A Compiler-Based Approach for Dynamically Managing Scratch-Pad Memories in Embedded Systems

Mahmut Kandemir, *Associate Member, IEEE*, J. Ramanujam, *Member, IEEE*, Mary Jane Irwin, *Fellow, IEEE*, N. Vijaykrishnan, *Associate Member, IEEE*, Ismail Kadayif, and Amisha Parikh

***Abstract*—Optimizations aimed at improving the efficiency of on-chip memories in embedded systems are extremely important. Using a suitable combination of program transformations and memory design space exploration aimed at enhancing data locality enables significant reductions in effective memory access latencies. While numerous compiler optimizations have been proposed to improve cache performance, there are relatively few techniques that focus on software-managed on-chip memories. It is well-known that software-managed memories are important in real-time embedded environments with hard deadlines as they allow one to accurately predict the amount of time a given code segment will take. In this paper, we propose and evaluate a compiler-controlled dynamic on-chip scratch-pad memory (SPM) management framework. Our framework includes an optimization suite that uses loop and data transformations, an on-chip memory partitioning step, and a code-rewriting phase that collectively transform an input code automatically to take advantage of the on-chip SPM. Compared with previous work, the proposed scheme is dynamic, and allows the contents of the SPM to change during the course of execution, depending on the changes in the data access pattern. Experimental results from our implementation using a source-to-source translator and a generic cost model indicate significant reductions in data transfer activity between the SPM and off-chip memory.**

***Index Terms*—Compiler optimizations, dynamic memory management, embedded systems design, scratch-pad memories (SPM), systems-on-chip.**

Manuscript received July 5, 2001; revised April 29, 2003. A preliminary version of this paper appeared in the *Proceedings of the 38th Design Automation Conference* (DAC'01) Las Vegas, NV, June 2001, pp. 690-695. This work was supported in part by the Pittsburgh Digital Greenhouse through a grant from the Department of Community and Economic Development, Commonwealth of Pennsylvania, and in part by the National Science Foundation (NSF) under CAREER Awards 0093082 and 0093085. The work of J. Ramanujam was supported in part by the NSF under Grant CCR-0073800 and in part by the NSF under Young Investigator Award CCR-9457768. This paper was recommended by Associate Editor R. Gupta.

M. Kandemir is with the Department of Computer Science and Engineering, and the Microsystems Design Laboratory, The Pennsylvania State University, University Park, PA 16802 USA (e-mail: kandemir@cse.psu.edu).

J. Ramanujam is with the Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803 USA (e-mail: jxr@ee.lsu.edu).

M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh are with the Microsystems Design Laboratory, The Pennsylvania State University, University Park, PA 16802 USA (e-mail: mji@cse.psu.edu; vijay@cse.psu.edu; kadayif@cse.psu.edu; aparikh@cse.psu.edu).

Digital Object Identifier 10.1109/TCAD.2003.822123

I. Introduction

EMBEDDED systems are systems designed to execute one or a small set of dedicated application(s), e.g., a multimedia subsystem, a personal digital assistant, a hand-held computer, etc. Such systems provide an important substrate for information technology research since they form key elements of information appliances. They are often designed as a system-on-chip (SoC) architecture to cater to the demands of small form factors. The requirement of fast design cycle times fuels the need for a high degree of design reuse, in both hardware and software. Modern design libraries consisting of processor cores (e.g., the TI TMS320, Strong ARM from Advanced RISC Machines, SH5 from Hitachi), memories (e.g., embedded DRAMs, scratch-pad memories, cache memories), and modules implementing specific functions (e.g., ASICs) are now available. An SoC is designed by integrating such predesigned cells along with custom logic on a single chip.

An important characteristic of an SoC design process is the design of memory configuration and the management of the flow of data. While it is very important to select a correct memory configuration, it is equally important to choreograph the flow of data between on- and off-chip memories in an optimal manner.

Many SoC applications have significant data processing requirements. In particular, many codes from video processing and signal-processing domains manipulate large arrays of signals using multilevel nested loops. An important issue, then, is maintaining good data locality; that is, satisfying a majority of data accesses from fast on-chip memories instead of the slower off-chip DRAMs. Previous research [43] shows that ensuring good data locality requires exploiting inherent data reuse within loop nests.

Unfortunately, a simple on-chip cache hierarchy may not be very suitable for an embedded system, where meeting hard real-time constraints is critical [21], [22]. In most cases, such a constraint requires programmers to determine exactly how much processing time a given code segment will take. The presence of a cache memory (among some other architectural features, such as branch prediction and load/store speculation [20]), makes it nearly impossible to predict execution-time accurately. While it is possible for the programmer to determine the worst-case execution time (by assuming that each cache access will be a miss), there might be orders of magnitude difference between the actual execution time and this predicted worst-case behavior. Note that an inability to predict the execution time of a given piece of code can also adversely affect the scope and effectiveness of software optimizations.

Consequently, systems that contain a software-managed scratch-pad memory (SPM) can be of great value, since the software is in full control of the flow of data between the on- and off-chip memory in a such a system, thereby enabling an accurate prediction of data access times in real-time environments. While execution time predictability is one of the reasons for employing an SPM, it is not the only reason. Since the application code/compiler is in full control of the data flow in and out of the SPM, we can decide what should be in the SPM at any given time. In fact, it is possible to pin the most frequently used data (during execution of a given loop nest) in the SPM to maximize performance benefits. Note that, in general, it is not possible to do the same with a cache memory as, for example, conflict misses can cause the useful data to be replaced from the cache. In particular, for array-dominated embedded applications with regular data-access patterns, we can achieve a near-optimal managements of an SPM.

Previous work on SPM [37] investigates a static data-management scheme in which program data structures are partitioned between the off-chip memory and the SPM, and this data partitioning is maintained throughout execution. In this static partitioning scheme, the scalar variables are stored in the SPM, and the large data arrays that do not fit in the SPM are stored in the off-chip memory (and accessed through on-chip cache). Each of the remaining arrays is stored in either the SPM or the off-chip memory so as to minimize the potential conflict misses in the on-chip cache.

While several applications benefit from this static partitioning approach, we have come across many codes for which we need to perform dynamic data transfers between the off-chip memory and on-chip SPM during the course of execution. As an example, consider the matrix multiply code shown in Fig. 4(i) on page 9. Assuming that the capacity of the SPM is smaller than the aggregate size of the three arrays involved in this computation, there are three issues that need to be addressed for effective use of the SPM.

First, we need to decide which portions of the arrays should be brought into the SPM at a given time. This decision is critical because the elements brought in should have a high degree of reuse; otherwise, they would occupy space in the SPM unnecessarily and waste bandwidth. Consequently, for a given multidimensional array, it makes a great difference whether a row-chunk, a column-chunk or a square-chunk is brought into the SPM as far as data reuse is concerned.

Second, we need to divide the available SPM space among competing arrays. In this paper, we show that a simple strategy that divides the memory evenly between competing arrays may not work well in practice; that is, some arrays must be given more space in SPM than others.

Third, we need to modify the input code to schedule explicit data transfers to and from off-chip memory. We discuss a variant of iteration space tiling, a popular locality optimization technique to achieve this objective.

This paper addresses these issues in the context of SPM and array-dominated applications, which are common in the video signal processing domain. The proposed compilation framework has been implemented using SUIF, an experimental compiler [4], and tested on a suite of five applications. Experimental results and comparison with previous work show that our approach is very effective in reducing the activity between the on-chip SPM and off-chip memory. In general, such a reduction can lead to large savings in energy consumption [41], [25] and effective data-access latency [24].

II. Roadmap

We start in Section III, with our memory architecture and execution model. Our memory system is different from traditional cache-based systems as it contains an SPM. Since SPM is managed by software, our execution model is also different from that of a cache-based system. In particular, in a cache-based system, the software is not involved in data transfers between off-chip and on-chip memory. In an SPM-based scheme, on the other hand, the software (in our case, compiler) needs to schedule data transfers between SPM and main memory. It achieves this by using a data-oriented version of tiling as explained in Section III.

Section IV presents optimizations performed by our compiler to extract the best performance from SPM. First, in Section IV-B, we give the model used for calculating the cost of a given (access pattern, memory layout) pair. This is important to evaluate the performance of a given code under SPM. Section IV-C presents the desired access patterns, which will be the target of the optimizations presented in later subsections. Our optimization algorithm (working on a single nest) has three main steps. In the first step (Section IV-D), we use linear algebraic techniques to determine the most suitable memory layouts and loop transformations from the data locality perspective. After determining the transformations, our compiler partitions the available SPM space among competing array references as explained in Section IV-E. This is important because, given a nested loop, there might be multiple references (to different arrays) that exhibit locality and can benefit if the corresponding data tiles are stored in SPM. The next step deals with placement of data transfers in appropriate places in the code and is explained in Section IV-F. Note that suitable placement of transfers (in the code) is very important, as a suboptimal transfer can increase the traffic between SPM and main memory dramatically, thereby degrading the overall performance.

The approach explained in Sections IV-A through IV-F focuses on a single nest case. It should be noted that our optimization strategy uses memory-layout transformations. Since a given array can be accessed by multiple nests and each nest can demand a different memory layout for a given array, selecting a globally (program-wide) acceptable memory layout for each array is critical. To address this problem, in Section IV-G, we show how our strategy can be used (as a component) in optimizing multiple nests and how it can be extended to an interprocedural setting. Note that many previous locality optimizations (that target cache-based systems) lack interprocedural optimizations.

The remaining sections discuss experimental setup and results (Section V) and related work on software-directed data

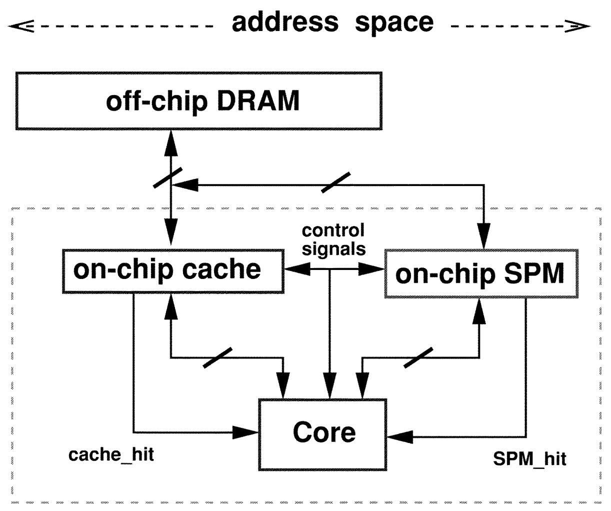

Fig. 1. System architecture and address space partitioning.

flow management (Section VI). Finally, Section VII concludes the paper with a summary and a brief discussion of ongoing work.

III. Memory Architecture and Execution Model

Our data memory architecture consists of three components: a cache memory, an SPM, and a main memory. The cache memory and the SPM are on-chip SRAMs (with the same access latency), and the main memory can be assumed to be an off-chip DRAM (with a higher access latency). As shown in Fig. 1, the address space is divided between the on-chip SPM and the off-chip memory (which is accessed through the on-chip cache).

Fig. 1 also shows the necessary control signals to activate either SPM or cache depending on the memory address issued by the core. The same data and address buses are fed into cache as well as the SPM. In this work, we are interested in managing the data transfers between the off-chip memory and the on-chip SPM in those situations where the total data size far exceeds the capacity of the SPM. Therefore, we assume the presence of a higher level mechanism that decides which data accesses should bypass the SPM and be accessed through the cache (if it exists). In that sense, our approach is complementary to the technique proposed by Panda *et al.* [37] that focuses on data partitioning between the off-chip memory and the SPM. We are more interested in dynamic management of the flow of data between the SPM and off-chip memory. Therefore, in the rest of the paper, we drop the cache from consideration. It can be assumed that the scalar variables in the code are stored in DRAM and accessed through on-chip cache. Alternately, a small portion of the SPM can be reserved for scalars throughout execution.

