Operation and Data Mapping for CGRAs with Multi-bank Memory

Yongjoo Kim
School of EECS, Seoul National University, Korea
yjkim@optimizer.snu.ac.kr

Jongeun Lee *
School of ECE, Ulsan National Institute of Sci. and Tech. (UNIST), Ulsan, Korea
jlee@unist.ac.kr

Aviral Shrivastava
CML Research Group, Arizona State University, USA
Aviral.Shrivastava@asu.edu

Yunheung Paek
School of EECS, Seoul National University, Korea
ypaek@snu.ac.kr

Abstract

Coarse Grain Reconfigurable Architectures (CGRAs) promise high performance at high power efficiency. They fulfil this promise by keeping the hardware extremely simple, and moving the complexity to application mapping. One major challenge comes in the form of data mapping. For reasons of power-efficiency and complexity, CGRAs use multi-bank local memory, and a row of PEs share memory access. In order for each row of the PEs to access any memory bank, there is a hardware arbiter between the memory requests generated by the PEs and the banks of the local memory. However, a fundamental restriction remains that a bank cannot be accessed by two different PEs at the same time. We propose to meet this challenge by mapping application operations onto PEs and data into memory banks in a way that avoids such conflicts. Our experimental results on kernels from multimedia benchmarks demonstrate that our local memory-aware compilation approach can generate mappings that are up to 40% better in performance (17.3% on average) compared to a memory-unaware scheduler.

Categories and Subject Descriptors C.3 [*Special-Purpose and Application-Based Systems*]: Real-time and embedded systems; D.3.4 [*Processors*]: Code generation and Optimization

General Terms Algorithms, Design, Performance

Keywords Coarse-grained Reconfigurable Architecture, Compilation, Multi-bank Memory, Bank conflict, Arbiter.

1. Introduction

The need of high performance processing is undeniable, not only in increasing our pace of learning by large-scale simulation of fundamental particle and object interactions, but also to fructify the increasing horizons of possibilities in automation, robotics, ambient intelligence etc. General-purpose high performance processors attempt to achieve this, but pay a severe price in power-efficiency. However, with thermal effects directly limiting achievable performance, *power-efficiency* has become the prime objective in high performance solutions. Figure 1 shows that there is a fundamental trade-off between "performance" and "ease of programmability" and the power-efficiency of operation. It illustrates that special-purpose and embedded systems processors achieve high performance by trading off "performance" and "ease of programming" for higher power-efficiency. While high-performance processors operate at power-efficiencies of 0.1 MIPS/mW, embedded processors can operate at up to two orders of magnitude higher, at about 10 MIPS/mW. Application Specific Integrated Circuits provide extremely high performance, at extremely high power efficiency of about 1000 MIPS/mW, but they are not programmable. Among programmable platforms, CGRAs or Coarse Grain Reconfigurable Architectures come closest to ASICs in simultaneously achieving both high performance and high power-efficiency. CGRA designs have been demonstrated to achieve high performance at power efficiencies of 10-100 MIPS/mW [18].

Category	Processor Name	MIPS	W	MIPS/mW
VLIW	Itanium2	8000	130	0.061
GPP	Athlon 64 Fx	12000	125	0.096
GPMP	Intel core 2 quad	45090	130	0.347
Embedded	Xscale	1250	1.6	0.78
DSP	TI TMS320C6455	9.57	3.3	2.9
MP	Cell PPEs	204000	40	5.1
DSP(VLIW)	TI TMS320C614T	4.711	0.67	7

Figure 1. CGRAs promise the highest levels of power-efficiency in programmable architectures

The simultaneous high performance, and high power efficiency comes with significant challenges. The hardware of CGRAs is extremely simplified, with very little "dynamic effects", and the complexity has been shifted to the software. CGRAs are essentially an array of processing elements (PEs), like ALUs and multipliers, interconnected with a mesh-like network. PEs can operate on the result of their neighboring PEs connected through the interconnection

* Corresponding author.

LCTES'10, April 13–15, 2010, Stockholm, Sweden.

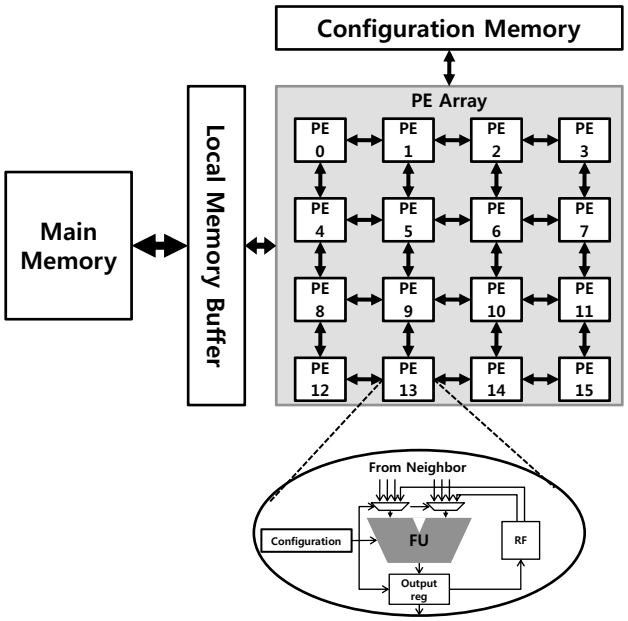


Figure 2. CGRA is just a 2-dimensional array of processing units, like adders and multipliers connection by a mesh-like interconnect. The computation has to be laid out in space and time, and the data routed through the interconnection explicitly in the application code.

network. CGRAs are completely statically scheduled, including the memory operations. One of the main challenges in using CGRAs is that the computation in the application must be laid out explicitly over the PEs, in space and time, and their data routed through the interconnection network. When we program general-purpose processor, the code just contains the "application" expressed in terms of the instruction set, and all this is automatically managed by the processor hardware. In contrast, this has to be explicitly done in the application code for CGRAs, and therefore compilation for CGRAs is quite tough.

A lot of work has been done towards this aspect of application mapping [5, 6, 12, 14–17, 20, 21], however, another aspect of application mapping, i.e., managing application data has been rather left untouched. Caches are an excellent dynamic structures, that ease "programmability" by automatically fetching the data required by the processor "on-demand" in general purpose processors. However, due to their dynamic behavior, high complexity and power consumption, CGRAs do not use caches, and use local memory instead. The local memory is raw memory, in the sense, that it does not store address tags for (or with) the data, and therefore form an separate address space than the main memory. The main challenge in using local memories is that, since there are no address tags, there is no concept of a "hit", or "miss". The application must explicitly bring the data that it will need next into the local memory and after its use, it must write it back and bring the data that will be needed after that.

