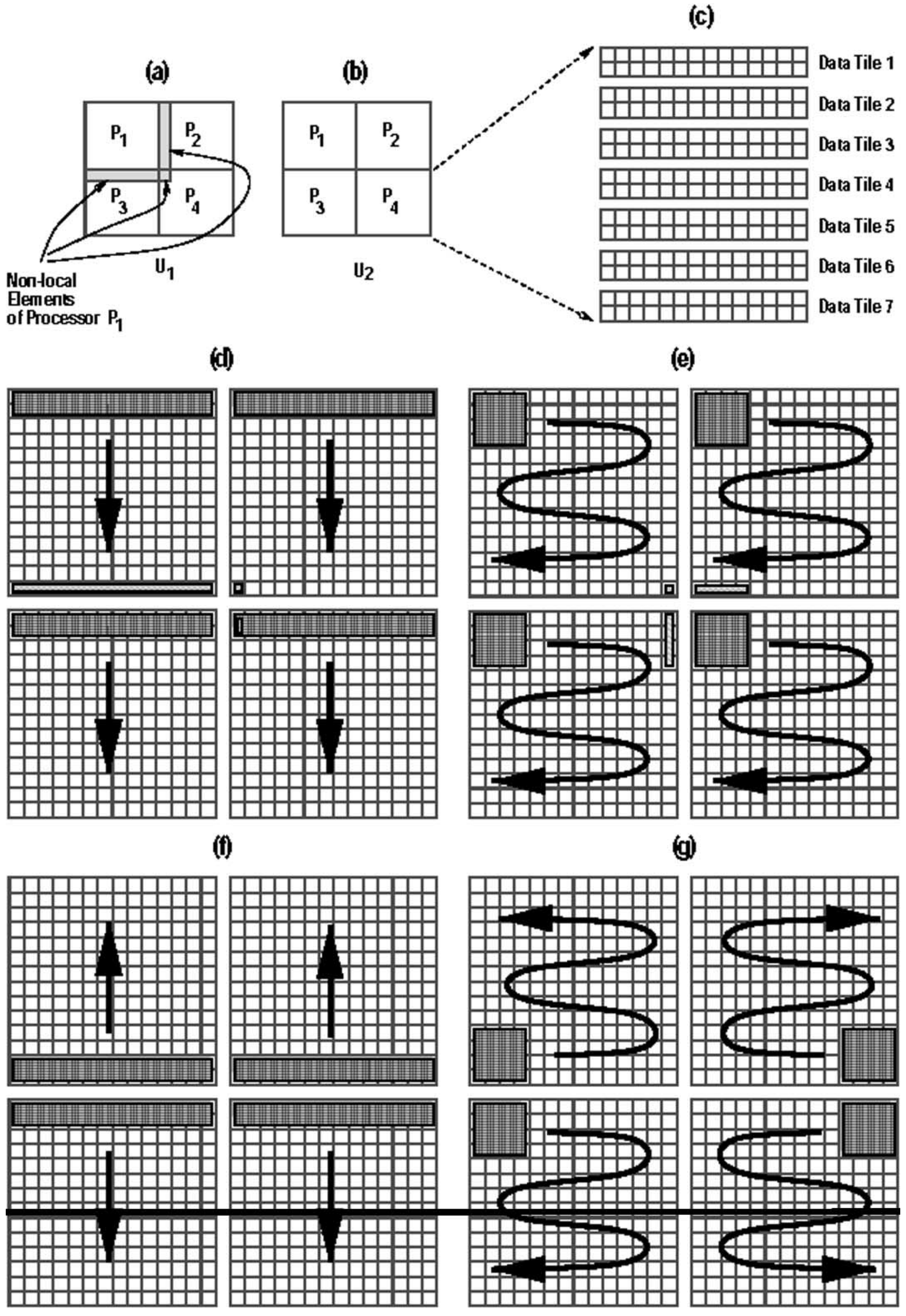

Fig. 3. (a) and (b) Local array portions of processors for the arrays accessed in Fig. 2. (c) Data tiles to be processed by a processor. (d) and (g) Different tile access patterns (arrows denote the order of tile accesses). The important point to note here is that the number off-chip memory accesses varies depending on how data tiles are visited by different processors. Specifically, while the tile access patterns in (d) and (e) incur extra off-chip memory accesses (coming from interprocessor communication), the corresponding access patterns in (f) and (g) eliminate these extra off-chip accesses. When there is no data dependence in the loop nest (as in the case of this example), the loop iterations can be executed in any order (i.e., the data tiles can be traversed in any order). However, when there is a data dependence, some tile access patterns may not be permissible.

this row is also needed when computing the elements of the new U_2 tile. This optimization is termed as local buffering or processor-centric data reuse.

