Memory Access Optimization in Compilation for Coarse-Grained Reconfigurable Architectures

YONGJOO KIM, Seoul National University
JONGEUN LEE, Ulsan National Institute of Science and Technology
AVIRAL SHRIVASTAVA, Arizona State University
YUNHEUNG PAEK, Seoul National University

Coarse-grained 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 multibank 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 in 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. To further improve performance on multibank memories, we propose a compiler optimization for CGRA mapping to reduce the number of memory operations by exploiting data reuse. Our experimental results on kernels from multimedia benchmarks demonstrate that our local memory-aware compilation approach can generate mappings that are up to 53% better in performance (26% on average) compared to a memory-unaware scheduler.

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

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Coarse-grained reconfigurable architecture, Compilation, multibank memory, bank conflict, array mapping

ACM Reference Format:
Kim, Y., Lee, J., Shrivastava, A., and Paek, Y. 2011. Memory access optimization in compilation for coarse-grained-reconfigurable architectures. ACM Trans. Des. Autom. Electron. Syst. 16, 4, Article 42 (October 2011), 27 pages.
DOI = 10.1145/2003695.2003702 http://doi.acm.org/10.1145/2003695.2003702

This work was supported in part by the Engineering Research Center program (grant 2011-0000975) and the NRL Program (grant 2010-0018465) funded by the Ministry of Education, Science and Technology (MEST) of the Korea government/the Korea Science and Engineering Foundation (KOSEF) and the IDEC, in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by MEST, under grant 2010-0011534, and in part by funding from the National Science Foundation grants CCF-0916652, CCF-1055094 (CAREER), NSF I/UCRC for Embedded Systems (IIP-0856090), Raytheon, Intel, Microsoft Research, SFAz, and Stardust Foundation.
Authors' addresses: Y. Kim and Y. Paek, School of EECS, Seoul National University, Seoul, Korea; J. Lee (corresponding author), School of ECE, Ulsan National Institute of Science and Technology, Ulsan, Korea; email: jlee@unist.ac.kr. A. Shrivastava, Department of CSE, Arizona State University.

DOI 10.1145/2003695.2003702 http://doi.acm.org/10.1145/2003695.2003702

1. INTRODUCTION

The need for 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 and soon. 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. Table I shows that there is a fundamental tradeoff between "performance" and "ease of programmability" and the power-efficiency of operation. It illustrates that special-purpose and embedded system 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 an extremely high power efficiency of about 1000 MIPS/mW, but they are not programmable. Among programmable platforms, CGRAs or coarse grained 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 [Singh et al. 2000].

The promise of 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 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 is routed through the interconnection network. When we program a general-purpose processor, the code contains just 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 done explicitly in the application code for CGRAs, and therefore compilation for CGRAs is quite tough.

A lot of work has been done on this aspect of application mapping [Mei et al. 2002; Park et al. 2006; Hatanaka and Bagherzadeh 2007; Ahn et al. 2006; Yoon et al. 2008; Shields 2001; Park et al. 2008; Venkataramani et al. 2001; Oh et al. 2009], however, another aspect of application-mapping, that is, managing application data has been left untouched. Caches are an excellent dynamic structure, that eases programming 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, but use local memory instead. The local memory is raw memory, in the sense that it does not store address tags for the data, and therefore forms a separate address space from the main memory. The main challenge in using local memories is that, since there are no address tags, there is no concept of "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 program execution, 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.

Table I. CGRAs Promise the Highest Levels of Power-Efficiency in Programmable Architectures

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	3.33	2.9
MP	Cell PPEs	204000	40	5.1
DSP (VLIW)	TI TMS320C614T	4.711	0.67	7

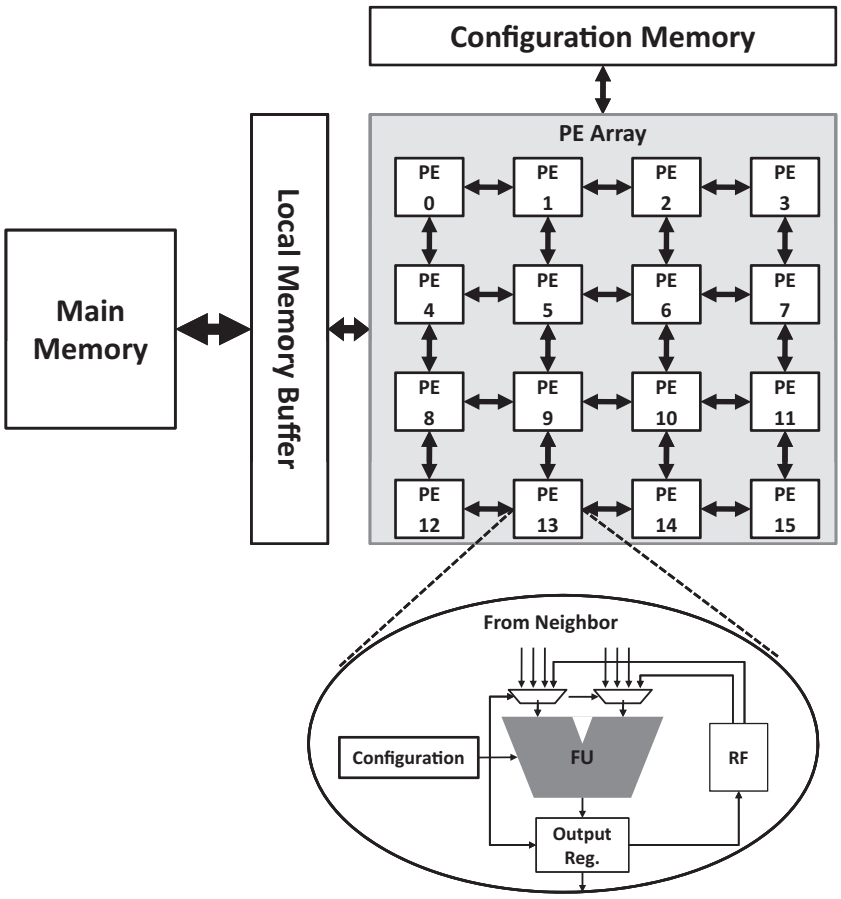


Fig. 1. CGRA is a two-dimensional array of processing units, like an adders and multipliers connection by a mesh-like interconnect. The computation has to be laid out in space and time, and the data explicitly routed through the interconnection in the application code.