To minimize the challenge, CGRAs could have large on-chip local memory so that all the required data may fit into the local memory which can be loaded once before the program, and then written back at the end of the program. Clearly this is not always possible, and in reality the on-chip local memories are rather small. Further complications arise, because PEs have to share the local memory especially in large, say 8x8 CGRA. If each PE should be able to read two data and write one data to the local memory, then we need 128 read ports, and 64 write ports. Even if one row of PEs access one port of the local memory. we need at 16 read and 8 write ports in the local memory. This is still quite large, and a more practical solution is to have multi-bank local memory, in which each bank has two read and one write port on the memory side, and a row of PEs sharing memory access on the PE array side. So that each PE can access data in any bank, a hardware arbiter between the memory requests generated by the PEs and the banks of the memory is used. We call such an architecture, that has arbiters in front of the memory ports of multiple banks, *Multiple Banks with Arbitration* (MBA) architecture, and most existing CGRA designs are MBA architectures [7, 13, 18].

Even in the MBA architecture, a fundamental restriction remains that a bank cannot be accessed by two different PEs at the same time remains. This is the challenge that we meet in this paper. Fundamentally there are two solutions to this. One is hardware solution, i.e., add a request queue in the arbiter, and increase the access latency of the memory operation, or second is to change the application mapping technique to explicitly consider the memory banking architecture, and map memory operations into rows, so that two different rows do not access the same bank simultaneously. We argue for the second technique and develop application data and operation mapping techniques to avoid memory bank conflicts. Our experiments on important multimedia kernels demonstrate that our memory memory aware compilation approach generates mappings that are up 17.3% better than the state-of-the-art memory unaware scheduler. As compared to the hardware approach using arbiters, our technique is on average 8.5% better, and promises to be a good alternative.

2. Background on CGRAs

2.1 CGRA Architecture

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. A PE is essentially a function unit (e.g., ALU, multiplier) and a small local register file. Some PEs can additionally perform memory operations (load/store), which are specifically referred to as load-store units. The functionality of each PE and the connections between PEs are controlled by *configuration*, much like the configuration bitstream in FPGAs. However, the configuration for CGRAs is coarser-grained (word level), and can be changed very fast, even in every cycle for some CGRAs [7, 13, 18].

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 [18] has 16 banks, every two of which may be accessed exclusively by each row of PEs (there are eight rows in total). However, this organization can severely limit the accessibility of the local memory, since a PE can access only its own share of the local memory. This limitation can be relaxed by providing arbiters or muxes at the interface (memory ports) to the local memory; for instance, a mux in front of a memory port allows the bank to be accessed by different load-store units at different cycles. We call such an architecture that has arbiters in front of the memory ports of multiple banks, *MBA (Multiple Banks with Arbitration)* architecture.

Even in MBA architecture, a fundamental restriction remains that a bank cannot be accessed by two different PEs at the same time, if the bank consists of single-port cells. (In the rest of the paper we assume that a bank consist of single-port cells, and thus

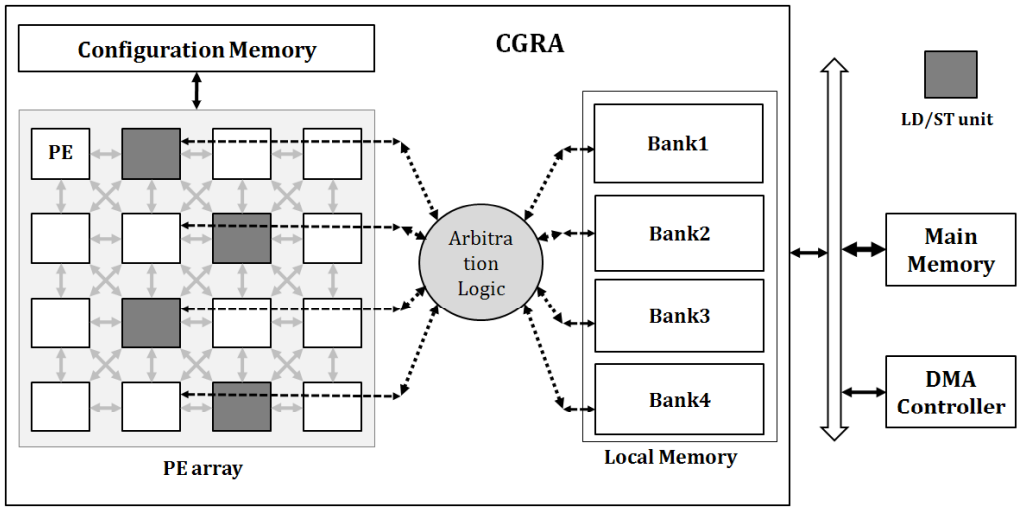


Figure 3. Multi-Bank with Arbitration (MBA) architecture: There is arbitration between the PE array and the memory banks, so that any PE can access data from any bank. However a fundamental limitation still remains: Two PEs cannot access data in the same bank simultaneously. This can be solved either by i) hardware approach of adding a queue to sequentialize the accesses, and ii) compiler approach, where compiler makes sure that this does not happen. This paper develops application operations and data mapping for the second approach and show that it is promising.

has only one port.) In MBA architecture, if two PEs try to access the same bank at the same time, a *bank conflict* occurs. CGRA hardware supporting MBA 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.

A solution proposed by [1] uses in front of each bank a hardware queue, called DMQ or DAMQ (Dynamically Allocated, Multi-Queue buffer) [19]. Though adding DMQ of length n $(n > 1)$ increases the latency of a load operation by $n - 1$ cycles, it allows up to n simultaneous memory requests to be serviced without a stall.[1] But since adding a queue cannot increase the bandwidth of a memory system, stall must be generated if the request rate exceeds the service rate, or the number of memory ports. We call such a memory architecture *MBAQ (Multiple Banks with Arbitration and Queues)* architecture, an example of which is the ADRES architecture. In this paper we present mapping algorithms for both MBA and MBAQ architectures, and compare them against using hardware solutions only.

2.2 Execution Model and Application Mapping

CGRA is typically used as a coprocessor, offloading the burden of the main processor by accelerating compute-intensive kernels. We assume blocking communication between main processor and CGRA coprocessor (i.e., no parallelism between them). For application mapping, first the loops that are mapped to CGRA are identified. The selected loops are then compiled for CGRA while the rest of the code is compiled for the main processor.

