Maximizing Data Reuse for Minimizing Memory Space Requirements and Execution Cycles [*]

M. Kandemir, G. Chen, and F. Li
Computer Science and Engineering Department
Pennsylvania State University
e-mail: {kandemir,gchen,feli}@cse.psu.edu

Abstract— Embedded systems in the form of vehicles and mobile devices such as wireless phones, automatic banking machines and new multi-modal devices operate under tight memory and power constraints. Therefore, their performance demands must be balanced very well against their memory space requirements and power consumption. Automatic tools that can optimize for memory space utilization and performance are expected to be increasingly important in the future as increasingly larger portions of embedded designs are being implemented in software. In this paper, we describe a novel optimization framework that can be used in two different ways: (i) deciding a suitable on-chip memory capacity for a given code, and (ii) restructuring the application code to make better use of the available on-chip memory space. While prior proposals have addressed these two questions, the solutions proposed in this paper are very aggressive in extracting and exploiting all data reuse in the application code, restricted only by inherent data dependences.

I. Introduction

Applications running on embedded/mobile systems face very different constraints than their counterparts executing on high-end systems. For example, power consumption, and memory size and form factor limitations can severely limit the datasets that can be handled. While sophisticated packaging techniques help squeeze large datasets in small memories, such techniques are costly and not scalable. Also, one does not have the option of arbitrarily increasing memory size due to power and form factor considerations. Therefore, techniques that reduce memory space requirements of embedded applications and those that can help designers to maximize the utilization of the available memory space are very important. In addition, the fact that the embedded/mobile applications keep increasing in both size and complexity makes the overall memory optimization problem a very challenging one.

Regarding the memory management of embedded devices, there are two critical issues: (i) designing the most suitable on-chip memory configuration and (ii) restructuring the application code to make better use of the available on-chip memory space. In particular, exploiting small, fast, and power-efficient on-chip memories is critical from both performance and power angles. While it is conceivable that a knowledgeable application programmer can restructure her application code for the best memory behavior; in general this is a very hard task and can benefit a lot from automation. Similarly, an automatic tool that helps the designer decide on the on-chip memory configuration (in particular, memory capacity) can be very important. Therefore, recent studies have addressed memory architecture design and automated software optimization for data reuse [4–6, 8, 13, 15, 18]. This paper mainly addresses two important questions:

• What is the minimum on-chip memory capacity (measured in terms of fixed size blocks) that minimizes the frequency and volume of data transfers between off-chip and on-chip memory in a two-level memory hierarchy? This is a particularly important problem when one wants to design an application-specific on-chip memory for a given data-intensive embedded application.

• How can we restructure an application code to make better use of the available memory hierarchy? This is a relevant problem in cases where one wants to adapt the behavior of a given application to an existing memory hierarchy that has both on-chip and off-chip components.

While prior proposals have addressed these two questions, the solutions proposed in this paper are unique because they are very aggressive in extracting and exploiting data reuse, to the extent allowed by intrinsic data dependences. This helps improve the quality of the solution for both the problems posed above. That is, by maximizing data reuse we can make better use of the available memory space and can also reduce the total required storage space. Our approach is based on dividing large data structures into logical blocks and clustering computations that access the same data block using a scheduler. While our current implementation uses a software-managed on-chip memory (e.g., a scratch-pad memory [15]), we believe that the proposed techniques can also be adapted to work with hardware-managed cache memories.

The rest of this paper is organized as follows. The next section presents the details of our approach to determining the minimum on-chip storage capacity for reducing the number of data transfers between off-chip and on-chip memories. Section III discusses our solution to code restructuring for an existing on-chip memory space. Section IV concludes the paper by summarizing our major observations.

II. Determining Minimum On-Chip Memory Capacity

In this section, we present the details of our approach to determining the minimum on-chip memory capacity. While the other aspects of the on-chip memory are also important, in this section, we focus our attention on the memory capacity problem.

[*]This work is supported in part by NSF Career Award #0093082 and by a grant from GSRC.

```
do i = 1, N
  do j = 1, N
    ... = U[i][j]
end do
do k = 1, N
  do l = 1, N
    ... = U[k][l]
end do
```
(a)

```
do t1 = 1, N
  do t2 = 1, N
    ... = U[t1][t2]
    ... = U[t1][t2]
end do
```
(b)

Fig. 1. An example fragment (a) and its transformed version (b).