Further complications arise because PEs have to share the local memory, especially in a large, say 8×8, 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 the local memory can be accessed by only one PE per each row, we need 16 read and 8 write ports in the local memory. This is still quite large, and a more practical solution is to have multibank local memory, in which each bank has 2 read and 1 write port on the memory side, and a row of PEs sharing memory access on the PE array side.

Thus each PE can access data in any bank, through a hardware arbiter between the memory requests generated by the PEs and the banks of the memory. We call such an architecture, which 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 [Singh et al. 2000; Mei et al. 2003; Kim et al. 2005].

Even in the MBA architecture, a fundamental restriction remains that a bank cannot be accessed by two different PEs at the same time. This is the challenge that we first address in this article. Fundamentally, there are two solutions to this. One is the hardware solution, that is, add a request queue in the arbiter and increase the access latency of the memory operation, and the other is to change the application, mapping technique to explicitly consider the memory bank architecture, and map memory operations into rows such 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.

In addition to the tight integration and optimization of data mapping into compiler, we present a compiler optimization that can reduce the number of memory operations for CGRA mapping by exploiting data reuse. Reducing the number of memory operations can fundamentally reduce the possibility and frequency of bank conflict. While eliminating some memory operations through data reuse is done routinely in compilers for conventional microarchitectures, including VLIW processors [Wolf and Lam 1991], both for straightline code and for loops, previous techniques relied on central register files to pass data across iterations, which is hardly applicable to CGRAs because CGRAs often have few or no central registers. Our memory operation-reduction technique converts expensive load operations into inexpensive data move operations without using central registers, which can be directly mapped to CGRAs through data routing. To do so, we extended the application graph semantics by introducing a new edge type. The new edge type requires special treatment during mapping, which is supported by our extended mapping algorithm.

Our experiments on important multimedia kernels demonstrate that our memory-aware compilation approach generates mappings that are up to 53% (26% on average) better than the state-of-the-art memory-unaware scheduler. As compared to the hardware approach using arbiters, our technique is on average 24.9% better, and promises to be a good alternative.

The rest of the aricle is organized as follows. We first describe our architecture model in Section 2, followed by a brief discussion of related work in Section 3. We present the problem of memory-aware mapping for CGRA in Section 4, and present two compilation techniques in Section 5 and Section 6. We discuss our experimental results in Section 7, and conclude article in Section 8.

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, although 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. Additionally, some PEs can 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 [Mei et al. 2003; Singh et al. 2000; Kim et al. 2005].

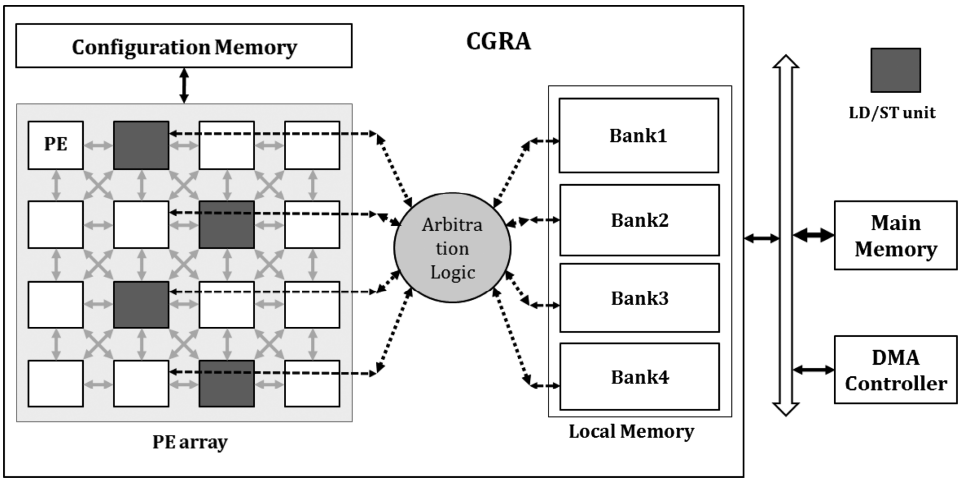


Fig. 2. Multibank with arbitration (MBA) architecture: There is arbitration between the PE array and the memory banks, so that any PE can access data from any bank. However a fundamental limitation still remains: Two PEs cannot access data in the same bank simultaneously.

The local memory of a CGRA is typically a high-speed, high-bandwidth, highly predictable random access memory (such as SRAM) that provides temporary storage space for array data, which often consist of the input/output of loops that are mapped to CGRAs. To provide high bandwidth, local memories are often organized in multiple banks (e.g., MorphoSys [Singh et al. 2000]). 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, which the has arbiters in front of the memory ports of multiple banks, MBA (multiple banks with arbitration) architecture.

Even in an 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 article we assume that a bank consist of single-port cells, and thus has only one read-write port.) In MBA architecture, if two PEs try to access the same bank at the same time, a *bank conflict* occurs. CGRA hardware that supports 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 Bougard et al. [2008] uses, in front of each bank, a hardware queue, called DMQ or DAMQ (dynamically allocated, multi-queue buffer) [Tamir and Frazier 1992]. Although 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, a 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

[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 simultaneously.

architecture. In this article 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 the 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 can thus 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. Hence our compiler approach uses sequential array mapping.[2]

3. RELATED WORK

CGRA memory architectures can be largely classified into implicit load-store architecture (e.g., MorphoSys [Singh et al. 2000] and RSPA [Kim et al. 2005]) and explicit load-store architecture (e.g., ADRES [Mei et al. 2003; Bougard et al. 2008]). Whereas implicit load-store architectures have data (array elements) prearranged 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 [Singh et al. 2000; Kim et al. 2005] assumed a 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 target architecture assumes explicit load-store with muxes and, optionally, queues.

Most previous CGRA mapping approaches [Lee et al. 2003b; 2003a; Mei et al. 2002; Park et al. 2006; Hatanaka and Bagherzadeh 2007; Ahn et al. 2006; Yoon et al. 2008; Shields 2001; Park et al. 2008; Venkataramani et al. 2001; Oh et al. 2009], consider computation mapping, but not data mapping. Yoon et al. [2008] consider computation mapping as a graph embedding problem from a data-flow graph into a PE interconnection

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

graph, and solves the problem using a known graph algorithm. Many others consider mapping as a scheduling problem targeting an array processor, a simpler form of which is a VLIW processor. Software pipelining and modulo scheduling are often used. Park et al. [2008] proposes an improved variant of modulo scheduling by considering edges rather than nodes (of an input data-flow graph) as the unit of scheduling. Oh et al. [2009] propose a scheduling algorithm for loops with recurrence relation (interiteration data dependence). In all these approaches, data mapping is only an afterthought, and is not included in the optimization framework.