The result of CGRA compilation for selected loops is configuration, which is fed to the PE array at runtime. The other component, the local memory, gets the necessary data through DMA from system memory. After loop execution the output data of the loop may be transferred back to system memory. Such data transfers and CGRA computation are often interleaved to hide the data transfer latency. For CGRAs with larger local memories, opportunities may exist to reuse data (usually arrays) between different loops, as the output of one loop is often an input to the next loop. For instance, ADRES allows a fairly large local memory of up to 1 Mbytes in total, which can provide input data of 100 Kbytes each for 10 loops. In such a case if the data can be reused between the loops without needing to move the data around on the local memory (e.g., to another bank), it can greatly reduce the runtime as well as energy consumption of CGRA execution.

There are two dominant ways of placing arrays on multiple banks. *Sequential* refers to placing all the elements of an array to an particular bank, whereas *interleaving* refers to placing contiguous elements of an array on different banks. Interleaving can not only guarantee a balanced use of all the banks, but also more or less randomize memory accesses to each bank, thereby spreading bank conflicts around as well. The DMQ used in the MBAQ architecture thus can effectively reduce stalls due to bank conflicts when used with bank-interleaved arrays. However, interleaving makes it complicated for compilers or static analysis to predict bank conflicts. Thus our compiler approach uses sequential array mapping.[2]

3. Related Work

Memory architectures of CGRA can be largely classified into implicit load-store architecture (e.g., MorphoSys [18] and RSPA [7]) and explicit load-store architecture (e.g., ADRES [1, 13]). Whereas implicit load-store architectures have data (array elements) pre-arranged in the local memory and PEs can only sequentially access them, explicit load-store architectures allow random access of data in the local memory. There are also variations in the connection between banks and PEs. Whereas earlier architectures [7, 18] have assumed one-to-one connection between PE rows and local memory banks, recent architectures like ADRES assume one-to-many connection through muxes or arbiters, and even load queues. Our

[1] It works as if the DMQ holds the values of n requests until all of them become available, which requires $n - 1$ additional cycles in a pipelined memory, when all the load values are returned simulataneously.

[2] To be fair, we compare our approach with sequential array mapping against hardware approach (DMQ) with interleaved array mapping.

target architecture assumes explicit load-store with muxes and optionally queues.

Most previous CGRA mapping approaches [5, 6, 9, 10, 12, 14–17, 20, 21] consider computation mapping only, but not data mapping. [21] considers computation mapping as a graph embedding problem from a data flow graph into a PE interconnection graph, and solves the problem using a known graph algorithm. Many others consider mapping as a scheduling problem targeting an array processor, of which a simpler form is VLIW processor. Software pipelining and modulo scheduling is often used. [16] proposes an improved variant of modulo scheduling by considering edges rather than nodes (of input data flow graph) as the unit of scheduling. [14] proposes a scheduling algorithm for loops with recurrence relation (inter-iteration data dependence). In all those approaches, data mapping is only an after-thought, and not included in the optimization framework.

There is little work that considers memory during CGRA mapping. [3] considers a hierarchical memory architecture and presents a mapping algorithm to reduce the amount of data transfer between L1 and L2 local memory. [4] proposes routing reused data through PEs instead of using the local memory, which can reduce the local memory traffic and thus improve performance. Our work is orthogonal to this and the effective is additive when applied together. [4] also considers the data layout with sequential memory. But it is limited as it considers only one loop and is based on simulated annealing; extending it to multiple loops does not seem straightforward. [11] proposes the idea of quickly evaluating memory architectures for CGRA mapping; however, it lacks a detailed mapping algorithm. Our earlier work [8] proposes a mapping algorithm that takes into account data mapping as well as computation mapping. However, the memory architecture assumed in [8] is much simpler, with no arbiter or queues. In this paper we provide a more general framework that considers data and computation mappings together.

4. Problem of Memory Aware Mapping

Given a sequence of loops and the CGRA architecture parameters, the problem of CGRA compilation is to find the optimal mapping of the loops on to the CGRA architecture, which includes the PE array and the local memory. A CGRA mapping must specify two pieces of information; i) **computation mapping:** mapping from each operation of the loops to a specific PE (where) and to which schedule step (when, in cycle), and ii) **data mapping:** mapping from each array in the loops to which bank of the local memory to use. A loop is represented by the data flow graph of the loop body, along with data dependence information and memory reference information (which array is accessed by a memory operation and the access function). We assume that the number of iterations is given at runtime before the loop entry and remains constant during loop execution (hence the actual number may not be available at compile-time). The optimality of mapping is judged by the schedule length of the mapping, which is equivalent to the II (Initiation Interval) in the case of modulo scheduling. For a sequence of loops, we take the average II, assuming equal weights for all loops.

Hence the goal of optimization problem is to minimize the average II. In addition to the usual constraints for computation mapping (e.g., [16]), we have additional constraints for optimal data mapping. One thing to understand to derive the constraints is that assuming sequential array placement and a sufficiently large local memory, the optimal solution should be without any expected stall. If there is an expected stall in the optimal mapping, one can always find a different mapping that has the same schedule length but no expected stall. Simply add a new cycle in the place where a stall is expected and schedule one of the conflicting memory operations at the new cycle, which does not increase the actual schedule length and has no expected stall. Thus we can limit our search to those with no expected stall, without losing optimality. This no-conflict condition translates into different forms depending on the memory architecture. For a MBA architecture (any load-store PE can access any bank through arbitration), the constraint is that there must be at most one access to each bank at every cycle. For a MBAQ architecture, the memory access latency is slightly increased. If the added latency is n cycle, there must be at most n accesses to each bank in every n consecutive cycles.

5. Our Approach

5.1 Overview

The main challenge of our problem comes from inter-dependence between computation mapping and data mapping; i.e., fixing data mapping creates constraints on computation mapping, and vice versa. Due to the inter-dependence, the optimal solution can only be obtained by solving the two subproblems simultaneously. However solving the two subproblems simultaneously is extremely hard, since even a subproblem alone, i.e., computation mapping, is an intractable problem. In [21], spatial mapping is solved by ILP formulation. In this experiments, ILP solver cannot find a solution within a day, if a node number excesses only 13. Our problem is also intractable, since it can be reduced to the computation temporal mapping problem, which is more intractable than spatial mapping. Hence our heuristic solves them sequentially, first clustering data arrays to balance utilization and access frequency of each bank, and then finding computation mapping through conflict free modulo scheduling.

