논문 2011-06-44

Efficient Use of On-chip Memory through Profile-Driven Array Reorganization

Doosan Cho, Jonghee Youn*

Abstract : In high performance embedded systems, the use of multiple on-chip memories is an essential architectural feature for exploiting inherent parallelism in multimedia applications. This feature allows multiple data accesses to be executed in parallel. However, it remains difficult to effectively exploit of multiple on-chip memories. The successful use of this architecture strongly depends on how to efficiently detect and exploit memory parallelism in target applications. In this paper, we propose a technique based on a linear array access descriptor [1], which is generated from profiled data, to detect and exploit memory parallelism. The proposed technique tackles an array reorganization problem to maximize memory parallelism in multimedia applications. We present preliminary experiments applying the proposed technique onto a representative coarse grained reconfigurable array processor (CGRA) with multimedia kernel codes. Our experimental results demonstrate that our technique optimizes data placement by putting independent data on separate storage. The results exhibit 9.8% higher performance on average compared to the existing method.

Keywords : Compiler, Memory Hierarchy, Execution time, Data Placement

Ⅰ. Introduction

As embedded systems grow more complex and large to satisfy diverse demands from the market, the processor-memory speed gap is becoming a critical design issue. Since the increase in memory access speed has not kept up with increases in processor speed, memory access contention has increased, resulting in a longer memory access latency in systems today. This makes the memory access cost much greater than the computation cost. Thus, improvement in memory performance is critical to the successful use of embedded systems.

To improve the overall performance, many DSPs employ Harvard architecture, which provides simultaneous accesses to separate on-chip memory modules for instructions and data [2, 3]. Some DSP processors are further equipped with multiple data-memory banks that are accessible in parallel, such as Motorola 56000 [2] and Gepard Core DSPs [3, 4]. Since data can be partitioned and allocated to separate data-memory banks and can be accessed simultaneously, the multiple data-memory bank architecture offers potentially higher memory bandwidth and, thus, improves the system performance. This architectural feature is very attractive for high-performance DSP applications. In fact, many DSP routines, such as finite impulse response (FIR) filters, require the convolution of multiple data arrays as a kernel operation. Processors with multiple memory banks can

* Corresponding Author
Manuscript received : 2011. 05. 04.,
Revised : 2011. 06. 21., 2011. 07. 04.,
Accepted : 2011. 07. 07.
Doosan Cho : Sunchon National Univ.
Jonhhee Youn : Gangneung-Wonju National University

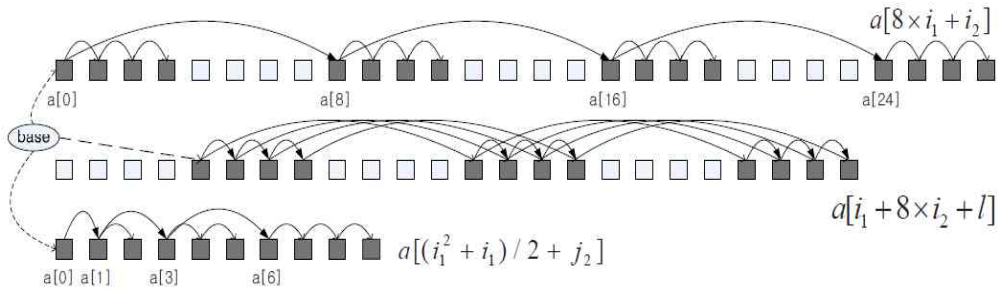


Fig. 1. Gray boxes represent the array elements accessed and arrows with black heads and white heads keep track of the access driven by indices i_1 and i_2 (or j_2), respectively. The base indicates the offset of the first accesses from the beginning of the array.

achieve higher memory bandwidth for this kind of application. However, many existing high level language compilers cannot exploit the advantage of multiple data-memory banks effectively.

As an example in Figure 1, the accessed region of array a by the references a[i_1 × 8+ i_2] and a[i_1 +8× i_2 +1] is not overlapped. Concurrent accesses of the references is desirable. However, a traditional naive approach places the array into a single memory, resulting into serialization of the references. Theoretically, it would seem that assigning a separate memory to each independently accessed region of the array would optimize the performance, but in practice, data dependences limit the amount of parallelism, resulting in no significant performance gain from an arbitrarily large number of memories.

In Figure 1, the non-affine subscripts often prevent to analyze the dependence relation with affine references. It is hard to detect parallelism of array references with such complex references through traditional approaches. To overcome this difficulty, we propose a technique to parallelize affine array accesses considering dependence relation with non-affine references. For simplicity of presentation, none of the algorithms are designed to directly handle non-affine subscript expressions. The candidates of our approach is the arrays with affine subscript expressions. It is important consideration since most audio and video processing and multimedia program consists of affine and non-affine references.

Specifically, we present an algorithm that performs an array reorganization for exploiting multiple memories driven by profile information, so that the array is mapped to memory according to access patterns in the code. We also describe how to reorganize the array in memory from the standard layout in a single memory to the optimized layout in multiple on-chip memories. We present a comprehensive set of performance results, derived automatically by our compiler for several multimedia kernels.