A. Data-Centric View for Reuse

Our approach focuses on array-based data-intensive embedded applications and operates with the concept of a *data block,* which represents both the minimum amount of data transferred from the off-chip memory in a single transfer and the minimum amount of on-chip storage. We use the term *on-chip block* to refer to the unit (i.e., the building block) for the on-chip storage, which is the same size as a data block. Then, our problem can be expressed as one of determining the optimal number of on-chip blocks for a given embedded application. At the high-level, the goal behind our approach is to determine the minimum on-chip memory size (capacity) in terms of the number of on-chip blocks so that increasing that number will not bring any additional performance benefits. To illustrate the idea let us consider the code fragment in Figure 1(a), where two separate loop nests iterate over a two-dimensional array (U). Assuming that there are no data dependences between the iterations of these loop nests, the entire code fragment can be restructured as shown in Figure 1(b). In this case, the references to the array are brought together; similar to the way a loop fusion algorithm [9] would perform. Now, to execute this fragment, only one on-chip memory block would be sufficient. For example, if we have an on-chip block that can hold K elements, we bring the first K elements of the array to the on-chip block and process them. Since these K elements are not needed for subsequent computation, once we are done with them, we can remove them from the on-chip storage and bring the next group of K elements, and so on. With the transformed code in Figure 1(b), having another on-chip block would not help futher improve performance at all. In contrast, in the original code fragment in Figure 1(a), the successive accesses to a given data block are far apart from each other; in fact, they occur in different nests. Therefore, if one wants to exploit this reuse, a total of N^2/K on-chip blocks are needed (so that each data block can be kept on-chip until its reuse takes place). As a more complex scenario, consider the code fragment in Figure 2(a). While the two nests shown in this example share a lot of data elements (i.e., they exhibit high data reuse), the code fragment needs many on-chip blocks to exploit this data reuse. In comparison, the transformed code depicted in Figure 2(b) – obtained automatically through our approach to be explained in this paper – needs only a single on-chip data block (though, this fact is not very clear from this transformed code description!). Note that, the transformation performed to obtain Figure 2(b) from Figure 2(a) is not a simple application of loop fusion.

In this section of the paper, we want to determine the *minimum* number of on-chip blocks such that increasing this number does not bring any additional performance benefits. Consequently, our approach comes up with the minimum on-chip memory capacity with the high performance, and is thus suit-

```
do i = 1, N
  do j = 1, N
    ... = U[i+j][j+3]
end do
do k = 2, N
  do l = 1, N
    ... = U[k-1][k+l+2]
end do
```
(a)

```
do G = 1, N
  do F = 1, N, K
    do t1 = max(G-N,G-K-F+4,1),
               min(G-1,N,G-F+3)
      ... = U[G][G-t1+3]
    end do
    if (G ≤ N-1 && G ≥ 0) {
      do t1 = max(-G+F-3,1), min(N,-G+K+F-4)
        ... = U[G][G+t1+3]
      end do
    }
  end do
end do
```
(b)

Fig. 2. An example fragment (a) and its transformed version (b).

able from the viewpoint of energy consumption (since a larger on-chip storage would only increase the power consumption without reducing the execution time any further). In doing so, we also modify the application code (re-schedule its loop iterations) as well. However, there exist several issues that make this problem very challenging:

- In general, there can be both intra-loop and inter-loop data dependences that prevent us from reusing a data block brought from the off-chip memory completely before being replaced (by another block). Therefore, one needs a suitable representation of data dependences across different loop nests. Most of the prior research focuses only on intra-loop dependences.
- Since there can be multiple arrays accessed by the application, this can increase the number of on-chip blocks required. However, one can potentially reduce this number by considering the lifetimes of the data blocks of different arrays.
- Array access patterns can in general be very complex. Consequently, the loop transformations required can be much more complex than the simple fusion-like transformation illustrated in Figure 1(b). In fact, the loop transformation used for obtaining the transformed code in Figure 2(b) from the one in Figure 2(a) is not easy to derive using the conventional (linear algebra based) loop transformation theory [14].