First we perform a pre-mapping, which is just computation mapping by traditional modulo scheduling without considering data mapping. The II resulting from pre-mapping serves as the minimum II in the ensuing iterative process. We then repeat the two steps of array clustering and conflict free scheduling, incrementing II, until a conflict free mapping is found. Pre-mapping can provide a tighter upper bound for II than traditional minimum II calculation considering resource and recurrence requirements only, and thus reduce the overall time for CGRA compilation.

5.2 Array Clustering

Like other CGRA architectures including ADRES, our target architecture is homogeneous across multiple rows though it may have heterogenous PEs within a row. This makes data mappings position-independent, meaning that the absolute position, or bank, that an array is mapped to is not important as long as the same set of arrays is mapped together.

Array mapping can affect performance (through computation mapping) in at least two ways. First, banks have a limited capacity. If arrays are concentrated in a few banks, not only does it reduce the effective total size of the local memory, but there is a higher chance of having to reload arrays during a loop execution or between loops, diminishing data reuse. And since the size of arrays can be very different from each other especially due to different strides, it is important to balance the utilization of banks and prevent pathological cases. Second, each array is accessed a certain number of times per iteration in a given loop, which is denoted by Acc_A^L for array A and loop L. In both MBA and MBAQ architectures, II cannot be less than the sum of access counts of all the arrays mapped to a bank. In other words, there can be no conflict free scheduling if $\sum_{A \in \mathcal{C}} Acc_A^L > II'_L$, where $\mathcal{C}$ is an array cluster (set of arrays) that will be mapped to a certain bank and II'_L is the current target II of loop L. Thus it is important to spread out accesses (per-iteration access count) across banks. Note that bank utilization balancing is static, or independent of loops, whereas balancing per-iteration access counts is dependent on which loop we are looking at.

We combine the two factors, viz. array size and array access count, into one metric, i.e., priority. Our clustering algorithm takes one array at a time and determines its clustering by assigning a cluster to it. Because of the greedy nature of the algorithm the order of selecting arrays is important. We use priority to determine which array to cluster first. The priority of an array A is defined as:

$$priority_A = Size_A/SzBank + \sum_{\forall L} Acc_A^L/II'_L \quad (1)$$

where $Size_A$ is the size of array A and $SzBank$ is the size of a bank.

Once the priorities of all arrays are calculated, we begin assigning cluster to arrays, starting from the one with the highest priority. To make this decision of which cluster to assign to a given array, we compare the relative costs of assigning different clusters. Similarly to the priority definition, our cost model considers both array size and array access count, and is defined as follows. Given a cluster $\mathcal{C}$ and an array A, the relative cost of assigning $\mathcal{C}$ to A is,

$$cost(\mathcal{C}, A) = Size_A/SzSlack_{\mathcal{C}} + \sum_{\forall L} Acc_A^L/AccSlack_{\mathcal{C}}^L \quad (2)$$

where $SzSlack_{\mathcal{C}}$ and $AccSlack_{\mathcal{C}}^L$ are the remaining space of a cluster (total budget is $SzBank$) and the remaining per-iteration access count of loop L (total budget is II'_L), respectively, and are updated as assignments are made. In this formula, we use the remaining value to calculate cost. If one bank's size or access count is used up too much than other bank, we have to avoid assigning array to this bank for balancing and leaving it to other array which really need this resource. So, to avoid this case, when we calculate cost, we decide to divide by remaining resource. If remaining resources are small, the cost becomes bigger. In a consequence, the balancing is derived. Figure 4 shows the pseudo code of array clustering algorithm. From line 1 to 6, array information is analyzed. From line 7 to 10, priority for each array is calculated. And in the remaining parts, assignment is executed. The array which has high priority is assigned first.(11) Cost for each cluster is calculated(14-17) and then the cluster which shows minimum cost is chosen(19). If the assignment is failed by a lack of access count, increase II and start clustering again. But if the reason of fail is lack of memory size, then we need to reduce the number of loop considered together. After array clustering, we can decide a MII for conflict aware scheduling. In previous modulo scheduling algorithm, resMII and recMII is used to calculate MII. But in our work, we used third factor that is calculated in array clustering step. We define the third factor is MemMII(Memory-constrained minimum II). MemMII is related with the number of access to each bank for one iteration and a memory access throughput per a cycle. The array clustering is calculated with this information. So, the II, the result of array clustering, is MemMII.

Figure 5 illustrates array clustering. Figure 5(a) shows the parts of array analysis result(access frequency analysis). Then after calculating array priority once (Figure 5(b)), the minimum-cost clusters are assigned to arrays, in the decreasing order of array priority (Figure 5(c)). Figure 5(d) lists the number of accesses to each bank (per iteration), which is balanced across different banks and loops.

5.3 Conflict Free Scheduling

With previous memory-unaware scheduling, bank conflicts can occur even if array clustering is first done. It is because array clustering only guarantees that the *total* per-iteration access count to the arrays included in a cluster, or simply, the total (per-iteration) access count of a bank, does not exceed the target II(because it is already reflected by MemMII), which is only a necessary condition for a conflict free mapping. In other words, once array clustering is done, the total access count of a bank does not change because

```
ArrayInfo;        // a data structure that contains a size of array,
                     # access for each loop's iteration
AL;               // a list of arrayInfo
DFGs;             // a list of DFG
CandList;         // a set of candidate cluster

    // array analysis
1.      AL = initArrayList();
2.      for(each DFGs) {
3.        * let dfg be an element of DFGs;
4.        AL = arrayAccessAnalysis(dfg);
5.      }
6.      AL = arraySizeAnalysis();
    // cluster assignment
7.      for(each arrayInfo in AL) {
8.        * let x be an element of AL
9.        calcPriority(x);
10.     }
11.     sortArrayList(AL); // sort in descending order by priority
12.     while(AL ≠ Ø) {
13.       * let x be an element of AL
14.       for(#cluster) {
15.         * let y be a cluster number
16.         candList += calcCost(x, y);
17.       }
18.       if (candList ≠ Ø) {
19.         cl = getMinCostCluster(candList);
20.         assign(x, cl);
21.       }
22.       else {
23.         fail(); // increase II and do clustering again
24.       }
25.     }
```