While such a tiling strategy makes effective use of the SPM space as far as intraprocessor reuse is concerned, it fails to capture interprocessor data reuse. To show why it is so, let us consider how our four processors execute this nest in parallel. Fig. 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 nonlocal elements are also shown in Fig. 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 interprocessor communication requirements only) can be very costly from an energy consumption viewpoint. A similar scenario occurs when we consider a different data tile shape. Fig. 3(e) shows processor access patterns (to array U_1) when square data tiles are used. It also shows the nonlocal 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 nonlocal 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.

We now discuss our compiler-based solution using our current example. Since the code fragment in Fig. 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 a nonlocal array element is required (to perform some computation), it can be found it some remote SPM. If this can be achieved for all nonlocal accesses, we can eliminate all extra off-chip memory accesses due to interprocessor communication. Fig. 3(f) and (g) show how extra off-chip memory requests can be eliminated when row block and square data tiles are used, respectively.

There are at least two subproblems for compiling array-dominated applications in 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 maximum size is determined by the local SPM size).
- *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 interprocessor communication) are eliminated.

We define a tile access pattern matrix (scheduling matrix), denoted $\mathcal{H}$, which determines the order in which the data tiles are accessed. Our scheduling strategy makes use of the following lemma.

Lemma 1: Consider two processors i and j with their scheduling matrices $\mathcal{H}_i$ and $\mathcal{H}_j$, respectively. In a two-dimensional (2-D) array case, schedules denoted by these matrices eliminate extra off-chip memory accesses (when considering only these two processors) if and only if they satisfy the equality

$$\mathcal{H}_i^T \vec{v}_{i,j} = \mathcal{H}_j^T \vec{v}_{j,i}.$$

This equality is called the scheduling equality, and $\vec{v}_{i,j}$ is a vector that represents the direction of processor i with respect to processor j. Our scheduling algorithm consists of the following 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 (based on Lemma 1 given above). These equalities collectively represent the constraints that need to be satisfied for eliminating all extra DRAM accesses due to interprocessor communication.
3) Initialize the scheduling matrix of a processor with an arbitrary schedule and compute the corresponding scheduling matrices (tile access patterns) of the remaining processors by solving the scheduling equalities.

As an example, let us consider the successive-over-relaxation (SOR) loop shown in Fig. 4. If we apply our three-step strategy, one possible schedule would have the following scheduling matrices (assuming four processors)

$$\mathcal{H}_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \mathcal{H}_2 = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$$
$$\mathcal{H}_3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \quad \mathcal{H}_4 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

This is shown in Fig. 5(a). Since the schedules for processors 2 and 4 violate data dependences, they are not acceptable. An-

```
for(i = 2; i ≤ (N − 1); i++)
 parfor(j = 1; j ≤ (N − 1); j++)
  U1[i][j]+ = g(U1[i − 1][j] + U1[i][j + 1] + U1[i + 1][j]);
```

Fig. 4. SOR iteration. This is similar to the Jacobi iteration code given earlier; the difference is that since the same array is both read and written, we have data dependences. These dependences prevent the compiler from using some potential tile access patterns.

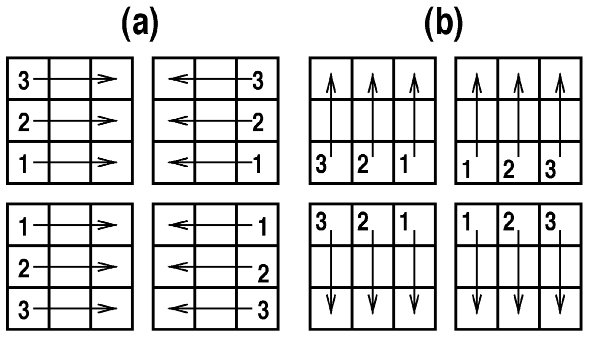

Fig. 5. (a) Illegal and (b) legal access patterns for the code fragment in Fig. 4. The legality of a tile access pattern is checked by considering the data dependences in the code.