In the rest of this section, we present a mathematical model, within which the potential problems posed above can be addressed.

B. Data Block Graph

We assume all array indices and loop bounds are affine functions of enclosing loop indices. Let us assume, without loss of generality, that the program to be optimized has v loop nests, and I_1, I_2, $I_3, \cdots$, I_v denote iteration spaces of these loop nests. Each iteration space is a set that contains the iteration points executed by a loop nest. For example, a loop nest with two loops each iterating N times has a total of N^2 iteration points. We say that these iteration spaces collectively define the *computation domain* of the application; that is:

$$I_D = \bigcup_{1 \leq i \leq v} I_I,$$

where I_D is the computation domain. We use $I_{U,i,j}$ to represent the set of iterations from loop nest i that access data block j of array U. In formal terms, an iteration (point) l belongs to $I_{U,i,j}$ if and only if the following holds:

$$\exists R \text{ and } \exists d \in \text{ data block } j \text{ of } U \text{ such that } R(l) = d,$$

where R is a reference in loop nest i to array U. Note that, we have:

$$\bigcup_i \bigcup_U \bigcup_j I_{U,i,j} = I_D.$$

That is, all $I_{U,i,j}$s when combined together cover the entire computation domain. A data dependence is said to exists between $I_{U,i,j}$ and $I_{U,m,n}$ if an iteration that belongs to $I_{U,m,n}$ depends on (the result generated by) an iteration of $I_{U,i,j}$. A data dependence imposes an execution order for the loop iterations in $I_{U,i,j}$ and $I_{U,m,n}$.

Conceptually, we can use a graph, called *data block graph* (*DBG*), to represent $I_{U,i,j}$ nodes and the data dependence relationships between them. Specifically, each node of this graph corresponds to a $I_{U,i,j}$, and a directed edge from $I_{U,i,j}$ to $I_{U,m,n}$ indicates a data dependence between them (which we extract using well-known techniques [2, 9]). Note that an execution of the computation domain means visiting each node of the corresponding DBG. Any *legal* execution is a traversal of this graph that respects all data dependences between them. That is, if there is a dependence from $I_{U,i,j}$ to $I_{U,m,n}$, the latter can be executed only after the former is finished. The subsections below discuss this scheduling problem in detail.

C. Scheduling Problem

In order to generate code for the application being optimized, the entire computation domain must be covered. As mentioned earlier, this can be achieved by visiting each node of the DBG and by observing data dependences during this traversal. While there may be many different traversal orders that can generate legal (dependence-observing) results, we are interested in ones that enhance *data block reuse;* that is, when a data block is accessed, we want to reuse its contents as much as possible before accessing the next data block. While conventional loop transformations attempt to achieve this by transforming each loop nest individually (and independently of the others) as in the case of linear loop transformations [14], or considering only neighboring nests as in the case of loop fusion [9, 14], in this paper, we go beyond these techniques, and consider the entire computation domain (i.e., the entire application), denoted by I_D, the entire computation domain. Our goal is to extract and exploit much more reuse than what is possible by well-known locality-enhancing techniques such as loop tiling, linear loop transformations, and loop fusion.

It is to be noted that the two nodes in a DBG, $I_{U,i,j}$ and $I_{U,i',j}$ (where $i \neq i'$), exhibit data reuse between them. More specifically, both these nodes access the same block j of array U. Consequently, if one wants to exploit this reuse, these two nodes should be scheduled one after another. However, as mentioned earlier, data dependences should also be accounted for. To simplify the problem, we first focus on the special case (Section D), where we have only one array in the code, and each loop nests operates on this array using an arbitrary number of references. We subsequently discuss how this scheme is extended to the more general case, where we have multiple arrays, each can be accessed by any loop nest using an arbitrary number of references (Section E).

D. Solution for Single Array Case

While it is possible to formulate the scheduling problem using known techniques such as integer linear programming

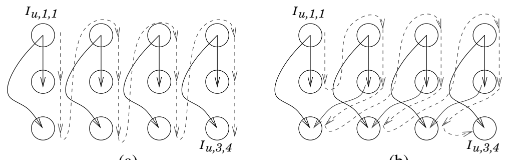

Fig. 3. Two different data block graphs (DBGs) and potential schedulings. The solid arrows denote data dependences, while the dashed arrows indicate scheduling (execution) order. For clarity, only two nodes are explicitly labeled in each DBG.