The essence of the problem in our work is how to schedule loops in the presence of tight memory constraints. While this has been partially addressed, at least for VLIW architectures, both with software pipelining [Wang et al. 2009; Xue et al. 2008] and without [Li et al. 2003], and some of them are even very similar to our load-reduction technique in terms of the general approach and motivation, there are important differences between our work and the VLIW scheduling work, mainly due to the fundamental differences in the target architectures. First, CGRAs typically lack a central register file, and therefore replacing a costly memory load with a cheaper register read simply doesn't work. An alternative may be to store the reused data in the network of PEs, but that generally requires careful rerouting of data, which might not always be profitable. Second, the idea of generating conflict-free schedules that ensure the maximal performance on the local memory side is not applicable if the processor cannot directly access multiple banks, as in the case of VLIW processors accessing memory through one or more caches.

There is little work that consider memory during CGRA mapping. Dimitroulakos et al. [2005] considers a hierarchical memory architecture and present a mapping algorithm to reduce the amount of data transfer between L1 and L2 local memory. Dimitroulakos et al. [2009] propose routing reused data through PEs instead of using the local memory, which can reduce the local memory traffic, and thus improve performance. The idea of reusing data through PE routing to save memory resources is further extended in our load reduction technique, which exploits PE routing for flow dependence (RAW) as well as input dependence (RAR), and thus can be more effective. On the other hand, our memory-aware mapping, the core of which is conflict-free scheduling, is orthogonal to this, and the effect is additive when applied together. Dimitroulakos et al. [2009] also consider the data layout with sequential memory. But this is limited, as it considers only one loop, and is based on simulated annealing; extending it to multiple loops does not seem straightforward. Lee et al. [2008] propose the idea of quickly evaluating memory architectures for CGRA mapping; however, this proposal lacks a detailed mapping algorithm. Our earlier work [Kim et al. 2010] also proposes a mapping algorithm that takes data mapping as well as computation mapping into account; however, the memory architecture assumed is much simpler, with no arbiter or queues. This article is an extension of Kim et al. [2010], with more thorough experimental results and a new optimization, *load reduction*, to reduce memory operations and thereby increase performance on multibank memories.

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,** that is, mapping from each operation of the loops to a specific PE (where) and to which schedule step (when, in cycle); and (ii) **data mapping,** tht is, mapping from each array in the loops to the bank of the local memory to be used. A loop is represented by the data-flow graph of the loop body, along with data dependence information and memory reference information (i.e., 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 weighted average of the IIs, with the weights given by the user (e.g., the number of iterations).

Hence, the goal of our problem is to minimize the average II. In addition to the usual constraints for computation mapping (e.g., Park et al. [2008]), memory-aware mapping has additional constraints. One thing to understand so as 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.[3] If there is an expected stall in the optimal mapping, we 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—this 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 cycles, there must be at most n accesses to each bank in every n consecutive cycles.

5. CONFLICT AVOIDANCE: ELIMINATING BANK CONFLICTS

5.1. Overview

The main challenge of our problem comes from the interdependence between computation mapping and data mapping; that is, fixing data mapping creates constraints on computation mapping, and vice versa. Due to the interdependence, 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 Yoon et al. [2008], spatial mapping is solved by ILP formulation. In this experiment, the ILP solver could not find a solution within a day if the number of nodes exceeds 13. Our problem is also intractable, since it can be reduced to the computation temporal mapping problem, whose complexity is at least as high as that of spatial mapping. Hence we propose a heuristic that solves them sequentially, first clustering data arrays to balance utilization and access frequency of each bank, then finding computation mapping through conflict-free modulo scheduling.

Figure 3 illustrates the overall flow of our heuristic mapping approach. First, we perform load reduction (Section 6). We next perform a premapping, which is just computation mapping by traditional modulo scheduling without considering data mapping. The II resulting from premapping 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 (Section 5.2). Premapping can provide a tighter lower bound for II than traditional minimum II calculation, considering

[3]Sequential array placement is the only method supported by some CGRAs (e.g., MorphoSys [Singh et al. 2000]), as it does not require costly hardware arbiters. In sequential array placement, large arrays that cannot fit in a single bank can be handled by splitting the loop iterations, or *loop tiling*, where only one tile is executed per CGRA invocation. Across different tiles array may be mapped in the same bank or in different banks.

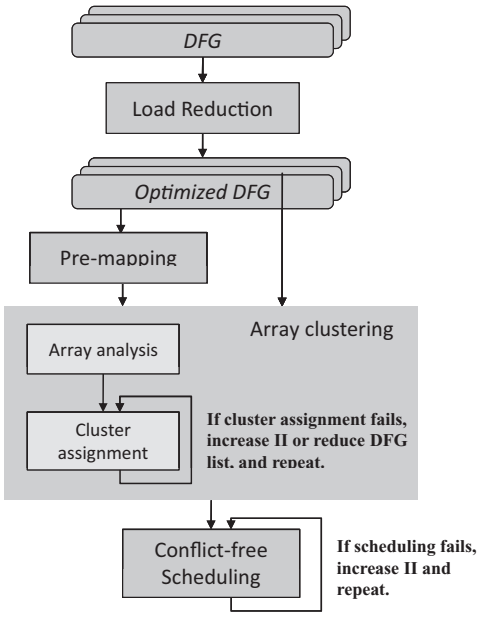


Fig. 3. Our CGRA compilation flow.

resource and recurrence requirements only, thus helping 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, although 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 arrays in the same set are mapped in the same bank.

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,[4] the effective total size of the local memory is reduced, and consequently there is a higher chance of having to reload arrays during a loop execution[5] 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.

[4]A rather extreme example of this is: all arrays are mapped to only two of the four banks available in the architecture while the other two banks are not used at all.

[5]Reloading arrays during a loop execution must happen if the total array size of a loop is greater than the effective local memory size.

5.2.1. ILP Formulation. It would be ideal to formulate the entire problem, including both array clustering and computation mapping; into one ILP problem; but this would be very complicated. Instead, here we consider the problem of array clustering only.

Consider a sequence of L loops, which collectively use N arrays, our problem is to allocate one of M banks for each array. In this problem formulation, i is used to refer to an array ($i = 0, \ldots, N-1$), j to a bank ($j = 0, \ldots, M-1$), and k to a loop ($k = 0, \ldots, L-1$).