The execution model adopted can be summarized as follows. The current computation operates only on the datasets available in the SPM.[1] If the computation needs a data item that resides in off-chip DRAM, the item is first read (brought) into the SPM, and then operated on. The data items modified while they are in the SPM should be written back to memory if: 1) the corresponding SPM space needs to be reused and 2) the said data items will be needed in the remainder of the computation. It should be noted that, in such a system, the on-chip SPM space is at a premium and, therefore, it is important to make effective use of the on-chip SPM for high levels of performance.

In order to generate optimized code for a memory architecture that contains an SPM, the compiler has to schedule explicit data transfers between the off-chip memory and the SPM, in addition to performing the conventional optimization steps. To accomplish this, the compiler needs to take into account: 1) the data layout in the off-chip memory; 2) the application access pattern; and 3) the available memory space in the SPM. As mentioned earlier, the portions of arrays required by the currently executing computation are fetched from the off-chip memory to the SPM. In this paper, we refer to these portions as data tiles or simply tiles. The larger the tiles the better it is, since working with large tiles reduces the number of off-chip accesses. However, the data tiles brought into the SPM should fit in the SPM, and exhibit a high degree of reuse. Note that the SPM space should be divided suitably among the data tiles of different arrays. Thus, during the course of execution, a number of data tiles belonging to a number of different arrays are brought into the SPM, the new values for these data tiles are computed, and the tiles are stored back into appropriate locations in off-chip memory as needed. Obviously, the input code should be transformed accordingly and explicit data transfer commands should be inserted into the code, in order to achieve this objective.

As an example, let us consider the two-level loop nest shown in Fig. 2(i). A key aspect of the compilation process is the use of a tiling-like transformation. Tiling (also known as blocking) [44], [27], [13], [23] is a technique used to improve locality and parallelism, and is a combination of strip-mining and loop permutation. It creates blocked (sub-matrix) versions of programs in a systematic way. When tiling is applied to a loop, it replaces it (in the most general case) with two loops: a tiling loop and an element loop. The loop nest in Fig. 2(i) can be translated by the compiler into the code shown in Fig. 2(ii). In this translated code, the outer loops `it` and `jt` are the tiling loops, and the inner loops $\mathtt{i}'$ and $\mathtt{j}'$ are the element loops.

It should be noted that explicit data transfer calls (i.e., **read_tile** and **write_tile**) are inserted at tile boundaries outside the element loops. The call **read_tile** $\mathtt{U[it{:}it} + S_a - \mathtt{1,\ jt{:}jt} + S_a - \mathtt{1]} \rightarrow \mathtt{U'[0{:}}S_a - \mathtt{1,0:}\ S_a - \mathtt{1]}$ copies the elements of the array `U` that satisfy the constraint $\{(\mathtt{it} \le \mathtt{i} \le \mathtt{it+T-1})$ and $(\mathtt{jt} \le \mathtt{j} \le \mathtt{jt+T-1}\})$ to the array $\mathtt{U'[0}:S_a - \mathtt{1,0}:S_a - \mathtt{1]}$. It should be noted that while array `U` resides in off-chip memory, array `U'` resides in the SPM. In other words, **read_tile** indicates an explicit copy operation from the off-chip memory to the SPM as depicted in Fig. 3. The implementation of the **write_tile** call is similar except that the direction of the transfer is reversed.

In the rest of this paper, for the sake of clarity, we will write the compiler-translated version as shown in Fig. 2(iii), where all element loops are omitted. Note also that each reference inside the element loops is replaced by its corresponding submatrix version (in terms of the original array). For example, a reference such as `U'[i'][j']`, is replaced by $\mathtt{U[it{:}it}+S_a\mathtt{-1,jt{:}jt}+$

[1]When we performed experiments with static data partitioning, we relaxed this constraint, and allowed the computation to work with off-chip data directly.

```
for(i=0;i<$n$;i++)
  for(j=0;j<$n$;j++)
    U[i][j] = U[i][j] + V[i][j]
```

(i)

```
for(it=0;it<$n$;it=it+$S_a$)
  for(jt=0;jt<$n$;jt=jt+$S_a$)
    {
      read_tile U[it:it+$S_a$-1,jt:jt+$S_a$-1] → U'[0:$S_a$-1,0:$S_a$-1]
      read_tile V[it:it+$S_a$-1,jt:jt+$S_a$-1] → V'[0:$S_a$-1,0:$S_a$-1]
      for(i'=0;i'<$S_a$;i'++)
        for(j'=0;j'<$S_a$;j'++)
          U'[i'][j'] = U'[i'][j'] + V'[i'][j']
      write_tile U'[0:$S_a$-1,[0:$S_a$-1] → U[it:it+$S_a$-1,jt:jt+$S_a$-1]
    }
```

(ii)

```
for(it=0;it<$n$;it=it+$S_a$)
  for(jt=0;jt<$n$;jt=jt+$S_a$)
    {
      read_tile U[it:it+$S_a$-1,jt:jt+$S_a$-1]
      read_tile V[it:it+$S_a$-1,jt:jt+$S_a$-1]
      U[it:it+$S_a$-1,jt:jt+$S_a$-1]=U[it:it+$S_a$-1,jt:jt+$S_a$-1]+V[it:it+$S_a$-1,jt:jt+$S_a$-1]
      write_tile U[it:it+$S_a$-1,jt:jt+$S_a$-1]
    }
```

(iii)

Fig. 2. (i) Two-level nested loop. (ii) Transformed code with explicit data transfers. (iii) Simplified version of (ii).

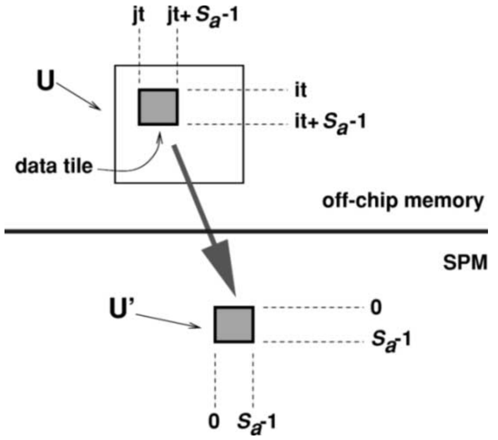

Fig. 3. Reading a data tile (of size $S_a \times S_a$) from off-chip memory to SPM.

$S_a - 1]$, where S_a is the size (in array elements) of a dimension of a data tile (called tile size or blocking factor). Note that the notation in Fig. 2(iii) clearly indicates the portions of the original arrays (tiles) involved in the computation for a given iteration of the tiling loops.

IV. Dynamic Management of Data Transfers

A. Overview

The efficient use of an SPM depends critically on maximizing the reuse of data portions (tiles) brought in from the off-chip memory. This is crucial because a data item resident in the SPM without reuse not only occupies space that could have been used for some other data but also wastes bandwidth (as it needs to be brought over the interconnect between SPM and off-chip memory). Therefore, one should be very careful in selecting the data items to bring into the SPM for a given time frame. This objective can be achieved by: 1) maximizing the number of accesses to the data in the SPM and 2) minimizing the number of data transfers to and from off-chip memory. We will see that data reuse analysis can be of great help here.

In order to maximize data reuse and minimize the number of data transfers, our approach follows three complementary steps.

i) **Memory Layout Detection and Loop Transformation:** Maximizing data reuse and minimizing the number of explicit data transfers between the off-chip memory and the SPM requires a good combination of loop access pattern and off-chip memory layout. Given a program that accesses multiple data arrays using multilevel nested loops, our approach (explained later in this section) tries to determine memory layouts for the arrays and loop transformations for the nests. Optimization algorithms that achieve this for the case of a single nest and for the case of whole programs (including interprocedural analysis) are explained in detail in Sections IV-D and IV-H, respectively.

ii) **Memory Space Partitioning:** The management of the on-chip SPM is important since the SPM is limited in capacity (i.e., size). A crucial issue here is to determine the partitioning of the available SPM space between competing arrays. Note that this space partitioning is dynamic and changes during the course of execution. Our partitioning approach is explained in Section IV-E.

iii) **Code Modifications:** After deciding the optimal loop access pattern, memory layouts and a suitable partitioning of the SPM space between arrays, the compiler

```
for(i=0;i<n;i++)
  for(j=0;j<n;j++)
    for(k=0;k<n;k++)
      U[i][j] = U[i][j] + V[i][k] * W[k][j]
```

(i)

```
for(it=0;it<n;it=it+S_a)
  for(jt=0;jt<n;jt=jt+S_a)
    for(kt=0;kt<n;kt=kt+S_a)
      {
        read_tile U[it:it+S_a-1,jt:jt+S_a-1]
        read_tile V[it:it+S_a-1,kt:kt+S_a-1]
        read_tile W[kt:kt+S_a-1,jt:jt+S_a-1]
          U[it:it+S_a-1,jt:jt+S_a-1]=U[it:it+S_a-1,jt:jt+S_a-1]
          +V[it:it+S_a-1,kt:kt+S_a-1]*W[kt:kt+S_a-1,jt:jt+S_a-1]
        write_tile U[it:it+S_a-1,jt:jt+S_a-1]
      }
```

(ii)

```
for(it=0;it<n;it=it+S_b)
  {
    read_tile V[it:it+S_b-1,1:n]
    for(jt=0;jt<n;jt=jt+S_b)
      {
        read_tile U[it:it+S_b-1,jt:jt+S_b-1]
        read_tile W[1:n,jt:jt+S_b-1]
        U[it:it+S_b-1,jt:jt+S_b-1]=U[it:it+S_b-1,jt:jt+S_b-1]+V[it:it+S_b-1,1:n]*W[1:n,jt:jt+S_b-1]
        write_tile U[it:it+S_b-1,jt:jt+S_b-1]
      }
  }
```

(iii)

```
for(jt=0;jt<n;jt=jt+S_c)
  {
    read_tile U[1:n,jt:jt+S_c-1]
    for(kt=0;kt<n;kt=kt+S_c)
      {
        read_tile V[1:n,kt:kt+S_c-1]
        read_tile W[kt:kt+S_c-1,jt:jt+S_c-1]
        U[1:n,jt:jt+S_c-1]=U[1:n,jt:jt+S_c-1]+V[1:n,kt:kt+S_c-1]*W[kt:kt+S_c-1,jt:jt+S_c-1]
      }
    write_tile U[1:n,jt:jt+S_c-1]
  }
```

(iv)

Fig. 4. (i) Matrix multiply code. (ii) Straightforward translation. (for SPM) using square data tiles for all arrays. (iii-iv) Optimized codes.

needs to modify the input code accordingly. Our approach, discussed in Section IV-F, achieves this without user intervention.

After presenting the generic cost model adopted and the ideal form of array references from the point of view of scratch-pad memories, the following subsections discuss each of these steps in detail. The implementation details and experimental data are presented in a later section.

B. Cost Model

We assume a generic cost model that can be easily adapted to work with several performance and/or energy metrics. Each data transfer from the off-chip memory to the SPM or vice versa is assumed to incur a fixed startup cost in addition to a cost proportional to the amount of data requested. The startup cost for a data access (read or write), C, is assumed to include all the costs due to book-keeping and hand-shaking activity between the SPM and the off-chip memory and the software overhead involved (e.g., the runtime call activation to initiate the transfer). Let the cost of transferring a single data item (e.g., an array element) between the off-chip memory and the SPM be t. Thus, the cost of transferring ℓ consecutive elements between the off-chip memory and the SPM can be modeled as $T = C + \ell t$. We refer to this cost as memory access cost, or simply access cost. Note that the per item transfer cost does not include the cost (e.g., time, energy) spent in accessing the off-chip memory circuitry. Note also that this model is highly simplified, but it is useful for the purpose of this paper.