(ILP) or genetic algorithms (GA), these approaches would be very expensive unless one focuses only on very small-sized graphs/ problems. Therefore, our goal is to come up with a heuristic solution that generates good results most of the time. As mentioned earlier, since DBG nodes $I_{U,i,j}$ and $I_{U,i',j}$ exhibit locality (as they access the same data block), the final schedule should place such nodes into consecutive slots as much as possible. In other words, the access to $I_{U,i,j}$ should be followed by an access to $I_{U,i',j}$. We propose a heuristic scheduling algorithm based on *list scheduling,* a scheduling paradigm used in the past by optimizing compilers [14] and high-level synthesis tools [7].

Our approach is a greedy heuristic that selects one node from the DBG at a time and schedules it. In selecting the next node to schedule, the proposed algorithm observes the following two rules: (1) all the DBG nodes on which this node depends must be already scheduled (data dependence constraint), and (2) this node should access the same data block as the previous node if possible (data reuse constraint). Note that, while breaking the first rule would lead to incorrect execution, breaking the second rule would reduce data reuse. If, at any step during scheduling, the algorithm could not satisfy the second rule, that means we are moving to another data block. This may happen due to two reasons. First, it is possible that all the nodes that access the current data block have already been scheduled. This is the preferable case as this means we can now use the same on-chip block for another data block, i.e., we do not need a new on-chip block for the new data block that needs to be accessed. The second potential reason is that, although we still have some unscheduled nodes that access the current data block, we cannot schedule any of them as the next node due to data dependences. As opposed to the previous one, in this case, we need another on-chip block if we do not want to incur any performance degradation.[1] Each time this second case occurs, we increment the number of on-chip blocks by 1. The algorithm terminates when all the DBG nodes have been scheduled. When this happens, the current number of on-chip blocks gives us the total minimum on-chip memory capacity required. However, during scheduling, whenever the last node that accesses a data block is scheduled, the on-chip block reserved for it is deallocated (recycled). Notice that, our approach takes care of data dependences both across the different loop nests and across the different iterations of the same loop nest.

We consider the following code fragment to illustrate how our approach schedules the nodes of a DBG. The DBG cor-

[1]The alternative would be displacing the current data block from the on-chip block, and placing the new data block into it. However, this incurs performance penalty when the current block needs to be reused in the future. Recall that our objective in this section is to determine the minimum number for on-chip blocks that generate the best performance.

responding to this code fragment is given in Figure 3(a), assuming (for illustration purposes) that each array occupies four data blocks.

```
do i = 1, N
  U[i] = ...
end do
do j = 1, N
  ... = U[j]
end do
do k = 1, N
  ... = U[k]
end do
```

The solid arrows in Figure 3(a) indicate the data dependences between the nodes. A possible schedule determined by our approach is also shown in the figure using dashed arrows. This schedule finishes one data block before moving to the next one. Consequently, it schedules the entire graph using only a single on-chip block. This on-chip block holds each data block in turn, and when a data block is resident in it, all the 3 nodes (coming from the different nests) are scheduled one after another. In other words, in this particular case, we achieve perfect data block reuse, i.e., even if we have more on-chip blocks, we could not achieve a better performance. To illustrate the impact of data dependences in preventing the scheduling from exploiting the maximum data locality, let us now consider the DBG in Figure 3(b). This DBG is similar to the one in (a), the difference being the additional dependence arcs entering into the nodes that belong to the third nest. These dependences prevent us from scheduling the third node (that accesses the same data block) right after the first two nodes. Consequently, one possible schedule is the one shown in the figure (using dashed lines/curves). To demonstrate our approach, let us trace the initial portion of the schedule shown in Figure 3(b). We use OB1, OB2, ..., etc to denote the on-chip blocks used. We start with $I_{U,1,1}$ and assign it (actually the data accessed by it) to OB1. Next, we move to $I_{U,2,1}$ and still operate on OB1. However, after this, we cannot execute $I_{U,3,1}$ from OB1 at this moment due to the data dependence. Instead, we access $I_{U,1,2}$ and assign it to OB2. We subsequently proceed with $I_{U,2,2}$ and continue to use OB2. Once we are done with $I_{U,2,2}$, we can execute $I_{U,3,1}$ using OB1. Now, since all the nodes that access the first data block of the array have been processed, we can discard the contents of OB1 (that is, it can be overwritten/recycled). So, we now use OB1 to load the third data block and schedule $I_{U,1,3}$ and $I_{U,2,3}$, at which point $I_{U,3,2}$ can get scheduled, and so on. To sum up, in this example, with only two on-chip blocks we can schedule the entire DBG and the total number of off-chip memory accesses (on-chip block loads/updates) is 4. Increasing the number of on-chip blocks further would not bring any performance benefits. For example, even if we have 4 on-chip blocks, we would need 4 off-chip memory accesses. We discuss in Section III how such graphs can be scheduled with a given (fixed) number of on-chip blocks.