Figure 4. Array clustering algorithm

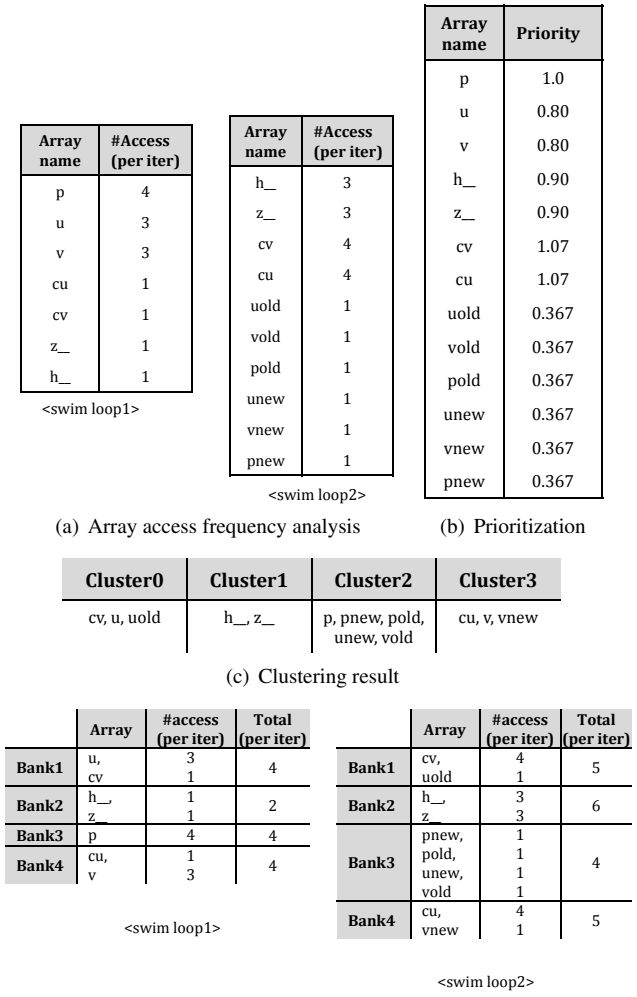
of scheduling, but temporary access count can. For instance if two memory operations accessing the same bank are scheduled at the same cycle, spontaneously two load-store units will try to access the same bank, which is a bank conflict. Thus we extend a previous modulo scheduling algorithm [16] developed for CGRAs to generate conflict free mapping.

5.3.1 Base scheduling algorithm

In this paper, we applied a previous scheduling research results to make a base modulo scheduling algorithm. We decide to use EMS(Edge-centric Modulo Scheduling) [16] for our base scheduler. In previous node-centric scheduling, the placements of source node and destination node are decided first. And then it try to find the routing path between these nodes. But in the edge-centric approach, routing from source node is tried first. During routing, if routing path pass through a place that destination node can be placed, then the placement is decided at this time. Figure 6 shows a pseudo code of our base scheduler. In this paper, we use several costs for placement decision, which are widely used in mapping algorithm such as resource cost, routing cost, relativity cost. Resource cost means the cost for using a PE for node placement. Its costs are set to be different according to function of PE. If target PE is expensive PE, such as the PE having memory access unit, the cost is set to higher. Routing cost means the cost for routing data to destination. The cost becomes bigger, if routing path is long or routing path passed by expensive PE. Relativity cost is used for placing related nodes at adjacent PEs. In these basic modulo scheduling environment, we added our approaches.

5.3.2 MBA Architecture

Modulo scheduling for CGRA uses placement and routing (P&R) technique to find feasible scheduling and resource allocation simultaneously. While the resources that are considered in previous mod-



Array name	#Access (per iter)
p	4
u	3
v	3
cu	1
cv	1
z__	1
h__	1

<swim loop1>

Array name	#Access (per iter)
h__	3
z__	3
cv	4
cu	4
uold	1
vold	1
pold	1
unew	1
vnew	1
pnew	1

<swim loop2>

(a) Array access frequency analysis

Array name	Priority
p	1.0
u	0.80
v	0.80
h__	0.90
z__	0.90
cv	1.07
cu	1.07
uold	0.367
vold	0.367
pold	0.367
unew	0.367
vnew	0.367
pnew	0.367

(b) Prioritization

Cluster0	Cluster1	Cluster2	Cluster3
cv, u, uold	h__, z__	p, pnew, pold, unew, vold	cu, v, vnew

(c) Clustering result

	Array	#access (per iter)	Total (per iter)
Bank1	u, cv	3 1	4
Bank2	h__, z__	1 1	2
Bank3	p	4	4
Bank4	cu, v	1 3	4

<swim loop1>

	Array	#access (per iter)	Total (per iter)
Bank1	cv, uold	4 1	5
Bank2	h__, z__	3 3	6
Bank3	pnew, pold, unew, vold	1 1 1 1	4
Bank4	cu, vnew	4 1	5

<swim loop2>

(d) Number of accesses for each loop

Figure 5. Array clustering example.(swim application)

ulo scheduling include only PEs and interconnects, our extension treats memory banks, or memory ports to the banks, as resources too.[3] This small extension, combined with our array clustering, allows us to find conflict free mapping.

As explained, if two memory accesses to the same bank occur at the same cycle, memory conflict occurs. Then one memory access request cannot be completed on time. But we know the array clustering information, we can prevent memory conflict by saving the time information that memory operation is mapped on. When a memory operation is mapped, update the cluster access information. And by using this information, we can prevent that two memory operations belonging same cluster is mapped on the same cycle.

Figure 7 illustrates our conflict free mapping algorithm through an example shown in Figure 7(b). Suppose that the architecture has four PEs, two of which are load-store units (PE0 and PE2), and it has two banks with arbiters (MBA architecture). Also assume that arrays have been clustered as listed in Figure 7(c) with the target II of 3, and that all the nodes from 1 through 7 have been scheduled as shown in Figure 7(d), and now node 8, a memory operation, is

[3] We use bank and cluster interchangeably since clusters are one-to-one mapped to banks.

```
DFG G;              // the data flow graph
NodeList N;         // the set of DFG node
CandSpace S;        // the set of mapping candidate
1.      prioritizeEdge(G);
2.      N = getOrderedNodes(G);
3.      while(N ≠ Ø) {
4.        targetN = pop(N);
5.        setSearchSpace(targetN);
                    // consider targetN's predecessor
6.        for(searchSpace) {
7.          * let s be a mapping space of searchSpace
                    // mapping space consists of a location of PE & time)
8.          if( spaceAvailable(s) && routable(s) ) {
              // spaceAvailable checks the functionality and availability of PE
              // routable function checks the routability from source nodes to s
9.            S += saveCandidate( s, calcCost(s) );
                  // resource, routing and relativity cost are used to calculate cost
10.         }
11.       }
12.       if( S≠ Ø ) {
13.         decision = getMinCostCand( S );
14.         update( decision );
15.       else {
16.         fail();          // increase II & do scheduling again
17        }
18      }
```