The organization of the paper is the following. The next section describes an architecture model which motivates the proposed approach. Section 3 presents the overview of the array reorganization algorithm. Section 4 describes the analyses and transformations to identify the parallel memory accesses. Section 5 describes how to map array partitions to a limited number of physical memories. Section 6 describes how we re-organize array data from/to a naive layout in a single memory to/from a reorganized layout in multiple memories. Section 7 presents a set of experimental results derived automatically by our compiler. We survey

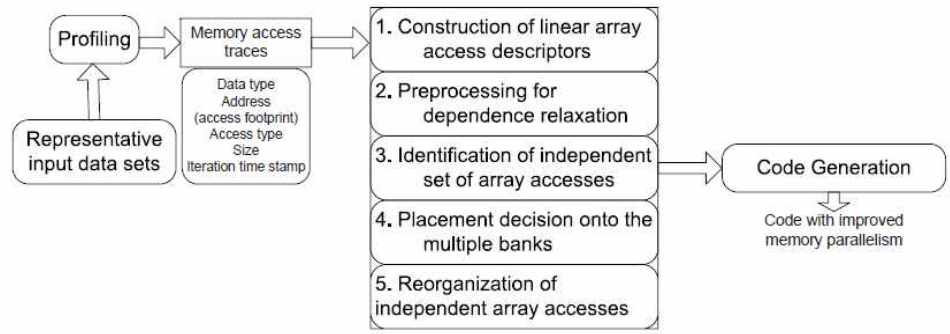


Fig. 2. Workflow of the proposed approach

related work in Section 8. In Section 9 we present conclusions.

II. Background on CGRAs

The main components of CGRA include the PE (Processing Element) array and the local memory. The PE array is a 2D array of possibly heterogeneous PEs connected with a mesh interconnect, though the exact topology and the interconnects are architecture dependent.

The local memory of a CGRA is typically a high speed, high bandwidth, highly predictable random access memory that provides temporary storage space for array data, which are often input/output of loops that are mapped to CGRAs. To provide high bandwidth, local memories are often organized in multiple banks. For instance the MorphoSys architecture [5] has 16 banks, every two of which may be accessed exclusively by each row of PEs (there are eight rows in total).

In our architecture model, it has a fundamental restriction that a bank cannot be accessed by more than two different PEs at the same time, if the bank consists of two port cells. (In the rest of the paper we assume that a bank consist of two read port and single write port cells, and thus has three ports.)

If more than two PEs try to access the same bank at the same time, a bank conflict occurs. CGRA's communication architecture must detect such a bank conflict and resolve it by generating

a stall. Hardware stall ensures that all the requests from different PEs are serviced sequentially, but is very expensive because most of the PEs will be idle during stall cycles. This can be solved by a compiler approach, where compiler makes sure that this does not happen. This paper develops such technique and show that it is promising.

III. Overall Workflow

In order to efficiently solve the array partitioning and mapping problem, we formulate them into two problems as itself. The first problem is to partition arrays within in-dependently accessed elements. The goal of the first problem is to improve memory parallelism. The second problem is to find an optimal layout of partitioned arrays in the multiple memory modules to maximize parallel array accesses.

In the preparation of our approach, it is performed to normalize a loop iteration step size, which involves replacing all the instances of the loop index variable i with $s \times i$, where s is the step size of loop l. Loop normalization is always legal.

The workflow of our approach is shown in Figure 2. The first step is to gather array access footprint through profiling. In profiling task, we have several times run our

benchmark code with various types of input set to gather array access footprints. The profiling information includes several data such as data type, data location, accessed time stamp. By using profiling information, linear array access descriptors are created to summary all array accesses. And then, array reference partitioning and memory mapping are performed by using the array access descriptors. In array partitioning step, we divide a set of array references into partitions that are accessed orthogonal region of the whole array, and map each partition to a separate memory. Assuming arrays accessed within their bounds, if two array references access mutually exclusive array indices in at least one dimension, they access independent array elements. In this case, we put them in separate partitions. Otherwise, we put them in the same partition, and derive a single unified data layout for them based on their common data access patterns. To maximize the opportunities of parallel memory accesses, we create as many partitions as possible.

In the next step, array mapping is performed to determine placement of partitioned arrays in multiple memory modules. Using a formal metric that considers profit to parallelize array accesses and overhead to transfer replicated array partitions, candidates of array partitions to be copied to the best placement are determined for a loop. In this step, physical mapping, the compiler binds each array partition to physical memory, taking into account memory access conflicts based on the array access order in the program, to exploit both memory access and instruction level parallelism.

After that, the compiler rewrites each array reference so that the transformed subscript expression takes into account the position within the newly formed array in the mapped memory. Based on the data access patterns of the code, we insert array distribution/gathering code to/from multiple memories. Finally, optimized code with the array partitions are generated. The following subsections describe this workflow in detail.

IV. The Proposed Approach: Array Partitioning

1. Summarizing Data Access Patterns

Prior techniques for array access analysis sometimes fail because they are unable to recognize some hidden simple access regions shown in Figure 1. To overcome this limitation, we use a linear array access descriptor, which is developed from [1], generated from memory access footprints. It is designed to represent the access pattern precisely and enable analysis techniques to expose the simplicity of array access patterns, thus improving the memory parallelism.

A linear array access descriptor is described by the triple start + [stride, span]. The start is the offset, from the first element of the array, for the first location accessed. A dimension is a movement through memory with a consistent stride and a computable number of steps. The stride gives the distance between two consecutive array accesses in one dimension. The span is the distance (in memory units) between the offsets of the first and last elements that are accessed in one dimension.