E. Solution for Multiple Array Case

In this paper we explore two different methods for handling the multiple array case. The first method, called *single array centric,* is based on two observations: (i) although an embedded application can access multiple arrays, there is typically one or two arrays whose accesses constitute a large fraction of overall memory accesses. (ii) Given a schedule in the data block graph, it is possible to estimate the number of on-chip blocks required to minimize the number of off-chip memory accesses (on-chip block updates). This is possible as the compiler can determine the data access patterns and on-chip block updates easily for array-based embedded applications. Based on these two observations, the single array centric method considers each array in the application in turn. When considering an array, it builds the DBG for this array (omitting the other arrays in the application), and performs the scheduling explained above for the single array case. After the scheduling, we compute the number of on-chip blocks required to minimize the number of on-chip block updates. The important point here is that, in determining this number (denoted L_u for array U), the accesses to other arrays are also accounted for. It is to be noted that this process restructures the entire computation around a single array. It is repeated for each array, and array V that generates the minimum L_v among all alternatives is selected to be the one, around which the application code (the computation domain) is restructured and the output code is generated. The main advantage of this method is its simplicity since it makes use of the scheme explained in the single array case as a subcomponent. Its main drawback is that it fails to capture the coupling between references to different arrays, and this can lead to inefficiencies for applications with more than a single dominant array.

Our second method, called *global,* tries to capture the interaction between different arrays by using a separate DBG for each array. Specifically, this method operates with a *combined DBG* (or CDBG for short), which is constructed as follows. First, we build a DBG for each array. Then, we add some extra edges to connect these DBGs. These extra edges, referred to as *locality edges,* indicate how the data blocks of different arrays are accessed together (by the same computation). To explain the idea, we first consider the simple code fragment shown below:

```
do i = 1, N
  U[i] = V[N-i]
end do
```

Let $I_{U,i,1}$ denote the DBG node that contains the iterations that access the first data block for array U, i.e., the block that contains the elements $U[1]$, $U[2]$, ..., $U[K]$, assuming that the an on-chip block can hold K elements. Note that, when these elements of array U are accessed by $I_{U,i,1}$, the elements $V[N-1]$, $V[N-2]$, ..., $V[N-K+1]$ are accessed from array V. Further, let us assume that there is a node $I_{V,i,f}$ in the DBG for array V that accesses these array elements. Consequently, in our CDBG, we put an (undirected) locality edge between nodes $I_{U,i,1}$ and $I_{V,i,f}$. In a sense, this edge should be visited whenever $I_{U,i,1}$ (or $I_{V,i,f}$) is accessed. After building CDBG, the goal of the global method is to come up with a scheduling such that the current contents of the on-chip blocks are maintained (i.e., reused) as much as possible. It should be noted that, in a sense, the locality edges impose constraints (for scheduling) similar to those imposed by dependence edges.