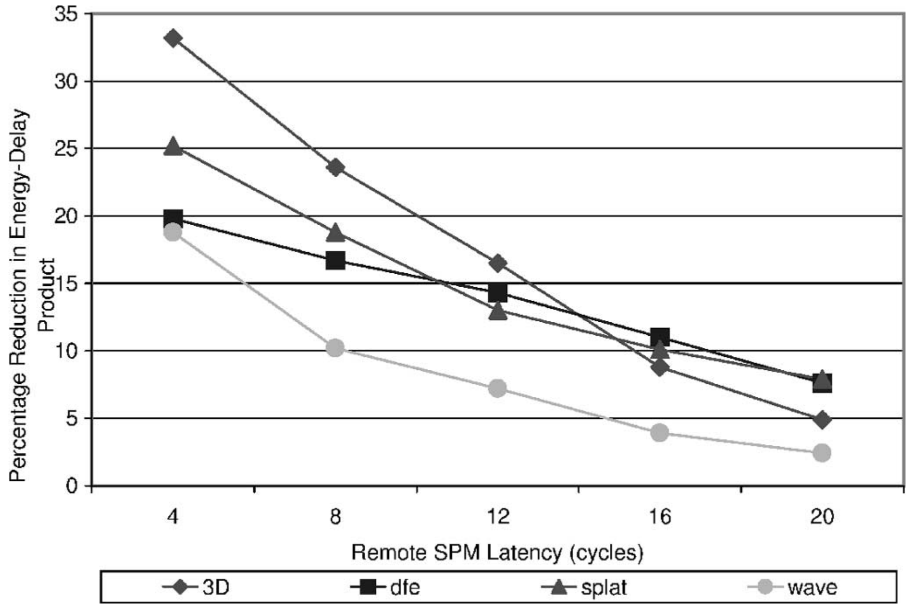

Fig. 6. Percentage savings in energy-delay product with different remote SPM latencies. We observe that the average (i.e., across all benchmarks) improvements when remote SPM latency is four cycles and 20 cycles are 24.3% and 5.7%, respectively.

other schedule (solution) would have the following scheduling matrices:

$$\mathcal{H}_1 = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \mathcal{H}_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$
$$\mathcal{H}_3 = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}, \quad \mathcal{H}_4 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

The tile access pattern corresponding to this schedule is given in Fig. 5(b). We note that this is a legal schedule. This is because the flow dependence in this code is carried by the i loop; and consequently, the tiles can be accessed in the j loop direction in any order (as long as the proper order in the i direction is maintained). In should be noted, however, that in the existence of flow dependences, it may not always be possible to eliminate all extra off-chip memory accesses. Note that our approach tries to come up with appropriate (and legal) scheduling matrices if it is possible to do so. However, it is also possible to check whether a given scheduling matrix is legal.

IV. Experiments

A. Experimental Setup and Benchmarks

Our experimental setup consists of a compiler environment and an inhouse simulator. Our optimization algorithm has been implemented using the Stanford University Intermediate Format (SUIF) experimental compiler infrastructure [3]. 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 programs are written so that they can operate on images of different sizes. Our simulator takes a parallel code written in C as input and simulates a multiprocessor architecture. Each simulated processor is a 100-MHz MIPS 4 Kp-like core with a five-stage pipeline that supports four execution units (integer, multiply-divide, branch control, and processor control). The default values of our simulation parameters are as follows. The simulated off-chip DRAM is 4 MB with an access latency of 80 cycles. On-chip multiprocessor has four processors, each with an SPM. Accessing the local SPM takes two cycles. We also define a parameter called slab ratio, which gives the ratio between the local SPM size and total array size that needs to be traversed by a processor. The default value for this parameter is 1/8. That is, the available SPM space is 1/8 of the local array portion of a processor. Finally, we use row block data tiles.

B. Results

Fig. 6 gives the energy-delay product improvements for different values of the remote SPM access latency with the default simulation parameters. It should be emphasized 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 eliminating the extra off-chip memory accesses (due to interprocessor communication requirements) helps to improve energy-delay product in all benchmarks and with all remote SPM latencies. Also, as expected, our approach generates the best savings with the small (remote SPM) latencies. However, even with large latencies, we have at least 5% savings in all benchmarks except `wave`. The average improvements when remote SPM latency is 4 and 20 are 24.3% and 5.7%, respectively. It should also