For example, an access footprint of 0,8,16, ..., 80 will be described by a linear array access descriptor as 0+ [8,80]. If the array is accessed in a two level nested loop, which the outer loop has step 1 and span 5, the descriptor might be a multi-dimension pattern like 0+ [1,5][8,80]. This has several attractive properties. It is simple and fast, and it works quite well if data accesses have predominantly linear patterns. We notice that, in practice, a significant portion of instructions do exhibit linear access behavior and hence can be captured by a small number of descriptors. We use the descriptor to describe both array partitioning and memory mapping. By unifying

the internal representation of partitioning and mapping, we can facilitate the memory parallelism in an efficient and precise way.

2. Preprocessing for eliminating unnecessry dependences

In this subsection we describe array replication techniques to enable compilers to uncover parallelism opportunities in loop computations that are traditionally impeded by both anti and output-dependences. We focus on partial array replication across loop iterations and the same loop iteration. When two computations, that execute serially, access the same array location, reading its previous value and then writing a new value into the location, this gives rise to an anti-dependence between them. Similarly when two computation use the same location to store consecutive values that are otherwise independent creates an output-dependence. These dependences can be eliminated by creating a copy of the partial array, that each computation freely accesses. Each computation uses a distinct memory location to write and read a value, and in the absence of true-dependences between these loops nest, they can execute concurrently. Therefore, we replicate repeatedly accessed array locations (or overlapped regions) to make both array references independent, thus, each array reference can be mapped to separate memory bank.

This partial array replication technique explores a space-time tradeoff. In order to eliminate anti-, output-and input-dependences, the implementation requires additional memory space. In addition, some execution time overhead is incurred in updating the copies to enforce the original program data dependences. The analysis abstractions, in cooperation with estimates of memory space usage, allow for an effective algorithm to manage this tradeoff and adjust, possibly dynamically, the performance of the implementation in response to available resources.

In this section we describe such replication procedure by intersecting both linear array access descriptors. Intersecting two arbitrary array access descriptors is very complex and probably intractable. But if two array descriptors have the same strides, or the strides of one are a subset of the strides of the other, which has been quite often true in our experiments, then they are similar enough to make the intersection algorithm tractable.

Input: linear array access descriptors A1=start1+[s1, span1], A2=start2+[s2, span2]

Procedure Intersection:
if start2 − start1 > 0 then
if start2 − start1 − span1 > 0 then return {A}; fi
l' = 0; l = $\left\lfloor \frac{start2 - start1 - 1}{s1} \right\rfloor \cdot s1 + s1$ **;**
Non-overlap = {start1+[s1, l − s1]};
else
if start1 − start2 − span2 > 0 then return {A}; fi
l=0; l'= $\left\lfloor \frac{start1 - start2 - 1}{s2} \right\rfloor \cdot s2 + s2$ **;**
fi
if start2 + span2 − start1 − span1 $\geq$ **0 then**
u = span1; u' = span2 − s2 - $\left\lfloor \frac{start2 - span2 - start1 - span1 - 1}{s2} \right\rfloor \cdot s2$ **;**
Non-overlap = Non-overlap $\cup$ **{start1 + u + s1 + [s1, span1 − s1 - u]};**
else
u' = span2; u = span1 − s1 - $\left\lfloor \frac{start1 - span1 - start2 - span2 - 1}{s1} \right\rfloor \cdot s1$ **;**
fi
if u − l < 0 or u' - l' < 0 then return Non-overlap; fi
S_{LC} **= Least Common Multiple (s1, s2)**
Sub1 = sub-region descriptors of start1 + l + [S_{LC}**, u - l];**
Sub2 = sub-region descriptors of start2 + l' + [S_{LC}**, u' - l'];**
Overlap = Common_descriptors(Sub1, Sub2);
return Overlap;

Fig. 3. The algorithm for finding the intersection of both linear array descriptors

To illustrate intersection with two array access pattern descriptors, we present in Figure 3 one simple intersection algorithm, which accepts two descriptors A and A', and produces a set of array access pattern descriptors that summarize the array regions represented by A ∩ A'. The output is a linear array access pattern descriptor set Over-lap. An array descriptor set called Non-overlap represents the area of A that does not overlap the area of A'. To compute Non-overlap, the algorithm compares the left ends and right ends of the regions represented by A and A'.

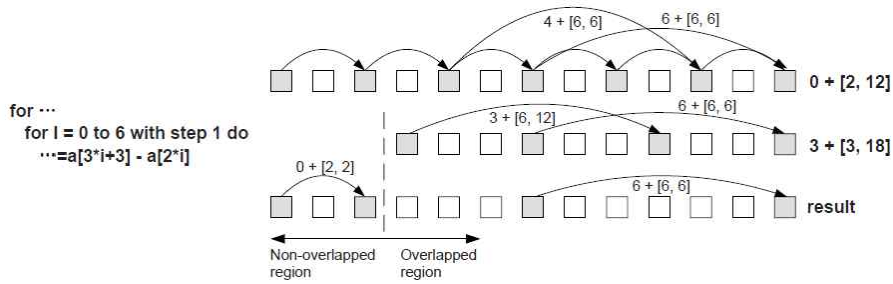


Fig. 4. Intersection of the two references

For this, the offset distance between the two access descriptors needs to be calculated. If there is an area of overlap, then the area is removed from A. The remaining elements in A form the set Non-overlap. Since all descriptors in Non-overlap can be part of the result, they could be combined in the final output later when the computation on Overlap is complete.