The input parameters are

—s_i: size of array i;
—c_j: size of bank j;
—d_{ik}: access count of array i in loop k;
—w_k: relative weight (importance) of loop k;
—MII_k: minimum II of loop k from memory-unaware scheduling.

Binary decision variables: x_{ij}, which is 1 if array i is mapped to bank j, otherwise 0.

The objective is to minimize the weighted sum of memMII's (memory-constrained minimum II) for all loops. The memMII of loop k is given as the maximum per-iteration access count among all the banks, or $memMII_k = \max_j \sum_i d_{ik} \cdot x_{ij}$. This is equivalent to saying that

$$memMII_k \geq \sum_i d_{ik} \cdot x_{ij} \quad \text{for } \forall j, k. \tag{1}$$

Another constraint on the memMIIs is that they cannot be smaller than minimum MII from memory-unaware scheduling.

$$memMII_k \geq MII_k \quad \text{for } \forall k. \tag{2}$$

From the definition of x_{ij}:

$$\sum_i x_{ij} = 1 \quad \text{for } \forall j. \tag{3}$$

Finally, the bank size constraint:

$$\sum_i s_i x_{ij} \leq c_j \quad \text{for } \forall j. \tag{4}$$

Combining (1) through (4) gives all the necessary constraints for our ILP formulation.

5.2.2. Heuristic. We combine the two factors, viz., array size and array access count, into one metric, priority. Our clustering algorithm takes one array at a time and determines its clustering by assigning a cluster to it. Due to 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,

$$priority_A = Size_A/SzBank + \sum_{\forall L} Acc_A^L/II_L', \tag{5}$$

where $Size_A$ is the size of array A and $SzBank$ is the size of a bank. To best utilize the limited bank size and the limited total access count for a bank (per II), we assign a higher priority to an array that is larger in size or has a higher access count than others.

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 C and an array A, the relative

ALGORITHM 1: Array_cluster(*MII*, *DFGs*)

```
1: clear array_to_cluster mapping info
2: extract the list of arrays(=AL) from DFGs
3: calcPriority(AL, MII)
4: for each array1 in AL in the descending order of priority do
5:    errorcode ← calc_cost_and_assign_min_cost_cluster(array1, MII)
6:    if errorcode is not SUCCESS then
7:       return errorcode
8:    end if
9: end for
10: return SUCCESS
```

ALGORITHM 2: Array_cluster_main(*MII*, *DFGs*)

```
1: loop
2:    errorcode ← array_cluster(MII, DFGs);
3:    switch ( errorcode ) do
4:       case notEnoughMem: reduce DFGs
5:       case notEnoughAccCount: increase MII
6:       default: return
7:    end switch
8: end loop
```

cost of assigning C to A is

$$cost(C, A) = Size_A/SzSlack_C + \sum_{\forall L} Acc_A^L/AccSlack_C^L, \tag{6}$$

where $SzSlack_C$ and $AccSlack_C^L$ 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. We use the remaining values to calculate cost, since the balancing requirement dictates that if one bank's size or access count is used up too much in relation to the other banks, we should avoid assigning arrays to that bank.

Algorithm 1 shows the pseudo code of our array-clustering algorithm. After extracting the list of arrays used in the loops represented by the DFGs (data flow graphs), we first compute the priority of each array. Next 2 the arrays are mapped, in decreasing order of priority, to a cluster with the minimum cost using the cost metric of (6). If any array fails to find an available cluster, array clustering is aborted, and started anew with a different set of parameters (Algorithm 2). If the failure is due to a lack in the access count budget, we increase the initial II, but if it is due to the lack of memory capacity, we reduce the number of loops.

The resulting II after array clustering is called memMII (memory-constrained minimum II), which depends on the number of accesses to each bank (per iteration) and memory access throughput (per cycle). In the previous modulo scheduling algorithm [Park et al. 2008], MII (minimum II) is determined only from resMII (resource-constrained minimum II) and recMII (recurrence-constrained minimum II). In our approach, however, we consider memMII as well, defining MII to be the largest of resMII, recMII, and memMII.

Figure 4 illustrates our array clustering heuristic. Figure 4(a) shows part of array analysis results (access frequency analysis).[6] Once array priority is calculated

[6]Array sizes are not shown because they are the same in this example, and the following parameters are used: the II after premapping is 5 for loop1 and 6 for loop2, a memory bank size is 5 times as large as one array size.

Array name	#Access (per iter)	
	Swim Loop 1	Swim Loop 2
p	4	-
u	3	-
v	3	-
cu	1	4
cv	1	4
z__	1	3
h__	1	3
uold	-	1
vold	-	1
pold	-	1
unew	-	1
vnew	-	1
pnew	-	1

(a) Array access frequency analysis

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

(b) Prioritization

Array name	Cost				Assigned Cluster
	Cl 0	Cl1	Cl2	Cl3	
cv	1.07	1.07	1.07	1.07	0
cu	x	1.07	1.07	1.07	3
p	1.25	1	1	1.25	2
h__	x	0.9	1.75	x	1
z__	x	1.5	1.75	x	1
u	1	x	x	1	0
v	x	x	x	1	3
pnew	0.83	x	0.42	0.83	2
pold	0.83	x	0.53	0.83	2
unew	0.83	x	0.75	0.83	2
uold	0.83	x	1.33	0.83	0
vnew	1.5	x	1.33	0.83	3
vold	1.5	x	1.33	1.5	2

(c) Cost calculation and cluster selection

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

(d) Clustering result

<swim loop1>

	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 loop2>

	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

(e) Number of accesses for each loop

Fig. 4. Array-clustering example. From the array information (all arrays have the same size, thus are not shown), priorities are first determined, followed by clustering by our heuristic algorithm. The lower two tables summarize key statistics after clustering.

(Figure 4(b)), the minimum-cost clusters are assigned to arrays, in decreasing order of array priority (Figure 4(c)), resulting in the clustering shown in Figure 4(d). Figure 4(e) 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 could occur even when array clustering was done first. This 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 a necessary condition for a conflict-free mapping only. In other words, once array clustering is done, the total access count of a bank does not change because of scheduling, but temporary access count can change. For instance if two memory operations accessing the same bank are scheduled at the same cycle, two load-store units will spontaneously try to access the same bank, which is a bank conflict. Thus, we extend a previous modulo scheduling algorithm [Park et al. 2008] developed for CGRAs to generate a conflict-free mapping.