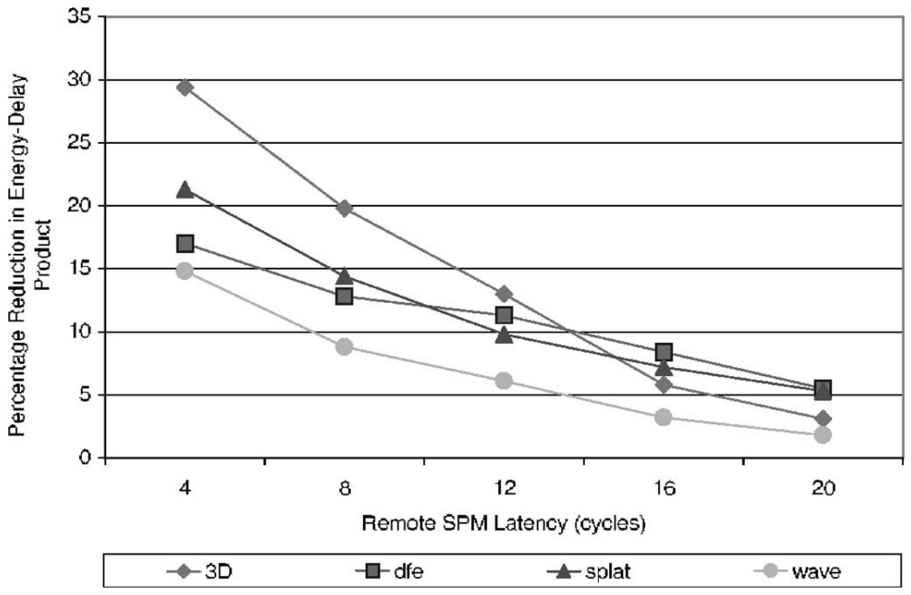

Fig. 7. Percentage savings in energy-delay product with different remote SPM latencies when the applications are optimized to minimize interprocessor data sharing. We observe that the average improvements when remote SPM latency is 4 and 20 are 20.6% and 3.9%, respectively.

be mentioned that the energy benefits brought by our strategy range from 1.4% (with `wave` with an SPM latency of 20 cycles) to 22.4% (with `3D` with a remote SPM latency of four cycles). The average energy savings when remote SPM latency is 4 and 20 are 16.1% and 3.1%, respectively. While these values are not as good as our energy-delay improvements, they are still significant.

To obtain the results in Fig. 6, we used the original benchmarks without any modification. The only modification introduced by our optimization strategy was to reorder the data tile accesses to reduce the number of extra off-chip memory requests. However, it should be mentioned that even in this case most of the loops in our applications, 67.7% to be specific, exhibited outermost loop parallelism (that is, we were able to parallelize the outermost loop in the nest without any code modification). In our next set of experiments, we measured the impact of code optimizations on the effectiveness of our strategy. Specifically, we optimized the input codes using a set of source-level optimizations that reduce the number of array elements shared by multiple processors. We achieved this as follows. Since our loop parallelization strategy is oriented toward parallelizing the outer loops, we tried to place all data dependences into the innermost loop positions. While it is possible to do this using a variety of compiler-based techniques designed for exploiting loop-level parallelism [4], [10], in this work, we employed the compiler algorithm outlined in [7]. Basically, this algorithm uses both loop and data transformations to bring as much data reuse as possible into the innermost loop positions. After this optimization, we collected experimental data with and without our VS-SMP optimization strategy. The results are given in Fig. 7. Comparing these results with those shown in Fig. 6, we see that the effectiveness our approach slightly reduces. However, we still achieve 20.6% and 3.9% savings in the energy-delay product with remote SPM latencies of four cycles and 20 cycles, respectively.

V. Conclusions

This paper presents a compiler-directed optimization strategy for exploiting software-controlled, shared SRAM space in an embedded multiprocessor system. Our approach is oriented toward eliminating extra off-chip DRAM accesses caused by interprocessor communication (data sharing). The results obtained using four array-intensive applications show that significant reductions in energy-delay product are possible using this approach.

References

[1] T. Van Achteren, R. Lauwereins, and F. Catthoor, “Systematic data reuse exploration methodology for irregular access patterns,” in *Proc. 13th ACM/IEEE Symp. System-Level Synthesis*, Madrid, Spain, Sept. 2000, pp. 115–121.

[2] ——, “Data reuse exploration techniques for loop-dominated applications,” in *Proc. 5th ACM/IEEE Design Test Europe Conf.*, Paris, France, Apr. 2002, pp. 428–435.

[3] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng, “The SUIF compiler for scalable parallel machines,” in *Proc. 7th SIAM Conf. Parallel Processing Scientific Computing*, Feb. 1995.

[4] J. M. Anderson, “Automatic computation and data decomposition for multiprocessors,” Ph.D. dissertation, Computer Systems Lab., Stanford Univ., Stanford, CA, Mar. 1997.