To illustrate how the global method works, let us consider the code fragment shown below, assuming for clarity that both arrays have two data blocks:

```
do i = 1, N
  U[i] = V[N+1-i] + ...
end do
do i = 1, N
```

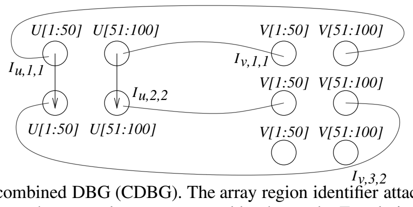

Fig. 4. A combined DBG (CDBG). The array region identifier attached to a node indicates the array elements accessed by that node. For clarity, only four nodes are labeled.

```
  ... = V[i] + ...
end do
do i = 1, N
  ... = V[N+1-i] + ...
end do
```

The CDBG for this fragment is illustrated in Figure 4. Attached to each node are the elements of the data block accessed. The locality edges are between the DBGs of the two arrays. Now, let us trace the scheduling using this CDBG. We start with $I_{U,1,1}$ and assign it to OB1. As a result, the locality edge forces us to assign $I_{V,1,2}$ to OB2. We next move to $I_{U,2,1}$ which reuses OB1. We can now schedule $I_{V,2,2}$ and $I_{V,3,2}$ as well. Since, at this point, all accesses to the first block of U and the second block of V are completed, OB1 and OB2 can be deallocated. In the remaining part of the schedule, we assign OB1 to the second block of U and OB2 to the first block of V. Thus, we can schedule the entire CDBG with only two on-chip blocks.

F. Code Generation

Once the DBG (or CDBG) has been scheduled (i.e., an order in which its nodes are to be traversed has been determined) as explained above, the next task is *code generation.* To do this, we propose to use a polyhedral tool such as the Omega Library [11]. The Omega library manipulates integer tuple relations and sets, which are described using Presburger formulas, a class of logical formulas which can be built from affine constraints over integer variables, the logical connectives ($\vee$, $\wedge$, and $\neg$), and the existential and universal quantifiers ($\exists$ and $\forall$). In our context, the Omega library is used for generating a loop that iterates over the elements (iteration points) that access a data block; that is, the iterations that constitute a node in the DBG. For example, let us consider an array reference $U[i+j-1][j+k+3]$ that appears within a nest with three loops (i, j, and k from outermost to innermost). Assume further that LB_i and UB_i denotes the lower and upper bounds, respectively, for loop index i, and similar lower/upper bounds exist for the remaining two loops as well. The iteration space of this loop nest can thus be described as:

$$\begin{aligned} IS &= \{[i,j,k] : (LB_i \leq i \leq UB_i) \wedge (LB_j \leq j \leq UB_j) \\ &\wedge \quad (LB_k \leq k \leq UB_k)\}. \end{aligned}$$

Let us focus on a particular data block that contains array elements $U[G][F], U[G][F+1], U[G][F+2], \ldots, U[G][F+K-1]$. Within the Omega framework, this block can be defined as:

$$DB = \{[a,b] : (a = G) \wedge (F \leq b \leq F+K-1) \wedge ([a:b] \in DS)\}.$$

Here, DS represents the data (array) space, i.e., the elements of the data block should be within the array bounds. Then, the iterations in the DBG node that access this block can be expressed as the following Presburger formulation:

$$\begin{aligned} ND \quad = \quad & \{[i,j,k] : \exists a \exists b \text{ such that } (a = i+j-1) \\ & \wedge (b = j+k+3) \wedge ([i,j,k] \in IS) \wedge ([a,b] \in DB)\}. \end{aligned}$$

The last condition here forces the accessed element to be within the data block, and the condition before the last one makes sure that any iteration included in ND is legal, i.e., within the loop boundaries. Since an array has typically multiple data blocks, to iterate over all of them in sequence, we can construct the following code (assuming that K – the number of elements in an on-chip block – divides N evenly and omitting the condition on array bounds). In this loop nest, which has been generated with the help of the "codegen" utility in the Omega library, the first two loops iterate over the blocks of the array, whereas the inner two loops visit the iterations that access a given data block (indexed by the upper two loops).

```
do G = 1, N
  do F = 1, N, K
    if (K ≥ 1 && UB3 ≥ LB3) {
      do t1 = max(G-F-K+LB3+5,LB1,G-UB2+1),
              min(G+UB3-F+4,UB1,G-LB2+1)
        do t3 = max(LB3,-G+t1+F-4),
                min(UB3,-G+t1+F+K-5)
          ...U[G,G-t1+t3+4]...
        end do
      end do
    }
  end do
end do
```