5.3.1. Base Scheduling Algorithm. In this work we use EMS (edge-centric modulo scheduling) [Park et al. 2008], a known scheduling algorithm, as our base modulo scheduler. Unlike other schedulers, which place nodes first, followed by placement of routing paths, EMS tries routing from, source node first. During routing, if the routing path passes through a place where the where the destination node can be placed, the placement is decided at this time. Algorithm 3 shows a pseudo code of our base scheduler. In this article, we use several costs for placement decisions, which are widely used in mapping algorithms such as resource cost, routing cost, and relativity cost. The resource cost is the cost for using a PE for node placement. Its costs vary different according to the function of the PE. If the target PE is expensive, such as the PE having a memory access unit, the cost is set higher. The routing cost is the cost for routing data to the destination. The cost grows bigger if the routing path is long or passes through one or more expensive PEs. The relativity cost is used for placing related nodes at adjacent PEs. To these basic modulo scheduling environments, we added our approaches.

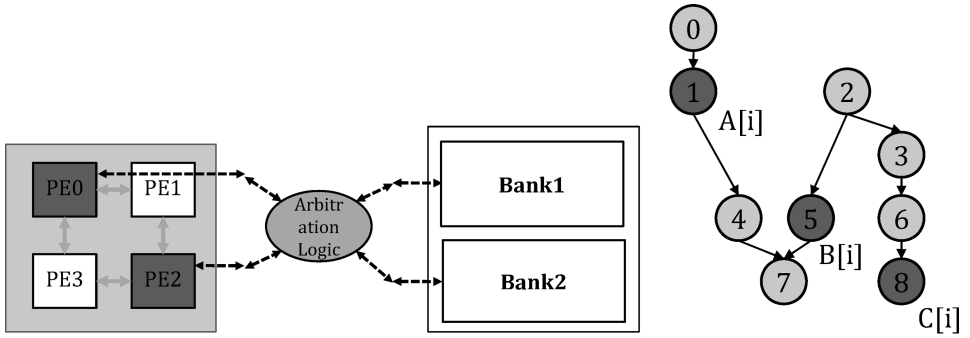
ALGORITHM 3: Modified-modulo-scheduling-algorithm (*DFG*, *MII*)

```
1: NodeList ← prioritize-nodes
2: for (II = MII; II < MaxII; II + +) do
3:    for each Node in NodeList do
4:       for each Place in SearchSpace(Node) do
5:          if isSpaceAvailable(Place) and isRoutable(Place) then
6:             calculate resource, routing, and relativity cost of Place
7:          end if
8:       end for
9:       if minimum-cost Place is found then
10:         select the minimum-cost placement and routing for Node
11:      else
12:         break // increase II and repeat
13:      end if
14:   end for  // break the outer loop if successful
15: end for
```

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 modulo scheduling include only PEs and interconnects, our extension treats memory banks, or memory ports to the banks, as resources, too.[7] This small extension, combined with our array clustering, allows us to find conflict-free mapping.

Memory conflict will occur if there are two memory accesses to the same bank at the same cycle. This means that one of the memory accesses cannot be completed on time, and every PE must stall for one cycle. Our strategy then is, since we have the array clustering information, we can avoid memory conflict by keeping track of and utilizing

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



(a) CGRA architecture (b) DFG

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

(c) Array-mapping result

	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.

Fig. 5. Conflict-free scheduling.

the schedule information as to when and where memory operations are mapped. We maintain cluster access information along with PE allocation information during our operation scheduling, so that two memory operations belonging to the same cluster may not be mapped on the same cycle.

Figure 5 illustrates our conflict-free scheduling algorithm via an example shown in Figure 5(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 5(c) with the target II of 3, and that all the nodes from 1 through 7 have been scheduled as shown in Figure 5(d), and now node 8, a memory operation, is about to be scheduled. The first candidate for node 8 is cycle 4 on PE2, which has a conflict, not in terms of computation resources (PE2 was not used in cycle 1 or 4), but in terms of memory resources (CL1 was already used in cycle 1). Thus choosing the first candidate means a bank conflict,

or one stall, per every II, effectively increasing II by one. Alternatively, node 8 can be scheduled at cycle 6 on PE0, albeit via a longer route. But since this choice does not cause any conflict for computation or memory resources, effectively II is not increased, resulting in much better performance than that of 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 the EMS algorithm, our clustering-aware scheduling approach is implemented in the isSpaceAvailable function (Algorithm 3 line 5). The spaceAvailable function checks several conditions to confirm that the space is available. The conflict-free approach just adds one more condition, so there is no hard problem in unifying the previous scheduling algorithm with our approach.

5.3.3. MBAQ Architecture. Multiple accesses to the same bank at the same cycle create bank conflict. The MBA architecture handles this problem by having all PEs stall until the conflict is resolved. A DMAQ architecture is a hardware solution to reduce stalls [Bougard et al. 2008; Bouwens 2006]. DMAQ architecture adds an arbitration logic with a queue for each bank, which necessarily increases memory load latency in terms of cycles, compared to what is achievable without such queues. During the additional cycles, the load data is temporarily stored in the arbitration queue, until, at the end of the load latency, the data is delivered to a PE.

In this manner, several accesses to the same array can be handled without processor stall by fetching data earlier, to give extra time for fetching other conflicted data. But the previous work assumed an interleaving memory architecture. However, in our case, we can predict bank conflict, so MBAQ architecture is used for relaxing the mapping constraint. MBA architecture doesn't permit bank conflict, but MBAQ architecture can permit several conflicts within a range of added memory operation latences. We distinguish two cases (n is the added memory operation latency cycle via the MBAQ approach)

(i) $II' \leq n$ (Target II is less than or equal to the DMQ length): Our array clustering guarantees that there are at most II' accesses per 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 the data layout. To ensure the absence of a 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 the DMAQ memory interface.

6. LOAD REDUCTION: REMOVING MEMORY OPERATIONS

Removing memory operations can fundamentally reduce the chances of bank conflicts. Also, since load operations typically have higher latency than simple arithmetic operations, removing them can help achieve higher performance. Further, a memory operation is usually accompanied by a chain of arithmetic operations to calculate the memory address, and therefore removing one memory operation gives an opportunity to eliminate all dependent arithmetic operations as well. All these give a strong motivation to reduce the number of memory operations before CGRA mapping.

6.1. Optimization Opportunity

An opportunity to remove memory operations can arise when there is a group of memory references with small offset differences. For example, Figure 6(a) has two array references with offsets that only differ by one. So the trailing reference $A[i-1]$ will access the same array element as the leading reference $A[i]$, only one iteration later. Thus, the idea of our optimization, dubbed *load reduction*, is to remove the trailing memory operation and provide the data from the leading memory operation. The optimization in this case can effectively reduce the number of load operations per iteration from 2 to 1.