As an example, let us consider the matrix multiply code given in Fig. 4(i). We assume for now that the SPM allocated for a given computation is of size M, and that this memory is divided evenly (i.e., equally) among all the arrays involved in the computation; we will relax this "equal division" assumption shortly. Let us also assume (for simplicity) that each array is of size $n \times n$, where n is also assumed to be the trip count (number

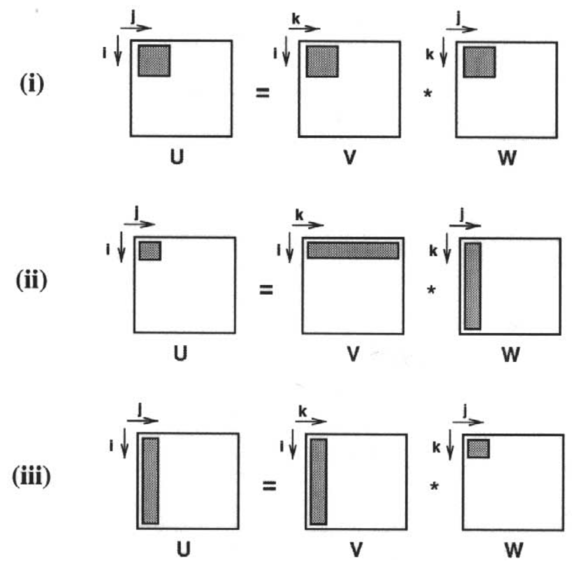

Fig. 5. Different tile allocations in the SPM for the matrix multiply code.

of iterations) of all the loops. We assume that $d \times n \leq M$ and that M is far less than n^2, where d is an integer. In those cases where $n > M$, we can apply the technique in this paper recursively (which, in a sense, corresponds to multilevel tiling [26]).

For now, let us suppose that the compiler works on square tiles of size $S_a \times S_a$ shown Fig. 5(i) and the default array layout is row-major. The access cost of a tile of size $S_a \times S_a$ is $S_aC + S_a^2t$, and n^2/S_a^2 of these data tiles are read (for array U). Consequently, the total read access cost for array U is $T_{\mathtt{U}} = (n^3/S_a^3)(S_aC + S_a^2t)$. The access costs for the other arrays can be computed similarly. Therefore, the overall access cost for the nest shown in Fig. 4(ii) (T^a_{overall}) considering all three arrays and the read activity alone can be calculated as

$$T^a_{\text{overall}} = T_{\mathtt{U}} + T_{\mathtt{V}} + T_{\mathtt{W}} = 3\frac{n^3}{S_a^3}\left(S_aC + S_a^2t\right) = \frac{3n^3}{S_a^2}C + \frac{3n^3}{S_a}t \tag{1}$$

under the memory constraint $3S_a^2 \leq M$. Here, $T_{\mathtt{U}}, T_{\mathtt{V}}$, and $T_{\mathtt{W}}$ denote the read access costs for arrays U, V, and W, respectively. Assuming that the entire available SPM capacity (M) will be used for this computation (i.e., $3S_a^2 = M$), this last formulation above can be rewritten in terms of M as

$$T^a_{\text{overall}} = \frac{3n^3}{M/3}C + \frac{3n^3}{\sqrt{M/3}}t. \tag{2}$$

This straightforward translation can be improved substantially using a more careful choice of memory layouts, loop ordering, and data-tile allocations. First, we observe from Fig. 4(ii) that it is not necessary to perform all the **tile_read** and **tile_write** activities inside the `kt` loop. In particular, a data tile of array U can be read (and written) between the tiling loops `jt` and `kt`. Second, reading square tiles does not allow the compiler to take advantage of the layout of data in off-chip memory. For example, the array V is stored in row-major order in the off-chip memory; therefore, reading a row-chunk instead of reading a square-chunk should result in fewer transfer calls. Fig. 4(iii) shows the code corresponding to this scenario. The corresponding tile allocation is shown in Fig. 5(ii). During the execution, tiles of sizes $S_b \times S_b$, $S_b \times n$, and $n \times S_b$ are accessed for arrays U, V, and W, respectively. In addition to determining the tile allocation and loop order, our approach assigns memory layouts to arrays. In the case of Fig. 5(ii), it assigns a row-major layout for V and a column-major layout for W. The layout of array U can be either row-major or column-major since we read square tiles for this array. Here, we have assumed that the trip counts of the loops are equal to the array sizes in the corresponding dimensions. Otherwise, the trip count if known compile time, or an estimate of the trip count based on profiling or such factors as the array size should be used instead of n.

Note that we have exploited data tile-level temporal locality by reading the data tiles as early as possible in the nest structure (instead of reading them repeatedly inside the innermost tiling loop), and reading as many data items as possible along the storage direction (the last dimension in the case of row-major order storage) in a single transfer call. Note also that since the compiler reads data tiles of sizes $S_b \times n$ and $n \times S_b$, for arrays V and W, respectively, the tiling loop `kt` does not appear in Fig. 4(iii). The overall read access cost of this loop order, layout assignment, and tile allocation scheme is

$$T^b_{\text{overall}} = \underbrace{\frac{n^2}{S_b^2}\left(S_bC + S_b^2t\right)}_{T_{\mathtt{U}}} + \underbrace{\frac{n}{S_b}(S_bC + nS_bt)}_{T_{\mathtt{V}}} + \underbrace{\frac{n^2}{S_b^2}(S_bC + nS_bt)}_{T_{\mathtt{W}}} = \left(\frac{2n^2}{S_b} + n\right)C + \left(2n^2 + \frac{n^3}{S_b}\right)t \tag{3}$$

under the assumption that $2nS_b + S_b^2 \leq M$, and that at most n array elements can be read with a single **read_tile** call from within the code. Note that we have made this assumption just to keep the presentation simple. Actually, if we can read/write more than n elements in a single call, the number of transfer operations in the code can be reduced further. In the best case, a single row-chunk or column-chunk can be read/written using a single call from within the code (with a suitable memory layout). It should also be noted that a single (data transfer) call in the code can correspond to multiple transfer activities at the hardware level (depending on the bandwidth between the off-chip memory and the SPM). A reduction in the number of transfer calls in the code also leads, in general, to a reduction in hardware-level transfer operations.

It should be emphasized that although S_b is different from S_a, the total memory (SPM) space used in both cases is the same (a capacity of M elements). As before, assuming that the maximum available SPM memory is used (i.e., $2nS_b + S_b^2 = M$), we can rewrite this last equation (in terms of M) as

$$T^b_{\text{overall}} = \left(\frac{2n^2}{\sqrt{n^2+M}-n} + n\right)C + \left(2n^2 + \frac{n^3}{\sqrt{n^2+M}-n}\right)t. \tag{4}$$

Let us now compare T^b_{overall} [from (4)] with T^a_{overall} [from (2)]. Assuming that $n = 128$ and $M = 8192$ (for illustrative purposes only), as compared to T^a_{overall}, this version improves the coefficient of C by 45%, and the coefficient of t by 12.2%. We should mention that even compared to a modified version of Fig. 4(ii) which reads the data tiles belonging to array `U` before the `kt` loop to exploit temporal locality, the code in Fig. 4(i) still brings about an additional reduction of 31.5% in the coefficient of C. It should be noted that the memory access cost of the code given in Fig. 4(iv) [and its corresponding tile allocation depicted in Fig. 5(iii)] is very similar to that of Fig. 4(iii). In this case, however, the compiler selects a different loop order, and assigns column-major memory layouts for both `U` and `V`.

This example clearly shows that selecting a suitable loop order, memory layout, and tile allocations can lead to significant reduction in the number of data transfers as well as in the amount of data transferred. In the remainder of this paper, we present a compiler-directed approach to automatically determine the loop orders, memory layouts, and tile allocations for a given piece of code. We first discuss the technique within the scope of a single multilevel nest, and then generalize it to multiple nests each accessing a subset of the arrays (of signals) declared in the program.

C. Desired Forms of Array References

In general, we can exploit data locality in the SPM for a given reference, if the reference exhibits either temporal or spatial reuse. Consequently, we want each reference to an m-dimensional row-major array `U` to be in either of the following two forms, where f_i and g_j are subscript functions.

- `U`$[f_1][f_2]\ldots[f_m]$: In this form, f_m is an affine function of all loop indices with a coefficient of 1 for the innermost loop index whereas f_1 through $f_{(m-1)}$ are affine functions of all loop indices except the innermost loop index.
- `U`$[g_1][g_2]\ldots[g_m]$: In this form, all g_1 through g_m are affine functions of all loop indices except the innermost loop index.

In the first case, we have spatial reuse for the reference in the innermost loop, and in the second case, we have temporal reuse in the innermost loop [24]. Note that the second case presents an opportunity for the compiler to keep data (actually the corresponding data tile) in the SPM throughout the execution of the innermost tiling loop. The first case, on the other hand, enables the compiler to read (resp. write) more consecutive data items in a single **read_tile** (resp. **write_tile**) call. In other words, the second case helps in reducing the coefficient of both C and t whereas the first case helps more in reducing the coefficient of C. The compiler's task, then, is to bring each array (reference) to one of the forms given above. In some codes, the array references are in one of these forms to begin with. However, in some other codes, we have references `U`$[h_1][h_2]\ldots[h_m]$ in which a subscript position other than the last contains the innermost loop index. Examples of this are references with coupled-indices such as `U[i+k][j+k]`, where `k` is the innermost loop index and `i` and `j` are other loop indices, and references, such as `V[k][j]`, where the access pattern (imposed by the innermost loop `k`) is orthogonal to the storage pattern (which is assumed to be row-major by default).

D. Algorithm for Determining Memory Layouts and Loop Order

In this section, we present a compiler algorithm to transform a given nested loop, such that the resulting nest will have references in one of the forms mentioned above. We assume that the compiler will determine the most appropriate memory layouts for each array as well as a suitable loop (i.e., iteration space) transformation that changes the loop execution order. Note that the off-chip memory layouts are not assumed to be fixed at any specific form. The algorithm, however, can be extended to accommodate those cases where some or all the off-chip memory layouts are fixed at specific forms.

The memory layout for an m-dimensional array can be in one of $m!$ forms, each corresponding to the linear layout of data in off-chip memory by a nested traversal of the array axes in some predetermined order. The innermost axis is called the fastest-changing dimension. For example, the second dimension is the fastest changing dimension for row-major memory layout of a two-dimensional array. The method that we present can also be used to handle blocked memory layouts by viewing each block (sub-matrix) as an array element. In other words, the methods presented in this paper are applicable, with appropriate modifications, to blocked layout cases as well. In the following, we assume that the transformed arrays will be stored in memory as row-major.

In our framework, each execution of an n-level nested loop is represented using an iteration vector $\bar{\mathtt{I}} = (\mathtt{i_1}, \ldots, \mathtt{i_n})$, where $\mathtt{i_j}$ corresponds to the jth loop from the outermost position. We assume that the array subscript expressions and loop bounds are affine functions of enclosing loop indices and loop-independent variables. With this assumption, each reference to an m-dimensional array `U` is represented by an access (reference) matrix $\mathcal{L}_u$ and an offset vector $\bar{l}_u$, such that $\mathcal{L}_u\bar{\mathtt{I}} + \bar{l}_u$, is the element accessed by a specific iteration $\bar{\mathtt{I}}$ [44]. As an example, consider a reference `U[i-2][j+2]` to a two-dimensional array `U` in a two-level loop nest with `i` is the outer loop and `j` is the inner loop. We have

$$\mathcal{L}_u = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \text{and} \quad \bar{l}_u = \begin{bmatrix} -2 \\ 2 \end{bmatrix}.$$

In general, if the loop nest is l-level and the array in question is m-dimensional, the access matrix is of size $m \times l$ and the offset vector is m-dimensional.