To compute Overlap, the algorithm first finds a number Θ, which is the least common multiple of the strides of both A and A', and forms two sets of sub-region access descriptors, respectively, of A and A' with Θ as their strides. Then, it finds which of the sub-region descriptors in the two sets access the same elements within Overlap. Note that a member of the sub-region descriptors in an input descriptor always represents a subset of the descriptor. Thus, we find a pair of sub-region descriptors, one from each set, which a reseparated by a distance equal to a multiple of Θ. All those remaining sub-region descriptors of A constitute the set Overlap.

To explain this algorithm with an example, consider the code in Figure 4. In order to partition the array a within the loop of Figure 4, we would have to identify what region of the array a is overlapped. For the code in Figure 4, we would need to perform 0+[2,12] ∩ 3+[3,18], then determine whether the result is empty. According to the intersection algorithm, the nonoverlapping area in 0+[2,12] should be first found, as shown in Figure 4:

Non-overlap =0+[2, 2].

In the overlapping area, we intersect 4+[2,12], which is the subregion of 0+[2,12] in the area, with 3+[3,18]. For this, we first find the least common multiple (LCM) of the strides of both accesses, LCM(2,3)=6. Then, we calculate a set of sub-region descriptors (which have Θ as a stride) for 4+[2,12], which is S1 = {4+[6,6], 6+[6,6]}, and a set of sub-region descriptors, which have stride= Θ, for 3+[3,12], which is S2 = {3+[6,12], 6+[6,6]}. By intersecting S2 with S1, we can obtain the results in the overlapping area. The operation S1 ∩ S2 is straightforward since they have a common stride 6; that is, it can be performed by simply comparing the elements of the sets. This results in:

Overlap =S1 ∩ S2 = {6+[6,6]}.

This process continues until it can either be determined that no overlap occurs, or until the inner-most dimension is reached where it can make the final determination as to whether there is an intersection between the two. A set of descriptors from the intersection procedure is returned, and as each recursion returns, a dimension is added to the results of descriptors.

We calculate the size of overlapped region from the descriptors as a result of the procedure. Based on the size of overlapped region, our approach determine whether it is beneficial to make partially replicated array partitions. This is determined in the placement decision step described in the next section. If our procedure determines to replicate some overlapped region (reused data), then the

descriptors will be manipulated as partitioned arrays in the placement decision step.

3. Partitioning Array References

When two array references access mutually exclusive array elements, and thus there is no data dependence between them, we can put them in separate partitions. For ex-ample, consider array references 0+[4,1000], 1+[4,1000], and 0+[2,1000]. 0+[4,1000] accesses a subset of array elements accessed by 0+[2,1000],but 1+[4,1000] accesses independent array elements. So, we derive a unified data layout for 0+[4,1000] and 0+[2,1000], and a separate data layout for 1+[4,1000]. The following proposition pro-vides a key property for the proposed partitioning algorithm.

Proposition 1. *If two n-dimension array access descriptors A and A′ access independent regions then they can be placed in separate partitions. This condition is represented by the following equation:*

$$(start'_i + span'_i) \bmod S_{GC} \neq (start_i + span_i) \bmod S_{GC}$$

where S_{GC} are the greatest common divider of the both strides, and $start_i$ and $start'_i$ are the offsets associated with two array access descriptors A and A′ for each dimension I.

We prove the proposition by contradiction. Assuming that the descriptors A and A' access dependent regions if the following equation holds:

$$(start'_i + span'_i) \bmod S_{GC} = (start_i + span_i) \bmod S_{GC}$$

Rearranging terms,

$$((start'_i - start_i) + (span'_i - span_i)) \bmod S_{GC} = 0$$

S_{GC} is the common factor of $(start'_i - start_i)$ and $(span'_i - span_i)$. Thus, overlapped region descriptors ⇔

$$(start'_i + span'_i) \bmod S_{GC} = (start_i + span_i) \bmod S_{GC}$$

□

The partitioning algorithm uses Proposition 1 for each array to divide array references into partitions. The algorithm for partitioning the array references in a loop nest is shown in Figure 5.

```
m : total number of array descriptors
Set : a set of partitions; i.e., {P}
P : a set of k independent references to an array
p : an array reference in P

Input: a set of array access descriptors of an array

Procedure Partition(P, m){
 //find independent set
 for i=0 to m {
  for j=0 to m //intersection returns their dependency
   if (Intersection(descriptor_i,descriptor_j))
    independentSet = descriptor_i; }

 //P has the set of independent descriptors
 P = independentSet;
 //strideSet returns all strides of the descriptors in P
 SGC = GCD(strideSet(P));

 for each p in P{
  //the first insertion creates a room to store a new set of
partitions in Set
  //start of p is the room number to distinguish partition groups
  if (SGC == 0) insert p into Setstart_p
  //the results of start_p mod SGC is the room number
  else insert p into Set(start_p mod SGC)
  }

  //needMorePartition returns 1 if it is possible to separate
partitions in Set
  if(needMorePartition(Set)){
   for k=0 to k=theNumberOfPartition(Set)
    Partition(Set(k),numberOfElement(Set(k)));}
  else return Set;
}
```

Fig. 5. Partitioning algorithm

Initially, all array references represented as linear array access descriptors belong to a single partition, Set. Then, procedure Partition separates the descriptors (array references) into different partitions whenever it can prove they are accessing in dependent array elements using Proposition1. Array references within a partition are recursively partitioned according to Proposition1 until no further partitioning is possible. The recursive function Partition derives possible subpartitions Set. If all references are mapped to the same subpartition, then that dimension's references cannot be partitioned, so the algorithm returns the result of partitioning the next dimension. Otherwise, it attempts to further partition a sub-partition according to the current dimension.