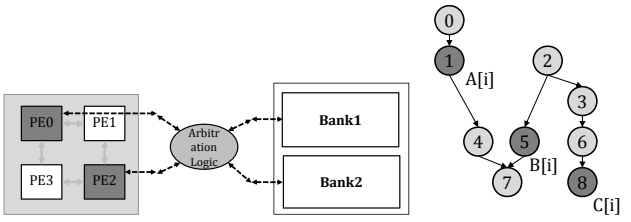
Figure 6. Base modulo scheduling(implemented based on EMS algorithm)

about to be scheduled. The first candidate for node 8 is cycle 4 on PE2, which has conflict not in terms of computation resources (PE2 was not used in cycle 1 nor 4) but in terms of memory resources (CL1 was already used in cycle 1). Choosing the first candidate thus means bank conflict, or one stall, per every II, effectively increasing II by one. Alternatively node 8 can be scheduled at cycle 6 on PE1 albeit via a longer route. But since this choice does not cause any conflict on computation or memory resources, the effective II is not increased, resulting in much better performance than the first candidate. Thus our extended modulo scheduling can find conflict free mapping, which works well with our array clustering. Moreover our approach can be easily applied to previous work. In EMS algorithm, our clustering aware scheduling approach is implemented in spaceAvailable function which is called Figure 6 line 9. spaceAvailable function checks several conditions to confirm the space is available. Conflict free approach just adds one more condition on these, so there is no hard problems with unifying previous scheduling algorithm with our approach.

5.3.3 MBAQ Architecture

As mentioned before, accesses to the same bank at the same time occur bank conflict. MBA architecture treats this problem by stalling CGRA processor until all requested data are ready. But stalling processor degrade CGRA performance, so some works[1, 2] proposed DMAQ architecture. They designed an arbitration logic have queues for each bank and increase memory operation latency longer than the lowest obtainable memory latency. During this additional cycles, the loaded data is stored in the arbitration logic queue. And at the end of latency, the data is delivered to PE. In this manner, several accesses to the same array can be handled without processor stall by fetching data earlier to give an extra time for fetching other conflicted data. But these previous works assumed interleaving memory architecture. In our case, we can predict bank conflict. So in our case, MBAQ architecture is used for relaxing the mapping constraint. MBA architecture doesn't permit bank conflict, but MBAQ architecture can permit several conflict within a range of added memory operation latency. We distinguish two cases(n is the added memory operation latency cycles by MBAQ approach)

i) $II' \leq n$ (Target II is no greater than the DMQ length): Our array clustering guarantees that there are at most II' accesses per



(a) (b)

Cluster1	Cluster2
A[i], C[i]	B[i]

(c)

	PE0	PE1	PE2	PE3	CL1	CL2
0		(0)	x	x		
1	(1)	(2)			A	
2	(3)	(4)	(5)			B
3		x	(7)	(6)		
4	x	x	(c1)	⊙	x	
5	x	x	x	⊙		x
6	(c2)	x	x	x		

II=3

(d) Scheduling table, with resources horizontally (CL1 means Cluster1) and time vertically.

Figure 7. Conflict free scheduling.

iteration to every bank; if such a clustering cannot be found, the target II is incremented. The worst scheduling of the II' accesses, from the bank conflict point of view, is if they are all scheduled at the same cycle(recollect MemMII)—and none until II' cycles later. But this case cannot generate a bank conflict because the DMQ can absorb at most n simultaneous requests. Therefore if $II' \leq n$, any schedule is conflict free.

ii) $II' > n$ (Target II is greater than the DMQ length): In this case, processor stall can occur if we do not consider data layout. To ensure absence of processor stall, the scheduler checks if the spontaneous request rate exceeds 1 whenever a new memory operation is placed, where the spontaneous request rate can be calculated as the number of memory operations during the last n cycles divided by n. If the request rate doesn't exceed 1, bank conflict can be absorbed by DMAQ memory interface.

6. Experiments

6.1 Setup

For the target architecture we use a CGRA that is very similar to the one illustrated in Figure 3. It has four load-store units in the locations shown in the figure. The local memory has 4 banks, each of which has one read/write port. Arbitration logic allows every load-store unit to access any bank. Similarly to ADRES, we assume that the local memory access latency is 3 cycles without DMQ; with DMQ whose length is 4, the local memory load latency is 7 cycles. We assume that the local memory size is unlimited for our experiments. Our CGRA has no shared register file, but each PE has its own register file whose size is 4 entries. The local registers are used for scalar variables or routing temporary data. A PE is connected to its four neighbor PEs and four diagonal ones.

We use important kernels from multimedia applications. To get performance numbers we run simple simulation on the mapping result as well as array placement, which gives the total number of execution cycles consisting of stall cycles and useful (non-stall) cycles. Because of randomness in the scheduling algorithm (in the placement and routing) we compile each loop ten times and the average performance is taken as the representative performance of the algorithm for that loop.

6.2 Effectiveness of Our Compiler Approach

To see the effectiveness of our approach, we compare our memory aware scheduling (MAS) with the hardware approach using DMQ to reduce bank conflict. For the hardware approach we use an existing modulo scheduling algorithm [16], which is referred to as memory unaware scheduling (MUS). MUS is also used as the base scheduler for our memory aware scheduling.

Figure 8 compares three architecture-compiler combinations. The first one, which represents the baseline performance, is the combination of MUS with hardware arbiter only. Having no DMQ, a stall occurs whenever there are more than one requests to the same bank. Bank conflict is detected and resolved at runtime. Interleaving array placement is used. The load latency is small (3 cycles) as DMQ is not used. The second case is the hardware approach, using DMQ to absorb some potential bank conflicts. Again, interleaved array placement is used to maximize the effectiveness of DMQ (by distributing bank conflicts). The use of DMQ results in longer load latency (7 cycles). The third case is our compiler approach, using MAS to generate conflict free mapping. Only hardware arbiter is required, but no DMQ or runtime conflict detection/resolution hardware. The load latency is small (3 cycles). Sequential array placement is used. The graph plots the runtimes normalized to that of the baseline. We assume that the clock speed is the same for all the cases.