The class of loop transformations we are interested in can be represented using nonsingular square transformation matrices [44]. For a loop nest of l-levels, the iteration space transformation matrix $\mathcal{T}$ is of size $l \times l$. Such a transformation maps each iteration vector $\bar{\mathtt{I}}$ of the original loop nest to an iteration $\bar{\mathtt{I}}' = \mathcal{T}\bar{\mathtt{I}}$ of the transformed loop nest [43], [44]. Therefore, after the transformation, the new array access can be represented by $\mathcal{L}_u\mathcal{T}^{-1}\bar{\mathtt{I}}' + \bar{l}_u$, meaning that the new access matrix is $\mathcal{L}_u\mathcal{T}^{-1}$ [44]. A data transformation, on the other hand, is applied by transforming the dimensions (subscript expressions) of the reference. A square nonsingular $m \times m$ data transformation matrix $\mathcal{M}_u$ transforms the reference $\mathcal{L}_u\bar{\mathtt{I}} + \bar{l}_u$ (of an array `U`) to $\mathcal{M}_u\mathcal{L}_u\bar{\mathtt{I}} + \mathcal{M}_u\bar{l}_u$. Thus, the access matrix is transformed to $\mathcal{M}_u\mathcal{L}_u$ [29]. In this paper, we are interested in only permutation matrices for data transformations, i.e., we are only interested in

```
INPUT: A nested loop and the access matrices for the references in the nest.
OUTPUT: A loop transformation matrix $\mathcal{T}$ and a data transformation matrix $\mathcal{M}_i$ for each array $i$.

1: determine an access matrix for each array $i$ ($1 \leq i \leq s$)
2: for each of the $2^s$ alternatives do
    2.1: determine target $\mathcal{L}'_1$, $\mathcal{L}'_2$,...,$\mathcal{L}'_s$
    2.2: using $\mathcal{L}_i \mathcal{T}^{-1} = \mathcal{L}'_i$ determine a $\mathcal{T}$
    2.3: for each array $j$ with the spatial locality do
        2.3.1: let $r_k$ be the row (if any) containing the only non-zero element in the last column for $\mathcal{L}'_j$
        2.3.2: find an $\mathcal{M}_j$ such that $\mathcal{L}''_j = \mathcal{M}_j \mathcal{L}'_j$ will be in the desired form (i.e., $r_k$ will be the last row)
    2.4: endfor
    2.5: record for the current alternative with the computed coefficients of $C$ and $t$
3: endfor
4: select the most suitable alternative (see the explanation in the text)
5: apply data-conscious tiling (see the explanation in Section 4.6)
```

Fig. 6. Optimization algorithm.

dimension permutations of which there is a total of $m!$. Note that transforming the dimensions of an array can be thought of modifying the corresponding memory layout. For example, transforming a given row-major array U using the data transformation matrix

$$\mathcal{M}_u = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

corresponds to converting its memory layout to column-major. Note that, after the transformation, the array will be stored in off-chip memory as row-major. However, since the array subscript functions are permuted by the transformation, the new access pattern imitates a column-major memory layout. Note that applying a data transformation also requires modifications to array declaration [24].

Given an access matrix $\mathcal{L}_u$, our aim is to find matrices $\mathcal{T}$ and $\mathcal{M}_u$, such that the transformed access matrix will be in one of the two desired forms, as discussed above. Note also that while $\mathcal{T}$ is unique to a loop nest, we need to find an $\mathcal{M}_u$ for each array. When both loop and data transformations are applied, a reference $\mathcal{L}_u \bar{\text{I}} + \bar{l}_u$, becomes $\mathcal{M}_u \mathcal{L}_u \mathcal{T}^{-1} \bar{\text{I}}' + \mathcal{M} \bar{l}_u$. Since, for a given $\mathcal{L}_u$, determining both $\mathcal{T}$ and $\mathcal{M}_u$ simultaneously, such that $\mathcal{M}_u \mathcal{L}_u \mathcal{T}^{-1}$ is in a desired form, involves solving a system of nonlinear equations (not a trivial task), we solve it using a two-step heuristic approach. In the first step, we find a matrix $\mathcal{T}$, such that $\mathcal{L}'_u = \mathcal{L}_u \mathcal{T}^{-1}$ will have a last column whose entries are all zero except for the element in the rth row, which is 1 (for spatial locality in the innermost loop); or we find a $\mathcal{T}$, such that the last column of $\mathcal{L}'_u = \mathcal{L}_u \mathcal{T}^{-1}$ consists only of zeros (for temporal locality in the innermost loop). If the reference is optimized for spatial locality in the first step, then, in the second step, we find a matrix $\mathcal{M}_u$, such that this rth row in $\mathcal{L}'_u$ (mentioned above) will be the last row in $\mathcal{L}''_u = \mathcal{M} \mathcal{L}'_u$, assuming that following the transformation the array in question will be stored in off-chip memory in row-major.

The optimization algorithm is given in Fig. 6. We assume that there is a single uniformly generated reference set (UGR) [17] in the nest for a given array. A UGR is a set of array references (to the same array) whose subscript functions differ only in the constant term. For example, for a two-level nested loop with indices i and j, the references U[i] and U[i+2] are uniformly generated whereas the references U[i] and U[j-1] are not. Later on in this paper, we discuss how to handle those cases where there are multiple UGRs for a given array. Note that as far as the locality characteristics are concerned, each UGR set can be treated as a single reference. Therefore, in the following, we use the terms "array" and "reference" interchangeably. In Step 1 (of the algorithm in Fig. 6), we determine the UGR for each array accessed in the nest. Since (for each array) we have two desired candidate forms (one corresponding to temporal locality in the innermost loop, and the other corresponding to spatial locality in the innermost loop), we exhaustively try all the 2^s possible loop transformations in Step 2, each corresponding to a specific combination of localities (spatial or temporal) for the arrays. In Step 2.1, we set the desired access matrix $\mathcal{L}'_i$ for each array i and, in the next step, we determine a loop transformation matrix $\mathcal{T}$ that obtains as many desired forms as possible. We describe a typical optimization scenario now. Suppose that, without loss of generality, in an alternative v, where $1 \leq v \leq 2^s$, we want to optimize references 1 through b for temporal locality and references $b+1$ through s for spatial locality. After Step 2.2, we typically have c references that can be optimized for temporal locality and d references that can be optimized for spatial locality, where $0 \leq c \leq b$ and $0 \leq d \leq (s-b)$. This means that a total of $s-(c+d)$ references (arrays) exhibit no locality in the innermost loop. We do not apply any data transformations for the c arrays that have temporal locality in the innermost loop, and the associated data tiles can be kept in the SPM throughout the execution of the innermost tiling loop. For each array (of a maximum of d arrays) that can be optimized for spatial locality, within the loop starting at Step 2.3, we find a data transformation matrix, such that the resulting access matrix will be in our desired form.

After computing the number of data references with temporal and spatial reuse for a given alternative, in Step 2.5, the algorithm records c, d, and $s - (c + d)$ for this alternative, and estimates and records for future use the coefficients of C and t. In estimating these coefficients, an important step is to build a tiled form of the code and compute the number and volume of data transfers (which is possible only after a memory space partitioning is done as explained in Section 4.5). However, code generation is not performed at this point, but postponed until the best (most suitable) alternative is decided upon. After all the alternatives have been processed, we select the most suitable one. Although different techniques can be employed to select the most suitable alternative, we adopt the following strategy which takes into account constraints (maximum limits) for these coefficients. If our approach determines four alternatives where coef_{Ci} and coef_{ti} denote the coefficients of C and t for a given alternative i, respectively, it selects the alternative that satisfies $\mathrm{coef}_{Ci} < \mathrm{coef}_{C\max}$ and $\mathrm{coef}_{ti} < \mathrm{coef}_{t\max}$, and this alternative has minimum coefficient values among all those that satisfy these constraints. Here, $\mathrm{coef}_{C\max}$ and $\mathrm{coef}_{i\max}$ are the maximum allowable coefficients for C and t, respectively. In case we have two alternatives, i and j, that satisfy these constraints, and $\mathrm{coef}_{Ci} < \mathrm{coef}_{Cj}$ and $\mathrm{coef}_{ti} > \mathrm{coef}_{tj}$, the choice depends on the actual values for the transfer initiation cost and per element transfer cost. The estimation of the coefficients for a given alternative and the partitioning of the memory space are discussed in the next subsection.

There are three important points to note here. First, for a given nest with s references, our approach tries 2^s different transformation strategies. This is because for each reference there are two alternatives: spatial locality and temporal locality. It should be noted, however, that these trials are done statically (i.e., at compile time). That is, the compiler checks whether each reference can be optimized temporal or spatial locality (in the innermost loop position). As explained above, this problem is reduced to the problem of finding a suitable transformation matrix to realize the desired access matrix (i.e., the desired access pattern). Note that for the spatial locality the last column of the desired access matrix should be $[0, 0, 0, \ldots, 0, 0, 1]^T$ (that is, the innermost index—after the transformation—should appear only in the last subscript position). Similarly, for the temporal locality, the last column of the desired access matrix should be $[0, 0, 0, \ldots, 0, 0, 0]^T$ that is, the innermost index—after the transformation—should not appear in any subscript position). Given the original access matrices and desired access matrices, the compiler then determines the transformation matrix. Second, in completing the partially-filled loop transformation matrix $\mathcal{T}$, we use the approach proposed by Bik and Wijshoff [9] to ensure that the resulting matrix preserves all data dependences [44] in the original nest. Third, we also need a mechanism in some cases to favor one or more arrays over the others. The reason is that it may not always be possible to find a $\mathcal{T}$, such that all $\mathcal{L}'_i$ arrays targeted in a specific alternative can be realized. In those cases, we need to omit some references from consideration, and attempt to satisfy (optimize) the remaining. Profile information can be used for this purpose.

For an example application of the algorithm of Fig. 6, consider once more the matrix multiply code in Fig. 4(i) on page 9. The access matrices for arrays U, V, and W are

$$\mathcal{L}_u = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$
$$\mathcal{L}_v = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
$$\mathcal{L}_w = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

In this nest, temporal locality can be exploited for only one of the three arrays. The remaining two arrays can be optimized for spatial locality. This gives us a total of three alternatives. To keep the presentation simple, we focus only on one of them. In this alternative, we attempt to optimize array W for temporal locality and arrays U and V for spatial locality. Consequently, the desired access matrices are of the form[2]

$$\mathcal{L}'_u = \begin{bmatrix} \times & \times & 1 \\ \times & \times & 0 \end{bmatrix}$$
$$\mathcal{L}'_v = \begin{bmatrix} \times & \times & 1 \\ \times & \times & 0 \end{bmatrix}$$
$$\mathcal{L}'_w = \begin{bmatrix} \times & \times & 0 \\ \times & \times & 0 \end{bmatrix}.$$

Here, $\times$ denotes 'don't care'. Now, using $\mathcal{L}_u, \mathcal{L}_v, \mathcal{L}_w, \mathcal{L}'_u, \mathcal{L}'_v$, and $\mathcal{L}'_w$, we can determine

$$\mathcal{T}^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

We have

$$\mathcal{L}'_u = \mathcal{L}_u \mathcal{T}^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$
$$\mathcal{L}'_v = \mathcal{L}_v \mathcal{T}^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$
$$\mathcal{L}'_w = \mathcal{L}_w \mathcal{T}^{-1} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

So, the transformed code is

```
for (j = 0; j < n; j++)
  for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
      U[i][j] = U[i][j] + V[i][k] * W[k][j].
```

Note that this new access pattern exploits temporal locality for W in the innermost loop, and imposes a column-major memory layout for U and V. If the default memory layout is row-major, these two arrays should be data-transformed using matrices

$$\mathcal{M}_u = \mathcal{M}_v = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

Note that, when obtaining the tiled version, we use this transformed code as the starting point. Note also that this version eventually transforms to the tiled code given in Fig. 4(iv). The

[2]Other alternatives are also possible.

code in Fig. 4(iii), on the other hand, is obtained using the identity matrix as loop transformation and exploiting temporal locality for `U` and spatial locality for `V` and `W`. It should be noted that, in some codes, after these transformations, our framework also uses loop fusion to improve locality further. Our loop fusion strategy is very similar to that proposed by McKinley *et al.* [32] and its details are beyond the scope of this paper.