For example, consider Set = {0+[2,100], 3+[4,100], 1+[8,100], 5+[8,100]}. The common stride GCD(2, 4, 8, 8) is 2. According to the condition in Proposition 1, Set is divided into two partitions {0+[2,100]}and {3+[4,100], 1+[8,100], 5+[8,100]}. The second partition is further partitioned into two subpartitions {3+[4,100]}and {1+[8,100], 5+[8,100]}, since GCD(4,8,8) = 4 and (3 mod 4) ≠ (1 mod 4) = (5 mod 4). Further, the second subpartition is divided into two subpartitions {1+[8,100]} and {5+[8,100]}. Therefore, all four references in Set access completely independent array elements, and are mapped to different bank in on-chip memories.

Ⅴ. The Proposed Approach: Array Mapping

In our architectural model (CGRA), the only resources that might have conflict are the memory ports. For example, typical architecture permits only one 1-cycle read from any memory per cycle. Two read operations accessing the same memory that appear at the same level of the precedence schedule must be scheduled at different levels of an execution schedule. Such multiple accesses to a memory force serialization of access even if the operations using the data are independent (and thus could be scheduled concurrently).

This concurrent accessing, however, raises the issue of resource contention (at the memory port) when two or more concurrently executing loop nests access the same array region, i.e., the loops exhibit input dependences. To overcome this memory contention, we take advantage of the flexibility of partitioned data placement decision in banked memory architectures. By optimizing placement of pieces of arrays, the parallel loop nests can therefore execute concurrently due to the absence of anti-dependences but also be contention-free. To that end, we use factor of memory access conflict to determine better data placement in banked on-chip memory, since it shows how many concurrent accesses happen into the same memory bank at the same iteration. Factor of memory access conflict is calculated as follow : Conflict(p) =

$\sum$(accessVector(p') ∩ accessVector(p))

where p is a partition and p' is the others of an array. Memory access pattern of an array in a loop is represented as a memory access vector in loop iteration space [6], thus, access Vector of p represents the set of accessing iteration number of loop iterations. It gives the number of conflict from some array partitions placing in the same memory bank. Based on the factor, we prevent large memory access conflict occurred by limit number of memory port. Consequently, we can avoid serialization of memory accessing. This factor used for calculating the profit described in the next subsection.

Memory mapping creates as many memories as needed to maximize opportunities of parallel memory accesses for each array in isolation, and in an architecture dependent way. In this section, we describe how to map array partitions to a limited number of physical memories such that the exposed parallel memory access opportunities are preserved as much as possible.

To map the partitioned arrays to a specific target architecture, we must take the following into account: (1) the number of physical memories or the number of bank of an on-chip memory M_p; (2) competing demands of multiple array partitions. Intuitively, we want to distribute array partitions across M_p physical memory modules as evenly as possible, since it preserves the exposed parallel memory access opportunities, and minimizes the address bits required for each physical memory.

The actual memory operations that can be

scheduled concurrently are affected by the physical memory mapping. We denote P_n as the total number of array partitions across all the arrays in a loop nest. If $P_n \leq M_p$, we distribute each array partition to a different physical memory. If $P_n > M_p$, some array partitions must be mapped to the same physical memory, thereby possibly sacrificing potential memory parallelism.

Some array partitions carry a scheduling constraint such that the operations on the right hand side of an assignment statement must be scheduled before the operations on the left hand side. We map the array partitions that carry the scheduling constraint to the same physical memory to give other less constrained array partitions more freedom to be mapped to separate physical memories.

1. Problem Definition

We describe the problem and solution for a set of memory banks with uniform latency. First of all, we define the problem as follow:

Definition 1. Optimal Array Reorganization Problem:

Objective function: find optimal placement results of the partitioned arrays P_n which maximized the performance gain,

$$\sum_{p \in P_n} (\text{Profit(p)} - \text{Overhead(p)}).$$

The profit and overhead of an allocation of array partitions p_1, ..., p_n to M_p banks for a loop l is given as the following:

- profit: This is the amount of memory access cycle reduction calculated by how many memory accesses can be parallelized, Profit(P).It is calculated by adding a dry-run stage with pre-determined placement with the minimum memory access conflict factor. Array partitions having large factor of the access conflict should be placed in different bank. If it is impossible, it can be permissable that such array partitions with lower conflict factor might be placed in the same bank. Thus, the value of profit is computed by the number of memory access cycle reduction per Conflict(P).
- Overhead: Data transfer overhead is represented by Overhead(P), which gives the increased data transfer overhead by partially replicated array partitions.