III. Scheduling under Capacity Constraints

In Section II, our main focus was on determining the minimum number of on-chip blocks such that the number of on-chip block updates is minimized (i.e., the memory capacity problem). This is an important problem if one targets at determining the minimum capacity of the on-chip storage for extracting the best performance from a given embedded application. Another important question that needs to be answered, though, is how can we schedule a DBG with a fixed number of on-chip blocks (i.e., fixed storage capacity)? Obviously, this problem is more relevant in cases where the on-chip memory capacity is fixed. Our objective in this case is to minimize the number of on-chip block loads/updates as much as possible.

We attack this (scheduling) problem by adopting a greedy strategy, which works as follows. Let us assume that we have r on-chip blocks of equal sizes, and that we have only one array. In the first step of our approach, we determine a set of *chains* in the DBG such that each chain consists of nodes that access the same data block. Let us assume that the number of such chains is t. If $r \geq t$, that is, the number of on-chip blocks is larger than the number of chains, we assign a private on-chip block to each chain. Note that, with such an assignment, the contents of an on-chip block are updated only for the initial load. Therefore, the total cost of such an assignment is t. Consider for example, the DBG depicted in Figure 5(a). We have five six chains in this DBG, i.e., t = 6. Assuming that we have six on-chip blocks (i.e., r = 6), we assign a private on-chip block per chain. Consequently, the DBG can be easily scheduled with only 6 (on-chip block) loads; i.e., each OB is reserved for a data block. The difficult scenario occurs when $r < t$. In this case, we still assign one on-chip block per chain. However, an on-chip block is *reassigned* to another chain (temporarily or permanently) during scheduling. More specifically, when $r < t$, we need to consider two cases:

• *When the first node of a chain is scheduled, all the remaining nodes can also be scheduled.* In other words, there are either no any inter-chain data dependences, or the existing inter-chain dependences are such that all the chains can be scheduled one after another. Consider the DBG in Figure 5(b), which is the same in Figure 5(a). If we only have 3 on-chip blocks (r = 3), the different chains need to share on-chip blocks. In this particular example, however, all the chains are independent of each other. Consequently, we can schedule the chains one after another using only 1 on-chip block. In this case, the total number of loads is 6. Even if we try to use all three on-chip blocks, the total number of loads would not change.

• *Due to inter-chain dependences, all the nodes in a given chain cannot be scheduled one after another.* In this case, we use a greedy heuristic. More specifically, we start with a chain and try to schedule all the nodes (in that chain) one after another until it is not possible to proceed further due to dependences. When this occurs, we select another node among the schedulable ones, and schedule it using another on-chip block if an unoccupied one is available. If not, we select an on-chip block among the ones that are already occupied (by another chain), and use it for the currently required block. While selecting this victim on-chip block can be done using different algorithms, in this paper we use a simple LRU-based one. That is, we reassign the on-chip block that has not been touched for the longest duration of time. This process is repeated until all the nodes in the DBG are scheduled. Figure 5(c) illustrates such a scenario with r = 2. The total number of on-chip block loads is 6. In contrast, assuming that r is 1, we would need a total of 12 on-chip block loads/updates.

This approach can also be extended to the multiple array case. While it is conceivable that both the methods described in Section E can be adapted to work for this case as well, our current implementation uses only the first method, i.e., the one tries each array in turn, and selects the best one to restructure the entire computation (single array centric). Implementing the global method and comparing it with the single array centric method are in our future agenda as such an implementation is more complex.

It should be noted that, the approach presented in this section is very different from the one discussed in Section II. This is because in the on-chip memory capacity problem our objective is to determine the *minimum* number of on-chip blocks. Therefore, during the scheduling process, we *increment* the number of on-chip blocks when needed. In contrast, in the scheduling problem addressed in this section, our objective is to schedule the DBG with a *fixed* number of on-chip blocks (i.e., under fixed capacity). Therefore, when we are short of on-chip blocks, we need to vacate one that is currently in use to open space for the new incoming block.