E. Memory-Space Partitioning

Once a suitable loop order and memory layouts for the arrays are determined, the compiler needs to partition the amount of available SPM between the arrays accessed in the nest. Let us assume first that there is a single reference per array. Our approach tiles all loops except for the innermost one. For example, assuming that the current alternative (in matrix-multiply code) is the loop nest given in previous subsection (i.e., the `j, k, i` loop), the compiler tiles both the loops `j` and `k` [as shown in Fig. 5(iii)].

Assuming that the loops are tiled using a blocking factor (tile size) S_a, if the reference in question is of the first desirable form discussed in Section 4.3, the compiler accesses a data tile of size $S_a \times S_a \times S_a \times \cdots \times S_a \times n$ (assuming that n is the number of iterations of the innermost loop, and the memory layout is row-major as in the C language). Note that accessing such a data tile tends to minimize the number of transfer calls per tile. As for the arrays that fit in the second form, we can access data tiles of $S_a \times S_a \times S_a \times \cdots \times S_a$ as the loops whose indices appear in some subscript function of this array are tiled. After that, the data tiles can be hoisted into upper loop positions depending on the loop indices used by their subscript functions.

However, if we consider the entire UGR set (which might contain multiple uniformly generated references to the same array), we need to be more careful in determining the data tile size for the array. In this work, we use the concept of the spread of a UGR set [1]. Informally, it consists of all of the elements that are accessed by the references in the set. Formally, given a UGR set $\mathtt{U}[f_{i1}][f_{i2}]\ldots[f_{im}]$, the spread of the subscript position k is $\max_{i,j}(c_{ik} - c_{jk})$, where c_{ik} and c_{jk} are the coefficient parts of the kth subscript for arbitrary references i and j, respectively. What this means that in order to accommodate all the array elements in kth subscript position, the normal tile size should be increased by at least $\max_{i,j}(c_{ik} - c_{jk})$ elements. As an example, if we have a UGR set that contains `U[i][j]` and `U[i + 3][j + 5]`, and the original tile size that we would consider if we had only the former reference is $S_a \times S_a$, the final tile size should be $S_a + 3 \times S_a + 5$ (to accommodate the whole set of elements accessed by both the references).

If, on the other hand, we have more than one UGR set for a given array (which is a rare occurrence in many embedded applications), our current implementation treats each UGR set as if it belongs to a different array. This simplification, of course, causes some performance loss in cases where the references belonging two different UGR sets partially overlap. However, it makes the code generation much simpler.

It should be noted that the compiler performs this memory space partitioning for each alternative (considered by the algorithm in Fig. 6); however, it does not generate code (i.e., does not actually tile the nest in question by modifying the internal data structures of the compiler) until it determines the most suitable alternative. The code modification step (explained in Section IV-F) is performed only for the selected best alternative.

F. Code Modification and Generation

An important issue is the placement of transfer calls in a given tiled nest. In fact, there are two subproblems here: 1) the insertion of the transfer calls in the code and 2) the determination of the parameters to be passed to them. Determining the insertion points for the transfer calls is relatively simple. For one, the calls should definitely be outside the element loops, as these loops work on the data tiles that are active in the SPM. Then, all that is left is to examine the subscripts of the reference in question to determine the indices that occur, and insert the transfer call associated with the reference in between the appropriate tiling loops. For example, in Fig. 4(iii), the subscript functions of array `V` use only the tiling loop index `it`. Since there are two tiling loops in this code, namely `it` and `jt`, we insert the read call for this reference just after the `it` loop, as shown in the figure. However, the other two references (`U` and `W`) use the tiling loop index `jt`; therefore, we need to place the read routines for these references inside the `jt` loop (just before the element loops). The write routines are placed using a similar reasoning.

For handling the second subproblem, namely, determining the sections to read and write, we use the method of extreme values of affine functions first used by Banerjee [7], [44] for data dependence testing. Given an affine function of a number of variables and inequalities which represent the bounds for the variables, the extreme values method determines the maximum and minimum values of the affine function in the bounded region. This method applies to nonrectilinear regions as well. We can describe this method using a simple example. Consider the affine function $f(i,j) = 5i - 4j + 4$ with the bounding region $\{(i,j) \mid 1 \le i \le 100 \text{ and } i + 2 \le j \le 200 - i\}$. Then

$$\begin{array}{rl} \text{For the upper bound :} & \text{For the lower bound :} \\ f(i,j) \le 5i - 4j + 4 & f(i,j) \ge 5i - 4j + 4 \\ f(i,j) \le 5i - 4(i+2) + 4 & f(i,j) \ge 5i - 4(200 - i) + 4 \\ f(i,j) \le i - 4 & f(i,j) \ge 9i - 796 \\ f(i,j) \le 96 & f(i,j) \ge -787. \end{array}$$

Therefore, the upper bound of $f(i,j)$ is 96 and the lower bound is -787. That is, the values that the function f take on fall in the interval $-787 : 96$.

We use a similar analysis to compute the range of array elements accessed by a complete execution of the element loops. Let us concentrate now on the reference `U[i][j]` in Fig. 4(i). Assuming the tiling strategy in Fig. 4(iii), we want to find the range of elements for each subscript function of this reference. Since $\mathtt{it} \le \mathtt{i} \le \mathtt{it} + S_a - 1$, the range of elements accessed by the first subscript is $\mathtt{it} : \mathtt{it} + S_a - 1$. Similarly, the range of elements accessed by the second subscript is $\mathtt{jt} : \mathtt{jt} + S_a - 1$. Consequently, the array section (data tile) that needs to be read from the off-chip memory for the reference `U[i][j]` is $\mathtt{it} : \mathtt{it} + S_a - 1, \mathtt{jt} : \mathtt{jt} + S_a - 1$. The array sections for the other references are found using the same approach.

```
INPUT: A series nested loops
OUTPUT: Optimized nest sequence and memory layouts for arrays

1: Transform the program into a sequence of independent loop nests using loop fusion [44], distribution [44], and
   code sinking [44].
2: Build an interference graph [5] and identify the connected components. The interference graph is a bipartite
   graph $(V_n, V_a, E)$, where $V_n$ is the set of loop nests, $V_a$ is the set of arrays, and $E$ is the set of edges between
   loop nodes and array nodes. There is an edge $e \in E$ between $v_a \in V_a$ and $v_n \in V_n$ if and only if $v_n$ references
   $v_a$.
3: For each connected component:
   3.1: Order the loop nests according to a cost criterion using profile information.
   3.2: Optimize the most costly (i.e., the most important) nest using the technique discussed in the previous
        subsection and then tile this nest.
   3.3: For each of the remaining nests in the connected component (according to their cost orders):
      3.3.1: Optimize the nest being analyzed using the mentioned technique. However, in determining the loop
             transformation, first apply the data transformations found so far to respective references. Then, tile
             the nest.
      3.3.2: Propagate the data transformations found so far to the remaining nests in the procedure.
```

Fig. 7. Procedure-wide optimization algorithm.

G. Multiple Nests and the Interprocedural Problem

The technique proposed in the previous subsections to bring an array reference into the desired form involves a data (memory layout) transformation. Unlike loop transformations, the impact of a data transformation is global in the sense that it affects the locality characteristics of all the references to the same array in every loop nest and in every procedure in the program. In other words, when a data transformation matrix is applied to a reference, it should also be applied to all the other references to the same array whether belong to the same nest or not. Next, we briefly discuss how to control this global effect. The details of a global locality analysis for off-chip memory arrays are beyond the scope of this paper, however.

First, we focus on the intraprocedural locality optimization problem and present an approach to optimize a series of loop nests collectively. Note that code sequences that contain consecutive loop nests are quite common in video and image processing applications. Given a series of nested loops that access (possibly different) subsets of arrays declared in the procedure, a sketch of our optimization strategy is shown in Fig. 7.

We now briefly discuss the interprocedural locality optimization problem. It is easy to see that a naive approach that remaps all arrays across procedure boundaries can be prohibitively expensive and can easily outweigh all gains obtained from SPM optimizations. Our current approach, instead, propagates data transformation matrices across procedures. It is similar in spirit to the global data distribution algorithm proposed by Anderson in her thesis [5], and can be summarized briefly as follows.

Our approach performs two traversals on the call graph representation of the program. A call graph $G_c = (V_c, E_c)$ is a multigraph where each node $p_i \in V_c$ represents a procedure, and there is an edge $e \in E_c$ between p_i and p_j if the former calls the latter [2]. In such a graph, the leaves represent the procedures that do not contain any calls. If desired, the edges can be annotated by some useful information related to call sites, such as the actual parameters passed to the procedure, the line number where the call occurs, and so on. Currently, we do not handle programs that contain recursive procedure calls, and we do not handle arrays that are explicitly reshaped across procedure boundaries.

Before the first traversal, we run an intraprocedural locality optimization algorithm (summarized above) on each leaf node. In the first traversal, called bottom-up, we start with the leaves and process each node in the call graph if and only if all the nodes it calls have been processed. After all the callee nodes for a given caller have been processed, we propagate a system of equalities (called layout or locality constraints) to the caller. The solutions to these equalities are such that they give us the loop and data transformations that collectively bring the references in the procedure being analyzed to one of the desired forms.

The caller adds this system to its own local set of equalities (obtained using the intraprocedural locality optimization algorithm) and propagates the resulting system to its callers, and so on. When we reach the root (the main program), we have all the locality constraints of the program. We solve these constraints at the root and determine the data transformation matrices for the (global and local) arrays accessed by the root and the loop transformation matrices.

Now, the top-down traversal phase starts; in this traversal, each caller propagates down the data transformation matrices determined so far to its callees. Using the equalities solved so far, the callees, in turn, determine the data transformation matrices for their local arrays as well as the loop transformations for the nests that they contain. Then, they apply tiling to these nests. When all the leaf nodes have been processed, the algorithm terminates.

For an illustration of the intraprocedural algorithm, consider the code fragment below that consists of a sequence of two separate nests.