Note that if two references are exactly the same, duplicates it will have already been removed by the compiler front-ends, as they are considered common expressions. Typical compiler optimizations also include loop-invariant code motion, which moves out of the loop the memory operations that are constant, or independent of the loop induction variable (iterator). Thus, our optimization targets only those references with group reuse [Wolf and Lam 1991].

We only consider pairs of references with group reuse, of which the trailing one is a read, thus either RAR (Read After Read) or RAW (Read After Write). If the trailing reference is a write, either it cannot be removed (WAR), or the leading reference is unnecessary and would be easily optimized away (WAW). Another case where our optimization is not applicable is when there may be an interfering write between two references connected with group reuse. While finding out whether two references may interfere with each other, or point to the same array element, is a computationally demanding problem, we conservatively assume that two references interfere if they are to the same array and have different strides. Such a case rarely happens in practice, thus even with our conservative assumption, we could find a fair number of candidates for our optimization.

For references of the same stride the, it is easy to check whether there is group reuse between them: group reuse exists if the stride divides the offset difference. We call the offset difference divided by the stride, *reuse distance* of the reference pair. The larger the reuse distance, the more difficult it becomes for the leading reference to provide the data for the trailing one. For a very large reuse distance, reusing the data might not even be profitable, since routing the data between the two memory operations also takes resources. Hence we select only those reference pairs with a short reuse distance, which is controlled by design parameter D (reuse distance upper limit). The profitable range of the reuse distance depends on many factors, including II, the number of registers in a PE, memory operation latency, and access frequency.

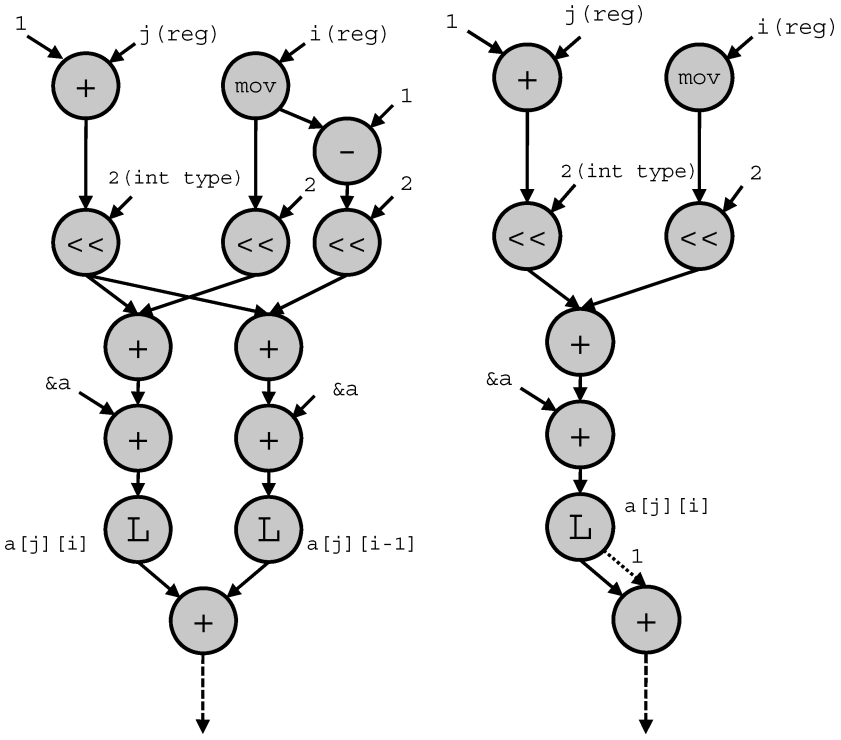
6.2. Graph Transformation

Once we identify reference pairs to which to apply our load reduction optimization, the next step is to modify the DFG to remove unnecessary operations. The procedure follows

For each identified reference pair,
(1) Remove the trailing memory operation (i.e., load).
(2) Remove all the operations that are used only to provide the address for the trailing load.
(3) Create one or more new edges, called *reuse edges*, from the leading memory operation to the successors of the trailing operation. If the leading operation is a store, use the data-side predecessor instead.
(4) Annotate the reuse edges with the reuse distance of the reference pair.

```
for(i=0 ; i<100 ; i++) {
… = a[j][i] + a[j][i-1];
…
}
```

(a) Example source code

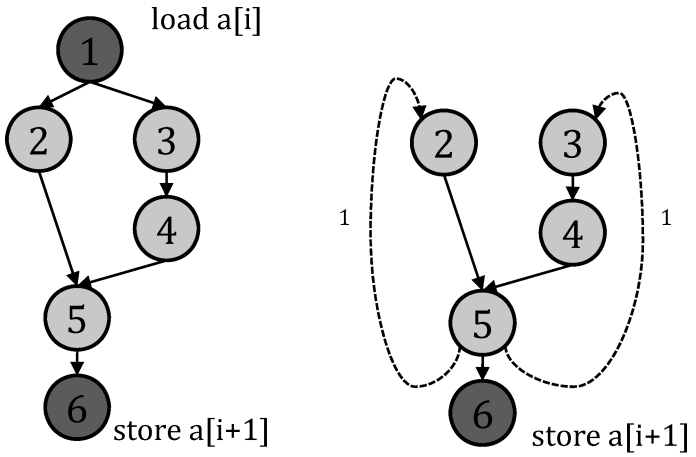


(b) Generated DFG (c) Optimized DFG

Fig. 6. DFG transformation.

Figure 6 illustrates the DFG transformation; Figure 6(b) is the original DFG for the code shown in Figure 6(a); Figure 6(c) is the DFG after the transformation. The trailing memory operation is removed and a connection between the leading one and the successor of the one trailing is made, with its weight set to the reuse distance.

In addition to the advantages already mentioned, our memory optimization has another important advantage, especially for RAW cases. In a RAW loop, the data for the trailing load operation can *only* come from the leading store operation, or actually its data-side predecessor, which usually gives the strongest constraint (i.e., recMII) on the achievable II for modulo scheduling. Without load reduction, the data must first be stored to the local memory and then copied back to the CGRA array, which can considerably increase recMII compared to simply routing the data through the CGRA array. Applying our load reduction optimization can help reduce recMII. Figure 7 illustrates how recMII can be decreased by our optimization. Assuming that every operation (including load/store) takes one cycle, recMII of the original DFG in



(a) Original DFG (b) Optimized DFG

Fig. 7. Optimization advantage: recMII reduction (address calculation operations are omitted for simplicity).