In the baseline case we observe that about 10 $\sim$ 30% of the total execution time is spent in stalls. Using the DMQ, the hardware scheme can effectively reduce the stall cycles, which now account for a very small fraction of the total execution time. The non-stall times are mostly the same as in the baseline case, with a few exceptions. The notable increase of the non-stall time of hardware scheme in some applications (CopyImg, Init_mbaff) is due to the increase in the load latency. Overall the hardware scheme can reduce the expected CGRA runtime, though not always, by 10 $\sim$ 27% (9.7% on average) compared to the baseline case. With our compiler approach the stall time is completely removed. The increase in the non-stall time is very small to modest in most cases, reducing the total execution time by up to more than 40%. The graph shows that our compilation technique can achieve in most cases far better performance (10 to 40% runtime reduction, 17.3% on average) compared to memory unaware mapping. Also our approach allows the removal of bank conflict resolution hardware, which can contribute to reducing the cost and energy consumption. Further, even compared to the hardware approach using DMQ, our approach can deliver higher performance in most loops (on average 8.5% runtime reduction). Considering that the use of DMQ can reduce the speed of the processor as well as complicate its design and verification, the advantage of our compiler approach is many-fold.

6.3 Effect of Using DMQ in Our Approach

Since DMQ can reduce bank conflicts and increase performance when used with a conventional memory unaware scheduler, it is interesting to know how effective it is with our memory aware compilation flow. Figure 9 compares the II (the average of ten trials) by our memory aware compiler with/without DMQ. Our compiler can generate different mappings for architectures with DMQ, by relaxing the bank conflict condition. Surprisingly, contrary to the significant performance improvement in the case of memory unaware scheduling, DMQ does not really help in the case of our memory

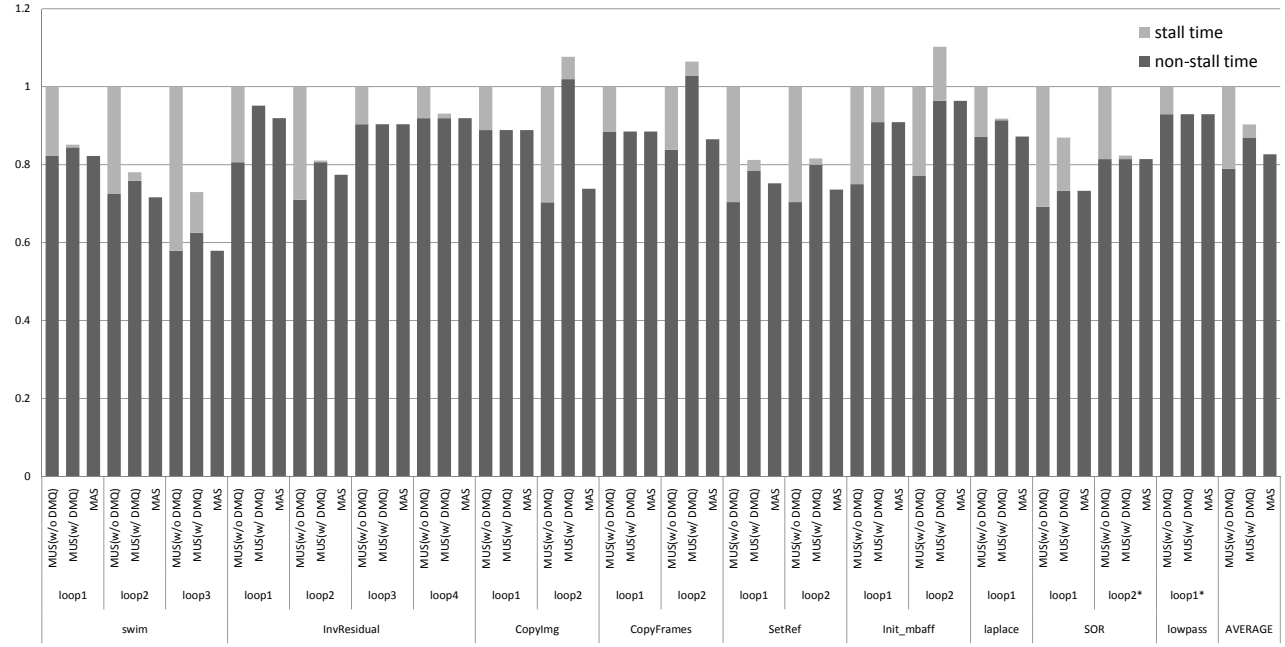
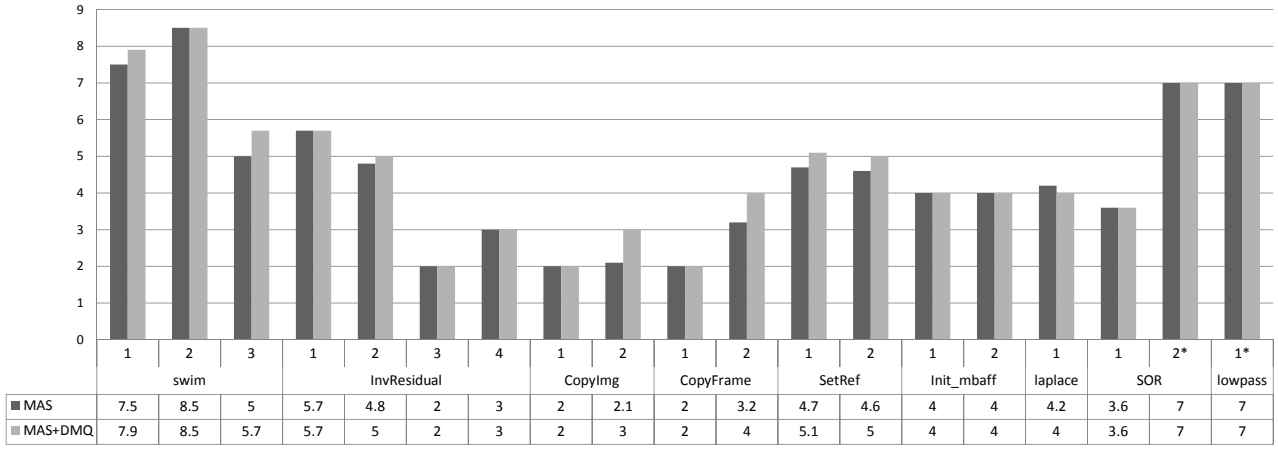


Figure 8. Runtime comparison, normalized to the baseline case (MUS w/o DMQ). Total execution time is the sum of non-stall time and stall time. Our compiler approach (MAS) completely eliminates stall time in all cases and can achieve up to 40% better performance than the baseline. Asterisk (*) indicates the loop has recurrent edge, where non-stall time can be increased due to data dependence