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    V[j][i] = V[j][i] + U[i][j]
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    W[i][j] = V[i][j] + 1
```

Assuming the the first nested loop is more important than the second nest, our approach proceeds as follows. First, it focuses on the first nest, and assigns a row-major layout for U and a column-major layout for V. It then moves to the second nest, and in optimizing this nest it takes into account the layouts of V and U determined during the optimization of the first nest. This column-major layout of V requires a loop interchange for the second nest, thereby placing the i loop in the innermost position. This new access pattern, in turn, imposes a column-major layout for array W. Of course, after the optimization of each nest, the nest should be tiled and explicit data transfer calls should be inserted in the code.

H. Discussion

In this subsection, we discuss the similarities and differences between the compiler support required by an SPM-based environment (such as ours) and that required by a cache-based environment. Note that both the environments benefit from data locality optimizations, such as code (loop) and array layout (data) transformations. In fact, both an SPM and a cache based environments benefit from obtaining unit-stride accesses in the innermost loop positions. Consequently, our layout optimization can be used in a cache-based environment as well.

However, there are also significant differences between two compiler supports. First, the SPM-based compilation strategy is inherently data oriented. That is, in order to exploit data locality as much as possible, it tries to reuse the data in the SPM. Consequently, given an SPM content, it needs to execute the iterations that use the data that reside currently in the SPM. This strategy leads to an automatic, data-oriented tiling of the code. In other words, in an SPM-based environment, tiling is mandatory. In contrast, in a pure cache-based environment, tiling is an optional optimization (for enhancing cache performance), and, in fact, if there is no temporal reuse in outer loop positions (of a given nest), applying tiling is useless. Even in cases (in a cache-based environment) where tiling is desired, it is applied in an iteration space oriented manner. This is because since it is not possible to know what data (at a given time) reside in the cache, a tiling strategy cannot restructure the code based on cache contents; rather, some heuristics are used.

Second, explicit management of SPM requires the compiler to insert explicit data transfer calls in the code. Note that these are necessary to manage the contents of the SPM dynamically. In cache memories, on the other hand, the compiler is not required to do such a thing as the data flow management is handled by hardware automatically. Third, as a result of explicit management of the SPM space, the compiler needs to divide this space between competing arrays as discussed earlier in the paper. Finally, since the compiler/user knows what data are in the SPM at a given time, we can estimate the execution time of the program given an access pattern and memory layouts. Such estimations can then be used for selecting appropriate memory layouts and loop transformations. While it is also possible to estimate cache hits/misses in an optimizing compiler, such estimates are, in general, far from being accurate. This is because it is extremely difficult to predict the occurrence and frequency of conflict misses. Note that in an SPM-based environment conflict misses do not exist.

V. Experimental Results

In this section, we present experimental results and compare our dynamic technique with four other approaches. Our experimental suite consists of five benchmarks: int_mxm, an integer matrix multiply program (that contains one initialization and one multiplication nest); full_search and parallel_hier, two different motion estimation codes; rasta_fft, a discrete Fourier analysis code from MediaBench [28]; and rasta_flt, a filtering routine from MediaBench [28]. In order to enable our implementation to handle rasta_fft*and*rasta_flt, the pointer references are replaced with their equivalent array counterparts. Unless stated otherwise, the data set sizes (input sizes) for int_mxm, full_search, parallel_hier, rasta_fft, and rasta_flt are 196 K, 171 K, 171 K, 224 K, and 119 K, respectively. We use five different versions of each code:

- **tiled** is the version in which all arrays involved in the computation are accessed using square data tiles, and all array layouts are fixed to be row-major. In this version, at a given time, the SPM space is divided between the involved arrays evenly. This is a straightforward SPM utilization strategy.
- **static** is the version that allocates the entire SPM space for one chunk of data throughout the execution. In order for this scheme to be beneficial, we place the most frequently used data chunk in the SPM.
- **c_opt** is the dynamic SPM management strategy proposed in this paper. It determines memory layouts, tile sizes, and SPM partitioning between arrays automatically, and modifies the input code accordingly.
- **hand** is an hand-optimized version. In selecting the tile shapes, it considers not only the loop nest in question, but it takes into account the opportunities for tile reuse between multiple nests. Consequently, it is more global than our approach. Integrating this version into our framework is in our future agenda.
- **cache** is the version that uses the available SPM space as a conventional cache; that is, the hardware controls the data transfers between the on-chip memory and the off-chip memory.

The **c_opt** version is implemented using SUIF, a source-to-source translator [4]. The compiler analyzes the input code, determines the inherent data reuse, selects the best option, tiles the code accordingly, and inserts explicit data transfer calls. It should be emphasized that the layout transformation approach used by our technique works both intraprocedurally and interprocedurally. Out of the five codes in our experimental suite, only two codes (rasta_fft and

program		tiled	c_opt	hand
int_mxm	*coef_C*:	147,456	55.2%	63.8%
	coef_t:	2,724,280	30.9%	47.1%
full_search	*coef_C*:	803,462	41.4%	50.8%
	coef_t:	8,077,528	27.0%	33.4%
parallel_hier	*coef_C*:	634,016	39.7%	47.0%
	coef_t:	5,993,860	26.2%	28.2%
rasta_fft	*coef_C*:	845,720	36.3%	47.7%
	coef_t:	10,661,036	28.5%	36.1%
rasta_flt	*coef_C*:	495,992	44.6%	48.1%
	coef_t:	6,393,520	40.6%	48.7%

Fig. 8. Number of data transfers and transferred data volume ($M = 4$ K). coef_C is the coefficient of C and coef_t is the coefficient of t. The fourth and fifth columns are normalized values with respect to the corresponding values in the third column.

rasta_flt) benefited from interprocedural layout optimization. For the purposes of this study, we also collect statistics during execution by instrumenting the code, such as the number of data transfer calls (i.e., coefficient of C), and the number of data items transferred between the off-chip memory and the SPM (i.e., coefficient of t). For the experiments that involve the **cache** version, we employ a trace-driven cache simulator (DineroIV) [15]. The memory traces are obtained by instrumenting the program and recording the address and size of each array access along with the access type (read or write). It should be noted that the **c_opt** and **cache** versions use exactly the same memory layouts and loop optimizations in order to isolate the benefits that are solely due to the management of the on-chip memory space.

Our presentation is in five parts. Fig. 8 shows the number of the data transfers (coefficient of C, denoted coef_C), the transferred data volume (coefficient of t, denoted coef_t), and the percentage improvements for a 4 K SPM for three different versions of each program. It should be noted the corresponding values for **static** are negligible as it loads the SPM only once, and never loads it again during execution. The column **tiled** in Fig. 8 gives the absolute number of data transfers and the transferred data volume (resulting from the straightforward SPM management) whereas the last two column present the percentage improvements (reductions in coefficients) over **tiled**. These results are very encouraging, and show that, on the average, our approach reduces the number of transfers by 43.4% and data volume by 30.6%. Considering the tile sharing pattern between different nests (that is, the **hand** version) brings an additional 8.1% improvement in the number of transfers, and an additional 9.1% improvement in the transferred data volume.

Fig. 9 gives the total data access costs (in millions) for four different versions, again assuming that the size of the SPM (M) is 4 K. The total data access cost has four components: the transfer initiation cost (C), the per item transfer cost (t), the off-chip memory access cost (K_{off}) which does not include the per item transfer cost, and the on-chip memory access cost (K_{on}). Fig. 9 presents results for nine combinations of the ratio $C : t : K_{\text{on}} : K_{\text{off}}$. For example, a ratio, such as $5 : 5 : 1 : 10$, indicates that the data transfer initiation cost and per item transfer cost are the same (5). At the same time, the cost of off-chip memory (circuitry) access, K_{off}, is twice that of C, and the cost ratio between the on-chip and the off-chip memory

program	tiled	static	c_opt	hand
$C : t : K_{on} : K_{off}$ = **5:5:1:10**				
int_mxm	58.37	272.42	45.30	38.70
full_search	141.95	180.77	107.59	99.92
parallel_hier	109.85	159.93	85.21	78.69
rasta_fft	180.92	1,106.61	133.02	121.32
rasta_flt	115.15	163.93	75.68	68.89
$C : t : K_{on} : K_{off}$ = **9:1:1:10**				
int_mxm	48.07	127.24	38.05	33.13
full_search	112.86	180.77	85.90	79.85
parallel_hier	88.41	159.93	68.99	63.97
rasta_fft	141.66	1,106.61	104.91	95.78
rasta_flt	91.56	163.93	61.42	56.37
$C : t : K_{on} : K_{off}$ = **5:5:1:20**				
int_mxm	85.62	254.48	64.10	53.13
full_search	222.73	361.55	166.56	154.04
parallel_hier	169.79	319.87	129.56	118.85
rasta_fft	287.53	2,213.22	208.71	189.55
rasta_flt	179.09	327.87	114.04	102.78
$C : t : K_{on} : K_{off}$ = **9:1:1:20**				
int_mxm	75.31	544.85	56.84	47.57
full_search	193.63	361.55	144.87	133.97
parallel_hier	148.35	319.87	113.34	104.13
rasta_fft	248.27	2,213.22	180.60	164.02
rasta_flt	155.50	327.87	99.79	90.25
$C : t : K_{on} : K_{off}$ = **5:5:1:5**				
int_mxm	44.75	63.62	35.90	31.48
full_search	101.56	90.38	78.11	72.86
parallel_hier	79.88	79.97	63.03	58.61
rasta_fft	127.61	553.30	95.17	87.20
rasta_flt	83.19	81.96	56.50	51.95
$C : t : K_{on} : K_{off}$ = **9:1:1:5**				
int_mxm	34.45	63.62	28.65	25.91
full_search	72.47	90.38	56.42	52.79
parallel_hier	58.44	79.97	46.81	43.89
rasta_fft	88.35	553.30	67.06	61.67
rasta_flt	59.60	81.96	42.24	39.42

Fig. 9. Total data access costs (in millions) for different versions ($M = 4$ K).

is $1 : 10$. Out of the six combinations considered in Fig. 9, the first group of two uses a ratio $1 : 10$ between the costs of on-chip and off-chip memory accesses (i.e., $K_{\text{on}} : K_{\text{off}} = 1 : 10$). The next (resp. the last) group of two alternatives investigates the case where the off-chip cost is increased (resp. decreased) with respect to the on-chip cost. Within each group, we experiment with different relative costs of C and t. Note that the performance of the version **static** is assumed to be independent of C and t as their contribution to the overall cost (for that version) is negligible. From these results, we make two main observations. First, for all the experiments except when $K_{\text{on}} : K_{\text{off}} = 1 : 5$, the performance of **static** is very poor, indicating the need for dynamic management of the SPM space. When $K_{\text{on}} : K_{\text{off}} = 1 : 5$, on the other hand, the accesses to the off-chip memory become less costly, making the performance of the static management better (although it is still much worse than others). In the experiments with the **static** version, we assumed that the data that reside in the off-chip memory (not resident in the SPM) is accessed directly and not cached (e.g., using a conventional cache). We also performed another set of experiments where the non-SPM data is accessed through an on-chip cache. The experiments performed with 1, 2, and 4 K direct-mapped and 4-way set-associative caches showed that the performance of the **static** version is improved by 8.1% to 10.2% (over **static** without cache support) which is still much worse than the performance of the other versions. Moreover, note that, the **static** version uses more on-chip storage space (i.e., cache + SPM) than the others in this second set of experiments. We do not present detailed results due to lack of space. Another observation from Fig. 9 is that **c_opt** significantly improves over **tiled** and its performance