The problem is subject to the capacity constraint. It is defined as following: Let the n on-chip memory modules have limited capacities $C=\{c_1,c_2,...,c_n\}$. For a set of assigned array partitions $p \in P_n$, $\sum_{p \in P_n} \text{Size(p)}$ must not exceed the each capacity C of the corresponding memory.

Our approach exploits a greedy approach to effectively seek an optimal placement. The following subsection presents a best first search method for each problem instance, P = p_1, ..., p_k. The problem instance consists of a set of array partitions and candidates of array partitions generated from non-partitioned arrays. Our mapping approach takes two steps. First, a set of array partitions are evenly assigned to physical memories. Second, a set of candidates of array partitions are assigned by the best first search method.

Best First Search: Each problem instance P is an objective of the best first search. The best first search is used to search for an optimal data layout in multiple memory modules.

The search algorithm builds a search tree, and stores at each node the maximum performance gain and the minimum performance gain on the objective function for the problem instance.

The search scheme repeatedly (1) selects an unprocessed array partition, (2) processes the partition and then creates its the best child, and (3) propagates new max and min values through the tree and uses these values to select the next node. It performs this sequence of three stages until the search tree contains no more unprocessed array partitions.

Note that whenever an array partition is observed, its best child is immediately created, producing a search tree. Thus, the search tree's a new leaf is always the child which have the best values. Let us now consider the three major steps in more detail.

The first step is to find the node to process next. The best first search selects a leaf array partition by descending the search tree, starting at the root and taking the child with the best values at unobserved candidate arrays. Our implementation orders the child from left to right so that their values are non-decreasing with a priority queue.

The second step is to process and expand the node. For each of these unobserved nodes, maximum and minimum performance gain on its objective function is obtained, and a best unobserved array partition is chosen to branch on. The node is created and then processed and expanded in the same way. At each step, the set of nodes contributing to the maximum performance gain is stored to a solution set.

The third step is to propagate the new performance gain and prune the tree. Starting at the nodes just created and working up the tree to the root, the value of the maximum performance gain and the minimum performance gain are updated for each node. As this stage assigns and reassigns performance gain, it checks to see if any node has one child whose maximum performance gain does not exceed the minimum performance gain of the other child. In such a case the array partition of maximum can be no better than that of the partition of minimum, so the partition of maximum and all its descendants are removed from the tree. Finally, this search procedure a placement of array partitions as an optimal solution.

Ⅵ. Experiments

We conducted several experiments to assess the effectiveness of our approach. We explored how much our approach influences the performance while minimizing the overhead. The goal of our experiments was to compare our approach for maximizing utilization of the multiple on-chip memories against without the proposed approach for a number of multimedia kernel codes from DSPstones [7], Mediabench [8].

1. Result

The proposed technique is implemented in a commercial C compiler framework, called ICD-C compiler from Dortmund University [9]. The compiler flag O3 is used, but loop permutations and loop tiling are explicitly disabled to isolate the influences of the proposed technique. The experimental input is a set of kernel applications written in C; namely, a digital finite impulse response filter (FIR), Fast Fourier Transform (FFT), Susan image noise filter (SUSAN), adaptive digital filter (LMS), convolution, and DOT product. Their input data sets are given in Table 1. Our technique is performed on unrolled codes (factor 2) to exploit instruction level and memory access parallelism. There is one exceptional case. Since Dot_product consists of few lines of codes, thus unroll factor 8 is used.

Table 1. Program and inputs

Program	LMS	FIR	Dot product	Convolution	FFT	Susan
Input size	192KB	192KB	8KB	64KB	14KB	109KB

The actual experiments were conducted on a coarse grained reconfigurable array processor, called RSPA [10]. RSPA consists of 16 (4x4) processing elements (PEs) in which each PE is connected to 4 neighboring PEs and 4 diagonal ones, as illustrated in Section 2. The local memory architecture has 4 banks (with two sets), each connected to each row. The local memory is double buffered in hardware and the buffers can be switched in one cycle. The size of each buffer is 8Kbytes,

Table 2. Memory access cycles and runtime reduction compared to the baseline [11]

Program	LMS	FIR	Dot product	Convolution	FFT	Susan	Ave.
Memory access cycle reduction (%)	25	20	16	25	10	19.2	19.2
Execution time reduction (%)	14.3	14.3	9	8.3	6.6	6.4	9.8

and is connected to the system memory through a high-performance 16-bit pipelined bus. The system memory operates at half the frequency of the processor, thus the memory bandwidth is 16 bits per 2 cycles.

As compared to a conventional architecture, the array processor have no instruction or data cache, and the microarchitecture is configured specifically for the target application. The target architecture for this experiment assumes a single array processor with multiple external SDRAM memories, and an external main processor that can load the data and configuration onto the array processor, initiate its computation, and retrieve its results, as illustrated in Section 2.

In the following experiment, we compare the performance obtained by our array reorganization with two sets of SRAM(16KB, 4 banks x 2KB x 2 sets) against without ours. We report results obtained from simulation of the designs derived after perform-ing these data and code transformations. For the comparison, since there exist no other profile-driven compiler method to reorganize arrays for CGRAs, we use the most general existing data un-aware code mapping method from [9]. It maps the entire array into a single bank. Thus we can purely obtain how much improve the performance by applying the proposed technique. The performance improvements (decrease in cycle counts) and memory access time reduction due to our technique were measured in percent, using formula ((ORIG-OPT) / ORIG)*100.