[5] L. Benini, A. Macii, E. Macii, and M. Poncino, “Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation,” *IEEE Des. Test Comput.*, pp. 74–85, Apr.–June 2000.

[6] E. G. Hallnor and S. K. Reinhardt, “A fully-associative software-managed cache design,” in *Proc. Int. Conf. Computer Architecture*, Vancouver, BC, Canada, 2000, pp. 107–116.

[7] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee, “Improving locality using loop and data transformations in an integrated framework,” in *Proc. Int. Symp. Microarchitecture*, Dallas, TX, Dec. 1998, pp. 285–296.

[8] 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 Conf.*, Las Vegas, NV, June 2001, pp. 690–695.

[9] M. Kandemir and A. Choudhary, “Compiler-directed scratch pad memory hierarchy design and management,” in *Proc. 39th Design Automation Conf.*, June 2002, pp. 628–633.

[10] A. W. Lim, G. I. Cheong, and M. S. Lam, “An affine partitioning algorithm to maximize parallelism and minimize communication,” in *Proc. 13th ACM SIGARCH Int. Conf. Supercomputing*, June 1999, pp. 228–237.
[11] P. R. Panda, N. D. Dutt, and A. Nicolau, “Efficient utilization of scratch-pad-memory in embedded processor applications,” in *Proc. Eur.Design Test Conf.*, Paris, France, Mar. 1997, pp. 7–11.
[12] ——, “Architectural exploration and optimization of local memory in embedded systems,” in *Proc. Int. Symp. System Synthesis*, Antwerp, Belgium, Sept. 1997, pp. 90–97.
[13] S. Steinke *et al.*, “Assigning program and data objects to scratchpad for energy reduction,” in *Proc. Eur. Design Test Conf*, Paris, France, 2002, pp. 1–7.
[14] S. Wuytack, J. P. Diguet, F. Catthoor, and H. De Man, “A formalized methodology for data reuse exploration for low-power hierarchical memory mappings,” *IEEE Trans. VLSI Syst.*, vol. 6, pp. 529–537, Dec. 1998.

Mahmut Kandemir (M’98) received the B.Sc. and M.Sc. degrees in control and computer engineering from Istanbul Technical University, Istanbul, Turkey, in 1988 and 1992, respectively, and the Ph.D. degree in electrical engineering and computer science, from Syracuse University, Syracuse, NY, in 1999.

Since August 1999, he has been an Assistant Professor in the Computer Science and Engineering Department, Pennsylvania State University, University Park. His main research interests include optimizing compilers, I/O intensive applications, and power-aware computing.

Dr. Kandemir is a Member of the ACM.

Ismail Kadayif (M’01) received the B.Sc degree, from the Department of Control and Computer Engineering, Istanbul Technical University, Istanbul, Turkey, in 1991, and the M.Sc degree from the Department of Computer Science, Illinois Institute of Technology, Chicago, in 1997. He is currently working toward the Ph.D. degree in computer science and engineering, Pennsylvania State University, University Park.

His research interests include high-level compiler optimizations, power-aware compilation techniques, and low-power computer architectures.

Alok Choudhary (SM’99) received the B.E. degree (Hons.) from Birla Institute of Technology and Science, Pilani, India, in 1982, the M.S. degree from the University of Massachusetts, Amherst, in 1986, and the Ph.D. degree in electrical and computer engineering, from the University of Illinois, Urbana-Champaign, in 1989.

His main research interests include high-performance computing and communication systems and their applications in many domains including multimedia systems, information processing, and scientific computing. In particular, his interests are in the design and evaluation of architectures and software systems (from system software such as runtime systems, compilers, and programming languages to applications), high-performance servers, high-performance databases and input–output.

Dr. Chaudhary is a Member of the ACM.

J. (Ram) Ramanujam (M’94) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Madras, in 1983, and the M.S. and Ph.D. degrees in computer science from The Ohio State University, Columbus, in 1987 and 1990, respectively.

He is currently an Associate Professor in the Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge. His research interests include embedded systems, compilers for high-performance computer systems, software optimizations for low-power computing, high-level hardware synthesis, parallel architectures, and algorithms.

Ibrahim Kolcu received the B.Sc. degree in computer engineering from Istanbul Technical University (ITU), Istanbul, Turkey, in 1990, and the M.Phil degree in genetic algorithms from Cranfield University, Cranfield, U.K., in 1997. He is currently working toward the Ph.D. degree, University of Manchester Institute of Science and Technology (UMIST), Manchester, U.K.

His research interests include genetic algorithms, neural networks, code optimization, search algorithms, and low-power computing.