SPM Capacity (M)	tiled	static	c_opt	hand
$C:t:K_{on}:K_{off}$ = 5:5:1:10				
2K	76.04	385.27	56.94	47.65
4K	58.37	272.42	45.30	38.70
8K	46.04	192.63	37.19	32.46
16K	37.39	136.21	31.46	28.06
32K	31.31	96.31	27.43	24.96
64K	27.03	68.10	24.59	22.78
128K	24.02	48.15	22.58	21.24
$C:t:K_{on}:K_{off}$ = 9:1:1:20				
2K	100.33	770.54	73.27	60.21
4K	75.31	544.85	56.84	47.57
8K	57.89	385.27	45.42	38.78
16K	45.71	272.42	37.37	32.60
32K	37.16	192.63	31.71	28.25
64K	31.16	136.21	27.72	25.19
128K	26.93	96.31	24.91	23.03

Fig. 10. Total data access costs (in millions) for different versions of the `int_mxm` code (total data size $= 196$ K).

is close to that of **hand**. For example, when $C : t : K_{\text{on}} : K_{\text{off}} = 5:5:1:10$, our approach reduces the total data access cost over **tiled** by 26.3% on average. With the same parameters, **hand** improves over **tiled** by 32.8%; that is, 6.5% better than **c_opt**.

We now focus on the `int_mxm` code and study the performance of different versions under different SPM capacities using two example sets of the $(C : t : K_{\text{on}} : K_{\text{off}})$ parameters. Note that the total input size is fixed across different SPM capacity values. We observe from the results given in Fig. 10 that the effectiveness of our approach (over **tiled**) increases as the capacity of the SPM is reduced. For example, with $C : t : K_{\text{on}} : K_{\text{off}} = 9 : 1 : 1 : 20$, the improvement obtained by **c_opt** (over **tiled**) is 7.5%, 18.2%, and 27.0% for $M = 128$ K, $M = 16$ K, and $M = 2$ K, respectively. These results indicate that our approach is able to take advantage of small on-chip memories better, which is good because the dataset sizes keep getting bigger and bigger. We observed a similar trend with other codes as well.

We now focus on the problem of exploiting SPM from a slightly different perspective. So far, we have shown that our approach significantly outperforms a straightforward strategy under the assumption of a fixed SPM capacity. In other words, we have focused on improving performance by keeping the on-chip memory capacity constant. We now investigate the problem of reducing the required SPM capacity by keeping the performance constant. For example, recall that the number of transfer calls for the codes in Fig. 4(ii) and (iii) were, respectively

$$\text{coef}_{S_a} = \frac{3n^3}{S_a^2} \quad \text{and} \quad \text{coef}_{S_b} = \frac{2n^2}{S_b} + n.$$

If we want to keep these coefficients equal, then we have

$$\frac{3n^3}{S_a^2} = \frac{2n^2}{S_b} + n \Longrightarrow S_b = \frac{2n}{\frac{3n^3}{S_a^2} - 1} = G(S_a)$$

where $G(.)$ is a function. Now, given an original SPM capacity M, from $M = 3S_a^2$, we can find an S_a, and then using $S_b = G(S_a)$, we can find an S_b, and, finally, substituting this S_b in $2nS_b + S_b^2 = M'$ can give us a new SPM capacity M' (smaller than M) which, if used in conjunction with the code in Fig. 4(iii), will result in the same number of transfer calls had

M	$\%red$	M	$\%red$	M	$\%red$	M	$\%red$	M	$\%red$
1K	24.3	2K	28.8	4K	33.1	8K	39.7	16K	45.7

Fig. 11. Percentage reduction in memory size keeping the performance constant for the `full_search` code.

we used original capacity M with the code in Fig. 4(ii). Note that a similar formulation can be constructed if we want to keep the transfer volume or even total access cost fixed. Fig. 11 gives (for the `full_search` code) the percentage reduction (denoted $\%\text{red}$) in SPM size when the total access cost is fixed for different values of M using $C : t : K_{\text{on}} : K_{\text{off}} = 5 : 5 : 1 : 10$. We see from this table that, on the average, a 34.3% reduction in SPM capacity is obtained by keeping the total access cost fixed.

Finally, we compare the performance of a dynamically managed SPM to the performance of a traditional data cache memory. Fixing[3] $C : t : K_{\text{on}} : K_{\text{off}} = 5 : 5 : 1 : 10$, Fig. 12 shows the percentage increases in total data access cost when an on-chip cache memory is used instead of the SPM (with **c_opt**). We experimented with sizes of 8 and 32 K. It is assumed that the block size used in transfers between the cache/SPM and off-chip memory is 32 bytes. These results show that, on the average (i.e., across all benchmarks and cache configurations experimented with), using a conventional cache instead of SPM increases the total data access cost by 39.8% (assuming that the cache and SPM have the same capacity). Note that similar results have also been reported by Benini *et al.* [8]. We also observe that increasing the associativity from 1 (direct-mapped case) to 2 improves cache performance whereas going from 2 to 4 in general degrades the performance of the conventional cache version (due to the overhead factor we used and the lack of a significant drop in the number of conflict misses as a result of increased associativity). These results clearly show that, even under the same set of optimizations, the SPM-version outperforms the cache version significantly.

It is also important to see whether working with a larger conventional cache can outperform SPM. To check this, we performed another set of experiments, where we kept the SPM size at 8 K and increased the data cache size from 8 to 128 K. The results given in Fig. 13 are the averages over all associativities (1, 2, and 4) and write policies (WT and WB). These results reveal that even if we use a data cache size of 32 K, we are still able to achieve better results with an SPM of 8 K. Only when the data cache size is increased to 64 K, the cache version starts to outperform the SPM-based system. What this means is that using an SPM in conjunction with our optimizations can generate better results (in some cases) than using a conventional cache with the same set of optimizations.

VI. Related Work

Several strategies have been proposed to improve cache performance, including prefetching [35], data copying between different portions of the address space [27], [40] and locality optimizations for array-based codes [6], [12], [24], [36], [43], [44]. Dynamic techniques, such as access reordering on-the-fly

[3]This is assuming a direct-mapped cache. For a two-way (resp. four-way) associative cache, we assume $K_{\text{on}} = 1.1$ (resp. $K_{\text{on}} = 1.2$) to take into account the additional overhead due to associativity.

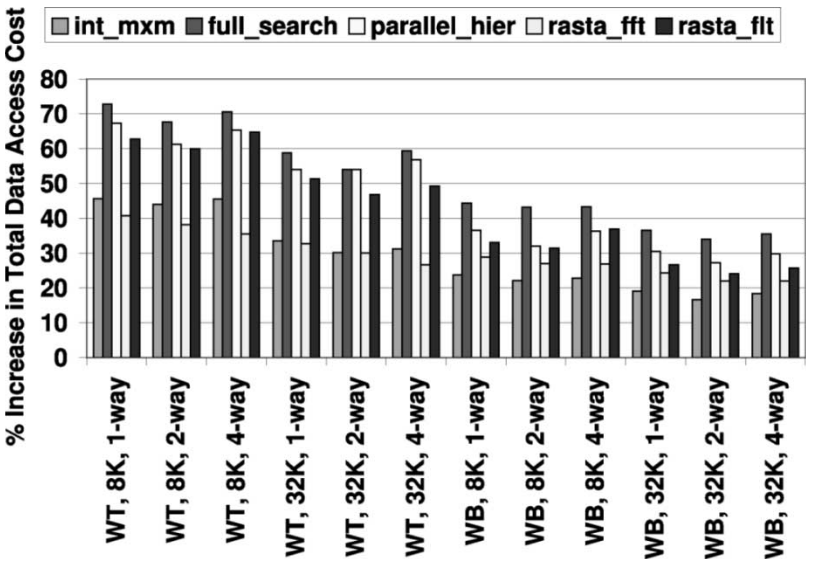

Fig. 12. Percentage increase (over dynamically-managed SPM) in total data access cost when a conventional on-chip cache is used $(\mathrm{WT} = \mathrm{write-through}; \mathrm{WB} = \mathrm{write-back})$.

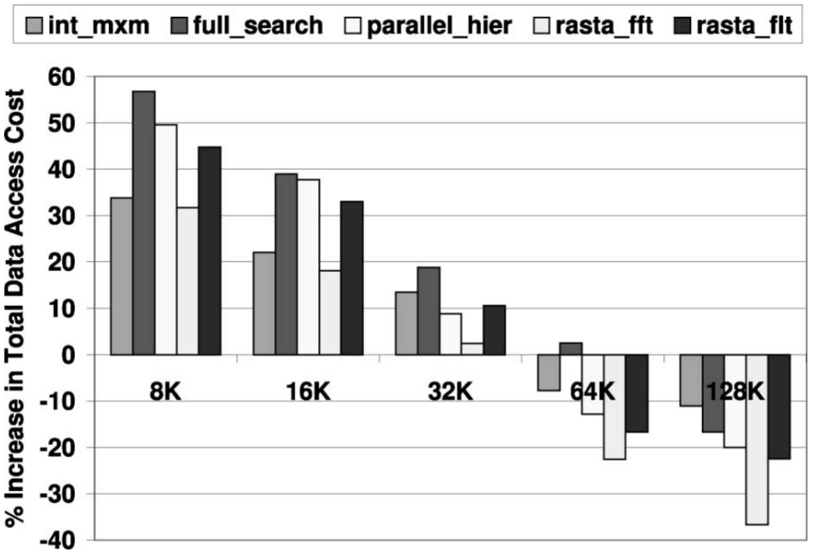

Fig. 13. Percentage increase (over an 8 K dynamically-managed SPM) in total data access cost when conventional data caches of different sizes are used.

between the processor and memory (using a specialized hardware unit), have also been used [13]. While these techniques reduce the number of cache misses, they do not completely eliminate them, and they do not solve the problem of unpredictable data access latency associated with cache memories. Recently, a number of runtime-based approaches [3], [16], [19], [33], [34] have been proposed to improve locality of irregular array applications. While these techniques specifically focus on irregular applications, the framework proposed in this paper targets embedded applications with regular access patterns.

The IMEC group in Belgium [10], [11] has 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 are addressed, among others, by Panda *et al.* [38] and Shiue and Chakrabarti [39]. Panda *et al.* [38] use the cache size and the processor cycle count as performance metrics, and propose a method for off-chip data placement. Shiue and Chakrabarti [39] extend Panda *et al.*'s approach to take into account the energy consumption as well. They show the importance of including energy as a key metric, and perform experiments with different cache topologies and tile sizes.

Hallnor and Reinhardt [18] propose a new software-managed cache architecture and a new data replacement algorithm. Mandhani *et al.* [30] discuss a technique that utilizes an existing cache as software-managed memory (with small architectural modifications.) They increase the instruction set architecture (ISA) with block load/store instructions, and discuss an addressing scheme for this architecture. In contrast, our approach employs an SPM, and we present an optimization strategy that utilizes both loop and data transformations for effective use of this memory. In addition, our approach does not require any ISA modification and does not incur any reconfiguration cost.

Cooper and Harvey [14] present two compiler-directed methods for the software management of a small cache for holding spilled register values. They show that using the coloring paradigm from register allocation can significantly reduce the amount of memory required for the program. Our work, on the other hand, tries to utilize the software-managed memory for general array-based computations, and not just for minimizing the negative impact of spill code on memory performance.

Wang *et al.* [42] propose a framework for analyzing the flow of values and data reuse for on-chip memories. They do not perform any interprocedural analysis and assume that the loops are perfectly-nested. They also impose specific forms on array references and assume that the subscripts are not coupled. The technique they use eventually results in suitable groupings of array statements from locality and bandwidth perspectives. Our technique focuses more on the optimization of the flow of data in a given loop nest (or whole program), and deals with issues, such as memory layout determination and insertion of explicit data transfer calls in the code. Also, we use a larger suite of optimizations (e.g., interprocedural analysis and layout transformations) and are able to handle a larger class of array references.

Panda *et al.* [37] present an elegant static data partitioning scheme for efficient utilization of SPM. Their approach is aimed at eliminating potential conflict misses due to the limited associativity of an on-chip cache. This approach benefits applications with a number of small (and highly reused) arrays that can fit in the SPM. Our work, in contrast, is oriented more toward dynamically managing the on-chip SPM by keeping the data transfers under compiler control. In a sense, these two works are complementary. In addition, our work uses a powerful optimization scheme and we show that the performance of the on-chip SPM may not be satisfactory in the absence of compiler optimizations.

Benini *et al.* [8] discuss a powerful memory management scheme that is based on keeping the most frequently used data items in a software-managed memory instead of a conventional cache. This approach is a static management technique as it does not adapt the contents of the on-chip memory to the dynamically changing working sets. On the other hand, it is not limited to array-dominated codes. Also, it is oriented more toward building such an application-specific memory architecture. Our work, on the other hand, focuses on a dynamic strategy in which portions of datasets involved in a given computation move back and forth between the on-chip and off-chip memory under compiler control. Our locality optimization framework attempts to decrease the frequency and volume of these explicit data transfers by using a suitable combination of loop and data transformations.

VII. Summary and Future Work

Conventional on-chip cache memories may not be very suitable for embedded systems where meeting hard real-time constraints is critical. Consequently, systems that contain a software-managed SPM can be of great interest as they exhibit highly predictable data access latencies. This paper presents a compiler-directed approach for the dynamic management of an SPM for array-based applications from the image and video processing domains. Our approach uses a set of compiler optimizations and a strategy for partitioning the on-chip memory space aimed at utilizing the on-chip memory space as effectively as possible. We compare the proposed scheme to a technique that adopts a static SPM management scheme (which fixes the contents of the SPM at the beginning of the execution), a technique that partitions the available on-chip space between competing arrays evenly, and a conventional cache scheme.