-ORIG : original code, and

-OPT : optimized code with the proposed technique.

The first set of results in Table 2, shows the time reduction (in percentage), for each of the program kernels with applying our approach against without ours. With higher memory latencies, the benefits of memory parallelism increase, so we conservatively assign a low memory latency for both reads and writes of two cycle each, which is the case on our target platform when all memory accesses are fully pipelined. As compared to solutions that reorganize computation to optimize for memory parallelism assuming a fixed data layout (without applying our technique), our approach yields high memory parallelism for a fixed computation order by reorganizing the data. We observe greater than a 30% in the number of memory accesses reduction in a loop code and memory access cycles reduction ranging from 10% to 25% for 8 banks, as compared to the baseline [9] with unrolling twice.

The table also shows performance improvements achieved by applying the pro-posed algorithm. The overall performance improvement from our technique ranges from 6.4% to 14.28%, and the average performance improvement is 9.81%. Considering there is no modification made to existing instruction scheduler and the performance comparison is made to highly optimized code (-O3), performance gain from our technique was impressive.

Lower speedups were obtained for FFT and Susan because these kernels are highly compute bound and are not able to take advantage of the additional memory parallelism

exposed by our data placement.

Figure 6 shows improvement of parallelism by the proposed technique. The y-axis illustrates how much improved the parallelism from normalized value 1 which is generated by the baseline [11]. In general, more improvement in memory parallelism leads to more performance improvement.

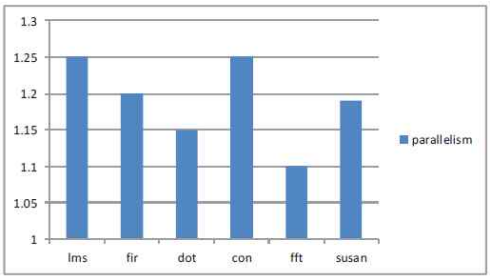


Fig. 6. Results of improved parallelism

In our studies, our approach is always better than the comparison. It is not surprising result since multimedia application have lots of memory parallelism. In addition, traditional approaches are normally hard to optimize such applications. As we go from 4 banks to higher number of banks, we see the growing importance of our optimization with larger unroll factors. Since larger unroll factors for the loops are needed for array reorganization to fully utilize the memory bandwidth of the platform. It is also important to note that increasing the number of banks and size sometimes gives a relatively small additional benefit on average. The reason is that complex data dependence limits inherent memory parallelism. Such a case is also seen for parallelizing compilers. A very large banks does not always lead to great performance improvement than a moderate size of SRAM.

VII. Related Work

Most previous work on CGRA [10, 11, 12] does not explicitly consider the local memory architecture or data placement. They assume that all the required data is already present in the local memory, and every load/store PE can access that data whenever they need to. One exception to this is [13], which assumes a hierarchical memory architecture, where the PEs are connected to a L0 local memory, which connects to the external main memory through an L1 local memory. Since both these local memories are scratchpads, and therefore statically scheduled, their main interest is in improving the reuse between the L0 and L1 local memories. An early work [14] on CGRA presents a methodology to evaluate memory architectures for CGRA mapping; however, it lacks a detailed mapping algorithm.

Optimizing locality if data accesses has been the main focus of several previous studies [15], see the references therein. Most of these techniques use linear loop trans-formations based on reuse vector space [16, 17] and cost based [13, 18] abstractions. The main limitation of these techniques is inherent data dependences in the code and imperfectly nested loop structures.

Many shared memory systems replicate data to enable concurrent read access [19, 20]. This optimization is clearly required to achieve any reasonable level of performance in systems that do not implicitly replicate data for concurrent read access, programmers explicitly replicate the data [21]. Similarly, renaming is designed to allow for concurrent operations that have output and anti-dependences but where there is no flow of values between statements of a loop nest. It has been used mainly for scalar variables as for arrays the additional memory costs make it very unprofitable for traditional high-end architectures. Array data-flow analysis [22, 23] focuses on data dependence analysis that is used to determine conditions for parallelization.

The work [1] is to show how linear memory access descriptor is used to analyze and simplify array access patterns in a program for more accurate compiler

optimization. Unfortunately, it is designed to a static analysis. In multimedia applications, such techniques are not sufficiently powerful to deal with all cases encountered in practice which is frequently. To overcome this limitation, the proposed approach is designed to a hybrid analysis (profile based technique). Thus, it provides the most accurate mem-ory access pattern summary. Based on such memory access pattern summary, we solve for an array reorganization problem to fully utilize memory parallelism. The proposed technique is based on the application specific data access patterns to optimize array partitioning and mapping.

Ⅷ. Conclusion

In this paper, we described an algorithm for deriving reorganized array data layouts in multiple memory banks for array-based computations, to facilitate high-bandwidth parallel memory accesses in modern architectures where multiple memory banks can simultaneously feed one or more functional units. By examining data dependences and array subscript expressions, our algorithm automatically derives application specific layouts in multiple memories.

A key consideration when applying this array reorganization algorithm is the feasibility of reorganizing data in memory. Here we considered loop nest computations, but when expanding to full applications, either the compiler must use the same layout throughout the program. Depending on the architecture and the application, such a re-organization could be more costly than the performance gain from increased memory parallelism.