IV. Conclusion

We witness an unprecedented proliferation of embedded/mobile applications. Most of the embedded environments that execute these applications have severe power, performance, and memory space constraints that need to be accounted for. This paper presents an optimization framework that maximizes data reuse to the greatest extent possible, by restructuring the application code based on data access patterns. The proposed infrastructure can be used for (i) deciding the capacity for on-chip storage and (ii) effective use of the available on-chip memory space.

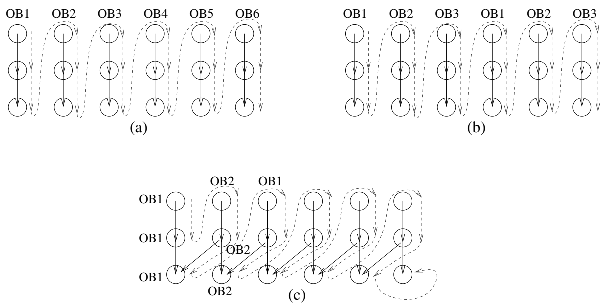

Fig. 5. Three different schedulings. The solid arrows denote data dependences, while the dashed arrows indicate scheduling (execution) order. OBi denotes the ith on-chip block. In (c), the nodes in the initial portion of the schedule are marked with the associated on-chip block.

References

[1] T. M. Austin. The SimpleScalar/ARM Toolset. http://www.eecs.umich.edu/∼taustin/simplescalar

[2] S. P. Amarasinghe et al. The SUIF Compiler for Scalable Parallel Machines. In Proc. the 7th SIAM Conf. on Parallel Proc. for Sci. Comp., 1995.

[3] F. Balasa et al. Background memory area estimation for multidimensional signal processing systems. IEEE Transactions on VLSI Systems, 3(2):157-172, 1995.

[4] L. Benini et al. Increasing Energy Efficiency of Embedded Systems by Application-Specific Memory Hierarchy Generation. IEEE Design & Test, 2000.

[5] E. Brockmeyer et al. Layer Assignment Techniques for Low Energy in Multi-Layered Memory Organizations. In Proc. Design Automation and Test in Europe, Messe Munich, Germany, 2003.

[6] E. Brockmeyer et al. Low Power Memory Storage and Transfer Organization for the MPEG-4 Full Pel Motion Estimation on a Multimedia Processor. IEEE Trans. on Multimedia, 1(2): 202–216, 1999.

[7] G. De Micheli. High-level synthesis of digital circuits. Advances in Computers, 37: 207–283, 1993.

[8] E. G. Hallnor and S. K. Reinhardt. A fully-associative software-managed cache design. In Proc. International Conference on Computer Architecture, pp. 107–116, Vancouver, British Columbia, Canada, 2000.

[9] K. Kennedy and K. McKinley. Optimizing for Parallelism and Data Locality. In Proc. the 1992 ACM International Conference on Supercomputing.

[10] I. Kodukula et al. Data-Centric Multi-Level Blocking. In Proc. ACM PLDI, 1997.

[11] W. Kelly and W. Pugh. Finding Legal Reordering Transformations Using Mappings. In Proc. LCPC, pp. 107–124, 1994.

[12] S.-T. Leung and J. Zahorjan. Optimizing Data Locality by Array Restructuring. Technical Report TR 95–09–01, Dept. of Computer Science and Engineering, University of Washington, September 1995.

[13] M. Miranda et al. Systematic Speed-Power Memory Data Layout Exploration for Cache Controlled Embedded multimedia applications. In Proc. ISSS'01, Montreal, 2001.

[14] S. S. Muchnick. Advanced Compiler Design and Implementation. Morgan-Kaufmann Publishers, San Fransisco, CA, 1997.

[15] P. R. Panda et al. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In Proc. the European conference on Design and Test, 1997.

[16] G. Rivera and C.-W. Tseng. Data Transformations for Eliminating Conflict Misses. In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, Montreal, Canada, June 1998.

[17] X. Vera et al. A Fast and Accurate Framework to Analyze and Optimize Cache Memory Behavior. ACM Transactions on Programming Languages and Systems, March 2004.

[18] L. Wang et al. Optimizing On-Chip Memory Usage through Loop Restructuring for Embedded Processors. In Proc. 9th International Conference on Compiler Construction, March 30–31 2000, pp. 141–156, Berlin, Germany.