Our current work along this direction includes dynamic management of multiple on-chip SPMs, and enhancing our optimization suite to include multiarray transformations, such as array interleaving. Work is also underway in addressing the problem of effective management of data flow in memory hierarchies constructed from multiple cache memories and SPMs.

References

[1] A. Agarwal, D. Kranz, and V. Natarajan, "Automatic partitioning of parallel loops and data arrays for distributed shared memory multiprocessors," in *Proc. Int. Conf. Parallel Process.*, 1993.

[2] A. V. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Reading, MA: Addison-Wesley, 1986.

[3] I. Al-Furaih and S. Ranka, "Memory hierarchy management for iterative graph structures," in *Proc. 12th Int. Parallel Process. Symp.*, Orlando, FL, Apr. 1998.

[4] 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 Process. Sci. Comput.*, Feb. 1995.

[5] J. Anderson, "Automatic computation and data decomposition for multiprocessors," Ph.D. dissertation, Stanford Univ., Stanford, CA, Mar. 1997.

[6] A.-H. A. Badawy, A. Aggarwal, D. Yeung, and C.-W. Tseng, "Evaluating the impact of memory system performance on software prefetching and locality optimizations," in *Proc. 15th Annu. Int. Conf. Supercomput.*, Sorrento, Italy, June 2001.

[7] U. Banerjee, *Dependence Analysis for Supercomputing*. Norwell, MA: Kluwer, 1988.

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

[9] A. Bik and H. Wijshoff, "On a completion method for unimodular matrices," Dept. Computer Science, Leiden Univ., Tech. Rep. 94–14, 1994.

[10] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. De Man, "Global communication and memory optimizing transformations for low power signal processing systems," in *Proc. IEEE Workshop VLSI Signal Process.*, 1994, pp. 178–187.

[11] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology—Exploration of Memory Organization for Embedded Multimedia System Design*. Norwell, MA: Kluwer, 1998.

[12] M. Cierniak and W. Li, "Unifying data and control transformations for distributed shared memory machines," in *Proc. SIGPLAN'95 Conf. Program. Lang. Design Implement.*, June 1995.

[13] S. Coleman and K. McKinley, "Tile size selection using cache organization and data layout," in *Proc. SIGPLAN'95 Conf. Program. Lang. Design Implement.*, June 1995.

[14] K. D. Cooper and T. J. Harvey, "Compiler-controlled memory," in *Proc. Int. Conf. Archit. Support Program. Lang. Operat. Syst. (ASPLOS'98)*, CA, Nov. 1998.

[15] Dinero IV Trace-Driven Uniprocessor Cache Simulator [Online]. Available: http://www.cs.wisc.edu/~markhill/DineroIV/.

[16] C. Ding and K. Kennedy, "Improving cache performance in dynamic applications through data and computation reorganization at runtime," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.*, GA, May 1999.

[17] D. Gannon, W. Jalby, and K. Gallivan, "Strategies for cache and local memory management by global program transformations," *J. Parallel Distrib. Comput.*, vol. 5, pp. 587–616, 1988.

[18] E. G. Hallnor and S. K. Reinhardt, "A fully-associative software-managed cache design," in *Proc. Int. Conf. Comput. Architect.*, Vancouver, Canada, 2000, pp. 107–116.

[19] H. Han and C.-W. Tseng, "Improving locality for adaptive irregular scientific codes," in *Proc. 13th Int. Workshop Lang. Compilers Parallel Comput.*, Yorktown Heights, NY, Aug. 2000.

[20] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed. San Mateo, CA: Morgan Kaufmann, 1995.

[21] J. Eyre and J. Bier, "DSP processors hit the mainstream," *IEEE Comput.*, vol. 31, pp. 51–59, Aug. 1998.

[22] J. Eyre and J. Bier, "The evolution of DSP processors: From early architecture to the latest developments," *IEEE Signal Process. Mag.*, vol. 17, pp. 44–51, Mar. 2000.

[23] F. Irigoin and R. Triolet, "Super-node partitioning," in *Proc. 15th Annu. ACM Symp. Principles Program. Lang.*, San Diego, CA, Jan. 1988, pp. 319–329.

[24] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee, "Improving locality using loop and data transformations in an integrated framework," in *Proc. Int. Symp. Microarchitect.*, Dallas, TX, Dec. 1998, pp. 285–297.

[25] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye, "Influence of compiler optimizations on system power," in *Proc. 37th Design Automation Conf. (DAC'00)*, Los Angeles, CA, June 2000, pp. 5–9.

[26] I. Kodukula, N. Ahmed, and K. Pingali, "Data-centric multilevel blocking," in *Proc. SIGPLAN Conf. Program. Lang. Design Implement.*, June 1997, pp. 346–357.

[27] M. Lam, E. Rothberg, and M. Wolf, "The cache performance of blocked algorithms," in *Proc. 4th Int. Conf. Architect. Support Program. Lang. Operat. Syst.*, Apr. 1991, pp. 63–74.

[28] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. 30th Annu. Int. Symp. Microarchitect.*, 1997, pp. 330–335.

[29] S.-T. Leung and J. Zahorjan, "Optimizing data locality by array restructuring," Dept. Comput. Sci. Eng., Univ. Washington, Tech. Rep. TR 95-09-01, Sept. 1995.

[30] A. Mandhani, T. Cook, and U. Kremer, "Using Cache as a Local Memory," Dept. Comput. Sci., Rutgers Univ., New Brunswick, NJ, Tech. Rep. LCSR-TR394, Jan. 1999.

[31] S. A. McKee and W. A. Wulf, "Access ordering and memory-conscious cache utilization," in *Proc. 1st Symp. High-Performance Comput. Architect.*, 1995, pp. 253–262.

[32] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving data locality with loop transformations," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 4, pp. 424–453, 1996.

[33] J. M. Crummey, D. Whalley, and K. Kennedy, "Improving memory hierarchy performance for irregular applications," in *Proc. 13th ACM Int. Conf. Supercomput.*, Rhodes, Greece, June 1999, pp. 425–433.

[34] N. Mitchell, L. Carter, and J. Ferrante, "Localizing nonaffine array references," in *Proc. Int. Conf. Parallel Architect. Compilat. Tech.*, Newport Beach, CA, Oct. 1999.

[35] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of compiler algorithm for prefetching," in *Proc. 5th Int. Conf. Architect. Support Program. Lang. Operat. Syst.*, Oct. 1992, pp. 63–72.

[36] M. O'Boyle and P. Knijnenburg, "Integrating loop and data transformations for global optimization," in *Proc. Int. Conf. Parallel Architect. Compilat. Tech.*, Paris, France, Oct. 1998, pp. 12–21.

[37] 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. (ED&TC'97)*, Paris, Mar. 1997.

[38] ——, "Architectural exploration and optimization of local memory in embedded systems," in *Proc. 10th Int. Symp. Syst. Synthesis (ISSS'97)*, Antwerp, Belgium, Sept. 1997.

[39] W.-T. Shiue and C. Chakrabarti, "Memory exploration for low-power embedded systems," in *Proc. Design Automation Conf. (DAC'99)*, New Orleans, LA, 1999, pp. 140–145.

[40] 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. Supercomput.*, 1993, pp. 41–419.

[41] 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. Comput. Architect. (ISCA'00)*, June 2000, pp. 95–106.

[42] L. Wang, W. Tembe, and S. Pande, "Optimizing on-chip memory usage through loop restructuring for embedded processors," in *Proc. 9th Int. Conf. Compiler Construction*, Berlin, Germany, Mar. 30–31, 2000, pp. 141–156.

[43] M. Wolf and M. Lam, "A data locality optimizing algorithm," in *Proc. ACM SIGPLAN'91 Conf. Program. Lang. Design Implement.*, June 1991, pp. 30–44.

[44] M. Wolfe, *High Performance Compilers for Parallel Computing*. Reading, MA: Addison-Wesley, 1996.

Mahmut Kandemir (S'98–A'99) 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.

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

Prof. Kandemir is a member of the ACM.

J. Ramanujam (S'87–M'90) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Madras, India, 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 of electrical and computer engineering at Louisiana State University, Baton Rouge. His research interests are in embedded systems, compilers for high-performance computer systems, software optimizations for low-power computing, high-level hardware synthesis, parallel architectures, and algorithms. He has published over 90 papers in refereed journals and conferences in addition to several book chapters.

Dr. Ramanujam has served on the Program Committees of several conferences and workshops, such as the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'2001), the Workshop on Power Management for Real-Time and Embedded Systems, (IEEE Real-Time Applications Symposium, 2001), the International Conference on Parallel Architectures and Compilation Techniques (PACT'00), the International Symposium on High-Performance Computing (HiPC'99) and the 1997 International Conference on Parallel Processing. He co-organized a workshop on *Compilers and Operating Systems for Low Power*, V, held in conjunction with PACT'01 in October 2001 He co-organized the first workshop in this series as part of PACT 2000 in October 2000. He has taught tutorials on compilers for high-performance computers at several conferences, such as the International Conference on Parallel Processing (1998, 1996), Supercomputing'94, Scalable High-Performance Computing Conference (SHPCC'94), and the International Symposium on Computer Architecture (1993 and 1994). He has been a frequent reviewer for several journals and conferences. He received the National Science Foundation's Young Investigator Award in 1994.

Mary Jane Irwin (S'74–M'77–SM'89–F'94) received the Ph.D. degree in computer science from the University of Illinois, Urbana–Champaign, in 1977.

She has been on the faculty at The Pennsylvania State University since 1977, where she currently holds the title of Distinguished Professor of Computer Science and Engineering. She received an Honorary Doctorate from Chalmers University, Sweden, in 1997 and the Pennsylvania State Engineering Society's Premier Research Award in 2001. Her research and teaching interests include computer architecture, nanotechnology, embedded and mobile computing systems design, low-power design, and electronic design automation.

Prof. Irwin is a Fellow of The Association for Computing Machinery (ACM) and was elected to the National Academy of Engineering. She serves as a member of the Technical Advisory Board of the Army Research Laboratory, and as the Editor-in-Chief of ACM's *Transaction on Design Automation of Electronic Systems*. She has served as an elected member of the Computing Research Association's Board of Directors, the IEEE Computer Society's Board of Governors, and of ACM's Council, Chair of the National Science Foundation's Computer Information Sciences and Engineering Directorate's Advisory Committee, and as Vice President of the ACM.

N. Vijaykrishnan (S'97–A'98) received the B.E. degree in computer science and engineering from SVCE, University of Madras, Madras, India, in 1993 and the Ph.D. degree in computer science and engineering from the University of South Florida, Tampa, in 1998.

Since 1998, he has been with the Computer Science and Engineering Department, The Pennsylvania State University, University Park, where he is currently an Associate Professor. His research/teaching interests are in the areas of energy-aware reliable systems, embedded Java, secure computing, nano/VLSI systems, and computer architecture.

Prof. Vijaykrishnan has received several awards in recognition of his academic achievements including the IEEE CAS VLSI Transactions Best Paper Award in 2002, the Pennsylvania State Computer Science and Engineering Faculty Teaching Award in 2002, the ACM SIGDA Outstanding New Faculty Award in 2000, the Upsilon Pi Epsilon Award for Academic Excellence in 1997, the IEEE Computer Society Richard E. Merwin Award in 1996, and the University of Madras First Rank in Computer Science and Engineering in 1993.

Ismail Kadayif received the B.Sc. degree in control and computer engineering from Istanbul Technical University, Istanbul, Turkey, in 1991 and the M.S. degree in computer science from the Illinois Institute of Technology, Chicago, in 1997. He is currently pursuing the Ph.D. degree in computer science and engineering at the Pennsylvania State University.

He is interested in high-level compiler optimizations, power aware compilation techniques, and low-power architectures.

Amisha Parikh received the B.E. degree in computer engineering from Gujarat University, India, in 1998 and the M.S. degree in computer science from Pennsylvania State University, University Park, in 2001.

Her thesis topic was energy-aware instruction scheduling. She has been working with IBM Corporation, Silicon Valley, CA, since 2001.