A major focus of our current work is to formulate this array reorganization optimization as an interprocedural and global analysis problem, and compare the results with solutions that use efficient data reorganization. By using our technique, the experimental results show that the average improvement on performance is 9.8% compared with the existing method.

Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology (No. 2010-0024529 & 2011-0012522) and Sunchon National University Research Fund in 2011.

References

[1] Yunheung Paek, Jay Hoeflinger, and David Padua, "Simplification of array access patterns for compiler optimizations", In PLDI'98, pages60 - 71.

[2] Jean-Francois Collard and Daniel Lavery, "Optimizations to prevent cache penalties for the intel Itanium 2 processor", In Proceedings of the CGO'03, 105 - 114.

[3] P. Grun, N. Dutt, and A. Nicolau, "Access pattern based local memory customization for low power embedded systems", In Proceedings of the conference on DATE, 778 - 784.

[4] M. Gupta and P. Banerjee, "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers", IEEE Trans. Parallel Distrib. Syst., 3(2):179 - 193, 1992.

[5] Hartej Singh, Guangming Lu, Eliseu Filho, Rafael Maestre, Ming-Hau Lee, Fadi Kurdahi, and Nader Bagherzadeh, "Morphosys: case study of a reconfigurable computing system targeting multimedia applications", In Proceedings of DAC, 573 - 578, 2000.

[6] M. Wolfe, "More iteration space tiling", In Proceedings of the ACM/IEEE conferenceon, Supercomputing'89, 655 - 664.

[7] Nainesh Agarwal and Nikitas Dimopoulos,

"Dspstone benchmark of codel's automated clock gating platform", In Proceedings of the IEEE VLSI, 508 - 509, 2007.

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite", In Proceedings of the WWC-4. 2001.

[9] ICD-C compiler framework, University of Dortmund, .http://www.icd.de/es/icd-c/

[10] Yoonjin Kim, Mary Kiemb, Chulsoo Park, Jinyong Jung, and Kiyoung Choi, "Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization", In Proceedings of DATE'05, 12 - 17.

[11] A. Hatanaka and N. Bagherzadeh, "A modulo scheduling algorithm for a coarse-grain re-configurable array template", In Proceedings of the IPDPS'07, 1 - 8, 2007.

[12] Hyunchul Park, Kevin Fan, Manjunath Kudlur, and Scott Mahlke, "Modulo graph em-bedding: mapping applications onto coarse-grained reconfigurable architectures", In Proceedings of CASES'06, 136 - 146.

[13] Kathryn McKinley and Steve Carr, "Improving data locality with loop transformations", ACM Transactions on Programming Languages and Systems, 18: 424 - 453, 1996.

[14] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Adres: An architecture with tightly coupled vliw processor and coarse grained reconfigurable matrix", In Proceeding of Field Programmable Logic, FPL'03, 61 - 70.

[15] Michael Joseph Wolfe, "High Performance Compilers for Parallel Computing", Addison-Wesley Longman Publishing Co., USA, 1995.

[16] Wei Li, "Compiling for numa parallel machines", PhD thesis, Ithaca, NY, USA,1993.

[17] Michael E. Wolf and Monica S. Lam, "A data locality optimizing algorithm", In Proceedings of the ACM SIGPLAN 1991, 30 - 44.

[18] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen, "Combining loop transformations considering caches and scheduling", In MICRO29, 274 - 286, 1996.

[19] Daniel Edward Lenoski, "The design and analysis of DASH: a scalable directory-based multiprocessor", PhD thesis, Stanford, CA, USA, 1992.

[20] Kai Li, "Shared virtual memory on loosely coupled multiprocessors", PhD thesis, 1986.

[21] S. Lumetta, L. Murphy, X. Li, D. Culler, and I. Khalil, "Decentralized optimal power pricing: The development of a parallel program", In IEEE Parallel and Distributed Technology, 240 - 249, 1993.

[22] V. Balasundaram and K. Kennedy, "A technique for summarizing data access and its use in parallelism enhancing transformations", In Proceedings of the ACM SIGPLAN 1989, 41 - 53.

[23] Chau wen Tseng, "Compiler optimizations for eliminating barrier synchronization", ACM SIGPLAN, 144 - 155, 1995.

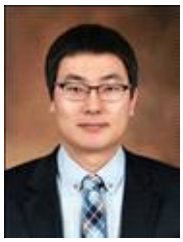
저 자 소 개

Doosan Cho

2001 : B.S. degree in EE from HUFS, Korea.
2003 : M.S. degree in EE from Korea Univ. Korea.
2009 : Ph.D degree in EE from Seoul National Univ. Korea.

2009~2010 : research director at Recores Inc.
2010~current : full time instructor of EE at Sunchon National Univ. Korea.
Research Interests : embedded software, optimizing compiler, low power design, system optimization, MPSoC.
Email: dscho@sunchon.ac.kr

Jonghee Youn



2003 : B.S. degree in EECS from Kyungpook National Univ. Korea.
2011 : Ph.D degree in EECS from Seoul National Univ. Korea.

2011~current : Lecturing professor of CSE at Gangneung Wonju National Univ. Korea.
Research Interests : embedded systems, optimizing compiler, software optimizations, MPSoC, GPGPU and computer architecture.
Email: jhyoun@gwnu.ac.kr