	1	2	3	1	2	3	4	1	2	1	2	1	2	1	2	1	1	2*	1*
	swim			InvResidual				CopyImg		CopyFrame		SetRef		Init_mbaff		laplace	SOR		lowpass
MAS	7.5	8.5	5	5.7	4.8	2	3	2	2.1	2	3.2	4.7	4.6	4	4	4.2	3.6	7	7
MAS+DMQ	7.9	8.5	5.7	5.7	5	2	3	2	3	2	4	5.1	5	4	4	4	3.6	7	7

Figure 9. Comparing memory aware mappings with and without DMQ. Y-axis represents II (average of ten trials).

aware mapping. Mostly the II is the same, and in some cases the II actually increases if there is DMQ. This is because while DMQ relaxes the bank conflict condition, it also increases the load latency, complicating the job of the scheduler. Thus we conclude that one of the best architecture-compiler combinations is our memory aware mapping plus MBA (multiple bank with arbitration) architecture, which again does not require runtime conflict detection and contributes to reducing the complexity of the memory interface design.

7. Conclusion

We presented a data mapping aspect of CGRA compilation, focusing on how to efficiently and effectively reduce bank conflicts for realistic local memory architectures. Bank conflict is a fundamental problem and can cause a serious degradation of performance. To reduce or eliminate bank conflicts either hardware approach or compiler approach can be used. We define the mapping problem for compiler approach, and also propose a heuristic approach, since the problem is computationally intractable like many problems in compilation. Our approach is a two-step process, first determining the array mapping considering statically known information, and then finding a computation mapping that is free of any bank conflict. Our experiments demonstrate that our memory memory aware compilation approach can generate mappings that are up to 40% better in performance (on average 17.3%) compared to memory unaware scheduler. Even compared to the hardware approach using DMQ, our technique is on average 8.5% better, and can be a good alternative to the hardware solution. Moreover, our compiler guarantees

that all the mappings are free of bank conflict, which can be used to eliminate conflict resolution hardware, which is another advantage of our approach.

Acknowledgments

This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST)/ Korea Science and Engineering Foundation(KOSEF)(R11-2008-007-01001-0), Seoul R&BD Program(10560), the IDEC and the Korea Science and Engineering Foundation(KOSEF) NRL Program grant funded by the Korea government(MEST) (No. 2009-0083190), the Korea Research Foundation Grant funded by the Korean Government(MOEHRD) (KRF-2007-357-D00225), 2009 Research Fund of UNIST, and grants from National Science Foundation CCF-0916652, Microsoft Research, Raytheon, SFAz and Stardust Foundation.

References

[1] B. Bougard, B. De Sutter, D. Verkest, L. Van der Perre, and R. Lauwereins. A coarse-grained array accelerator for software-defined radio baseband processing. *IEEE Micro*, 28(4):41–50, 2008.

[2] F. Bouwens. Power and performance optimization for adres. Master's thesis, Delft University of Technology, 2006.

[3] G. Dimitroulakos, M. D. Galanis, and C. E. Goutis. Alleviating the data memory bandwidth bottleneck in coarse-grained reconfigurable arrays. In *ASAP '05: Proceedings of the 2005 IEEE International Conference on Application-Specific Systems, Architecture Processors*, pages 161–168, Washington, DC, USA, 2005. IEEE Computer Society.

[4] G. Dimitroulakos, S. Georgiopoulos, M. D. Galanis, and C. E. Goutis. Resource aware mapping on coarse grained reconfigurable arrays. *Microprocess. Microsyst.*, 33(2):91–105, 2009.

[5] M. Ahn, J. Yoon, Y. Paek, Y. Kim, M. Kiemb, and K. Choi. A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 363–368, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[6] A. Hatanaka and N. Bagherzadeh. A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, March 2007.

[7] Y. Kim, M. Kiemb, C. Park, J. Jung, and K. Choi. Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 12–17, Washington, DC, USA, 2005. IEEE Computer Society.

[8] Y. Kim, J. Lee, A. Shrivastava, J. Yoon, and Y. Paek. Memory-aware application mapping on coarse-grained reconfigurable arrays. In *HiPEAC 2010*, LNCS 5952, pages 171–185, 2010. Springer-Verlag.

[9] J. Lee, K. Choi, and N. D. Dutt. An algorithm for mapping loops onto coarse-grained reconfigurable architectures. *ACM SIGPLAN Notices*, 38(7):183–188, 2003.

[10] J. Lee, K. Choi, and N. Dutt. Compilation approach for coarse-grained reconfigurable architectures. *IEEE D&T*, 20:26–33, Jan./Feb. 2003.

[11] J. Lee, K. Choi, and N. Dutt. Evaluating memory architectures for media applications on coarse-grained reconfigurable architectures. In *ASAP '03: Proceedings of the conference on application-specific systems, architectures, and processors*, pages 172–182, 2003. IEEE Computer Society.

[12] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Dresc: a retargetable compiler for coarse-grained reconfigurable architectures. pages 166–173, Dec. 2002.

[13] 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. pages 61–70. 2003.

[14] T. Oh, B. Egger, H. Park, and S. Mahlke. Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. *SIGPLAN Not.*, 44(7):21–30, 2009.

[15] H. Park, K. Fan, M. Kudlur, and S. Mahlke. Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 136–146, New York, NY, USA, 2006. ACM.

[16] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 166–176, New York, NY, USA, 2008. ACM.

[17] C. O. Shields, Jr. *Area efficient layouts of binary trees in grids*. PhD thesis, 2001. Supervisor-Ivan Hal Sudborough.

[18] H. Singh, G. Lu, E. Filho, R. Maestre, M.-H. Lee, F. Kurdahi, and N. Bagherzadeh. Morphosys: case study of a reconfigurable computing system targeting multimedia applications. In *DAC '00: Proceedings of the 37th Annual Design Automation Conference*, pages 573–578, New York, NY, USA, 2000. ACM.

[19] Y. Tamir and G. L. Frazier. Dynamically-allocated multi-queue buffers for vlsi communication switches. *IEEE Trans. Comput.*, 41(6):725–737, 1992.

[20] G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh, and W. Bohm. A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture. In *CASES '01: Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 116–125, New York, NY, USA, 2001. ACM Press.

[21] J. W. Yoon, A. Shrivastava, S. Park, M. Ahn, R. Jeyapaul, and Y. Paek. Spkm: a novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. In *ASP-DAC '08*, pages 776–782, 2008.