Figure 7(a) is 5, from the 1-3-4-5-6-1 cycle. In contrast, the optimized DFG in Figure 7(b) has a lower recMII, which is 3 from the 3-4-5-3 cycle. Here we assumed the latency of every operation, including memory operations, is 1; but in reality, memory load operations usually have higher latency, so recMII reduction is expected to be larger for real architectures. Thus, our load reduction optimization can help increase the performance of CGRAs, especially for loops with interiteration data dependence. Reuse edges thus introduced need special treatment during mapping, which is explained in the next section.

6.3. Routing Reuse Edge

Reuse edges, introduced by our load reduction optimization, need special treatment during modulo scheduling. By the time a reuse edge is about to be mapped, the source node of the reuse edge must be scheduled. Then, we replace the source node of the reuse edge with a new one. The new one is whatever is placed on the same PE as the old one, but N cycles earlier, where N is the product of reuse distance (or edge weight) and II'.

Figure 8 shows the process of mapping a DFG, including a reuse edge; Figure 8(a) shows the optimized DFG, and it will be mapped on the Figure 5(a) architecture; Figure 8(b) shows the mapping state that nodes from 0 to 5 are already mapped. The next node to be mapped is node 6. This node gets inputs from node 3 and node 4. But the edge connecting node 3 and node 6 is a reuse edge whose iteration distance is 1. So as you can see in Figure 8(c), the current iteration's node 4 and the previous iteration's node 3 are predecessors of node 6. So after the placement and routing algorithm, the placement of node 6 is decided as PE0 at t = 3. Figure 8(d) shows the final result of mapping. The nodes with a solid line show the mapping result and the nodes with a dotted line show the nodes that are executed concurrently due to modulo scheduling.

7. EXPERIMENTS

7.1. Setup

For the target architecture we use a CGRA that is very similar to the one illustrated in Figure 2. It has four load-store units the locations shown in the figure. The local memory has four banks, each of which has one read/write port. Arbitration logic allows

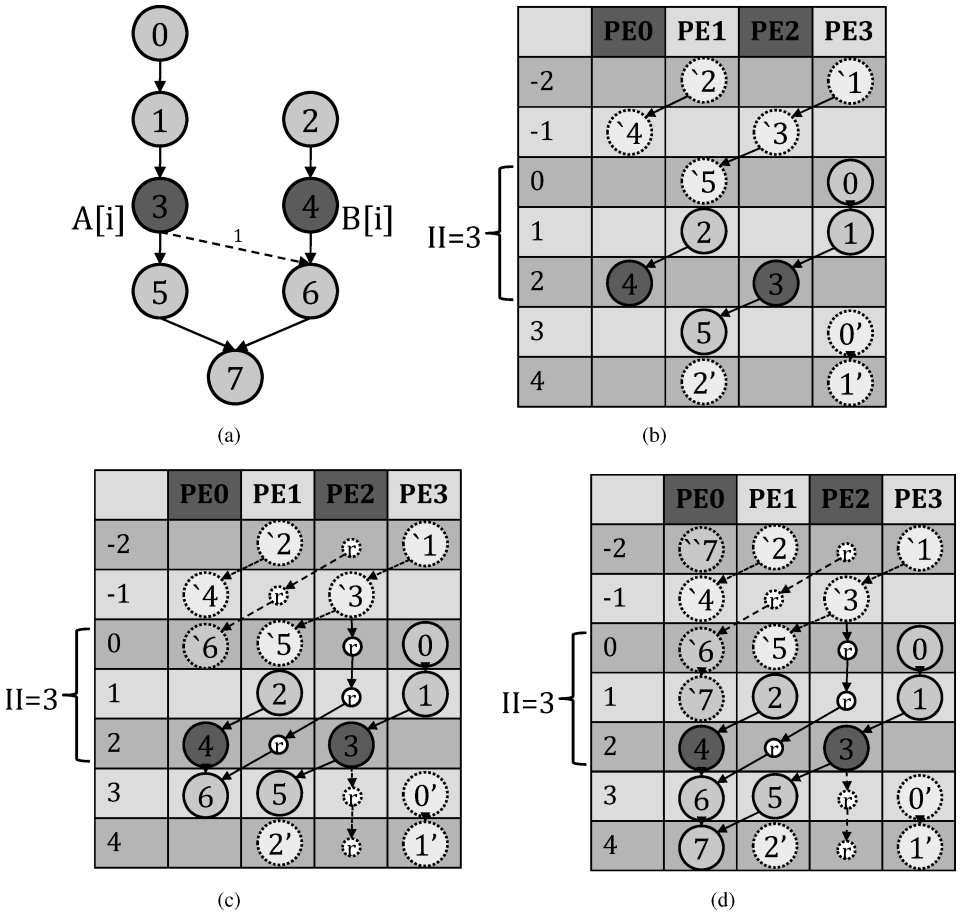


Fig. 8. An example of routing reuse edge (‘n means node n of the previous iteration, n’ means node n of the next iteration).

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 four, 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 four 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 ran 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 (nonstall) cycles. Because of the randomness in the scheduling algorithm (as when there is more than one minimum cost candidate), we compile and simulate each loop ten times and the average performance is taken as the representative performance of the algorithm for that loop.

7.2. Effectiveness of Conflict-Avoidance Only

To see the effectiveness of our compiler-based conflict-avoidance approach, we compare our conflict-avoidance mapping with the hardware approach that uses DMQ to reduce

Table II. Comparing Average II (Initiation Interval) between ILP and Our Heuristic

	Swim			InvResidual				CopyImg		CopyFrame		SetRef		Init_mbaff		Lapl ace	SOR		Low pass	GSR	Comp ress	Sum
ILP	8.3	9.9	5	5.6	4.9	2	3	2	2.4	2	3.6	4.8	4.8	4	4	9	3.5	11	11	7	6	113.8
Heuristic	8.6	9.9	5	5.7	4.8	2	3	2	2.1	2	3.2	4.7	4.6	4	4	9	3.6	11	11	7	6	113.2

bank conflict. For the hardware approach we use an existing modulo scheduling algorithm [Park et al. 2008], which is referred to as memory-unaware scheduling (MUS). MUS is also used as the base scheduler for our memory-aware scheduling (MAS). We do not apply load reduction optimization to this set of experiments.

We compare three architecture-compiler combinations. The first one, the *baseline*, is the combination of MUS with a hardware arbiter only. Having no DMQ, a stall occurs whenever there is more than one request to the same bank. Bank conflict is detected and resolved at runtime. Interleaved 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 our heuristic for array placement and MAS to generate conflict-free mapping. Only the 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.

7.2.1. ILP vs. Our Heuristic. To evaluate the performance of our heuristic array placement algorithm for our compiler approach, we first compare the result of ILP vs. our heuristic algorithm. Since the ILP formulation covers only the array placement part, the objective of which is to minimize the memMII, it is fair to compare it in terms of memMII. On the other hand, memMII is not a conclusive performance metric; only the final II is. Therefore we also compare them in terms of the final II for each loop. The result is that, for all the loops in our experiments, both methods yield the same result in terms of memMII. Moreover, in terms of the final II, they do not show any significant difference, as shown in Table II. The average difference is less than 1%, which should be attributed to the undeterministic nature of the scheduling algorithm. These results indicate that our heuristic algorithm can find near-optimal array placements that maximize the memory performance of CGRAs.

7.2.2. Compiler Approach vs. Hardware Approach. Figure 9 compares the runtime results for the three cases, 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 $10 \sim 40\%$ 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 nonstall times are mostly the same as in the baseline case, with a few exceptions. The notable increase in the nonstall time of the hardware scheme in some applications is due to the increase in load latency. Especially in some applications with recurrence edges (GSR, compress etc.), the hardware scheme shows a great increase in runtime increase. As those applications' executions are bounded by recMII, the load latency increase, which affects recMII, is critical for these cases. In the case of nonrecurrent loops, the hardware scheme can reduce the expected CGRA runtime, although not always, by $0 \sim 27\%$ (10.8% on average) compared to the baseline case. Overall, the hardware scheme reduces runtime by 2.1% on average. With our compiler approach the stall time is completely removed. The increase in the nonstall 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 in most cases can achieve far better performance (10 to 40% runtime reduction, 16.9% on average)

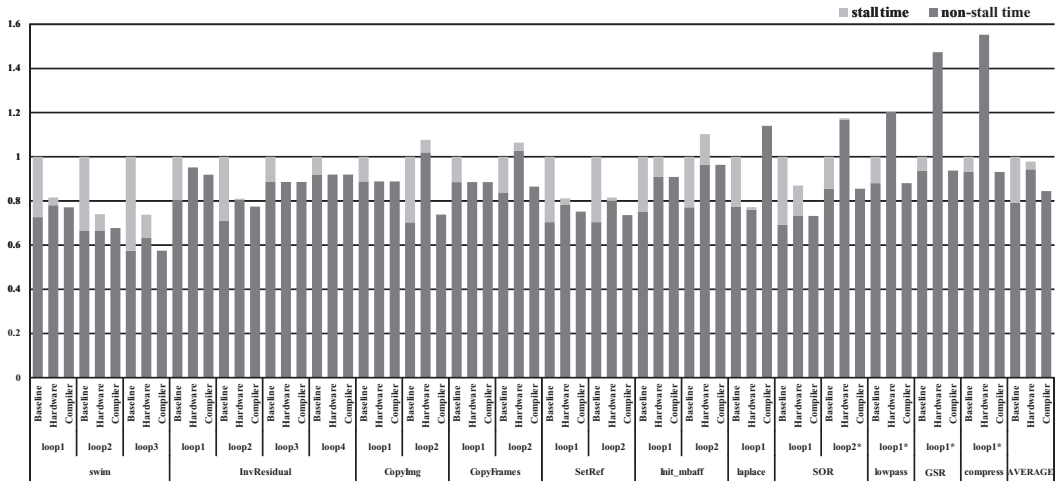


Fig. 9. Runtime comparison, normalized to the baseline case (MUS w/o DMQ). Total execution time is the sum of non-stall time and stall time. Our compiler approach (MAS) completely eliminates stall time in all cases, and can achieve up to 40% better performance than the baseline. The asterisk (*) indicates a recurrent loop, where nonstall time can increase due to interiteration data dependence. The Y-axis represents a normalized runtime.

Table III. Average Initiation Intervals (II) (of loops mapped, individually vs together (smaller is better))

	Individually	Together	II Reduction
Swim L1	8.6	8.6	0.0%
Swim L2	11	9.9	10.0%
Swim L3	8	5	37.5%

compared to memory unaware mapping. Our approach also allows the removal of bank conflict resolution hardware, which can contribute to reducing the cost and energy consumption. Further, compared to the hardware approach using DMQ, our approach can deliver better performance in most loops (on average 13.7% runtime reduction). Even compared with the hardware approach on nonrecurrent loops, our approach shows better performance (on average 6.7%). 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 manyfold.

7.2.3. Multiple Loops in Data Mapping. Often, a sequence of loops that appear together share data arrays between them. An output array of a loop may be an input in a succeeding loop, or an array may be used as input in multiple loops. Thus there is a strong motivation to consider multiple loops together when deciding array placement, as in our mapping flow. To see the effectiveness of multiloop data mapping, we apply our array placement heuristic individually to each loop vs. to all the loops of a benchmark at once, followed by the same MAS scheduling in both cases.

The result for the swim benchmark is shown in Table III in terms of the final II (the other applications show no significant difference). As expected, considering multiple loops together in data mapping can result in significantly better performance, especially for loops with a large number of arrays. Interestingly, the difference seems to increase for later loops in a loop sequence, because, for the first loop in a sequence, single-loop data mapping has no more constraints —and therefore should be no worse— than multiloop data mapping, and the constraints build up only as the following loops are mapped.

Table IV. DFG Statistics (with and without LR optimization. #N is # of all operations or #nodes in the DFG, #M is # of memory operations, #RE is # of reuse edges, and LRD is the largest reuse distance. Asterisk indicates recurrent loop)

	Before Optimization				After Optimization					
Kernel	#N	#M	recII	resII	#N	#M	recII	resII	#RE	LRD
Swim L1	65	14	–	5	57	10	–	4	10	1
Swim L2	78	20	–	5	66	14	–	5	9	1
Laplace L1	53	10	–	4	35	4	–	3	6	2
SOR L2*	59	12	11	4	51	10	7	4	3	1
Lowpass L1*	57	10	11	4	40	5	7	3	6	2
GSR L1*	34	6	7	3	29	5	3	2	1	1
Compress L1*	25	5	6	2	17	3	2	2	2	1

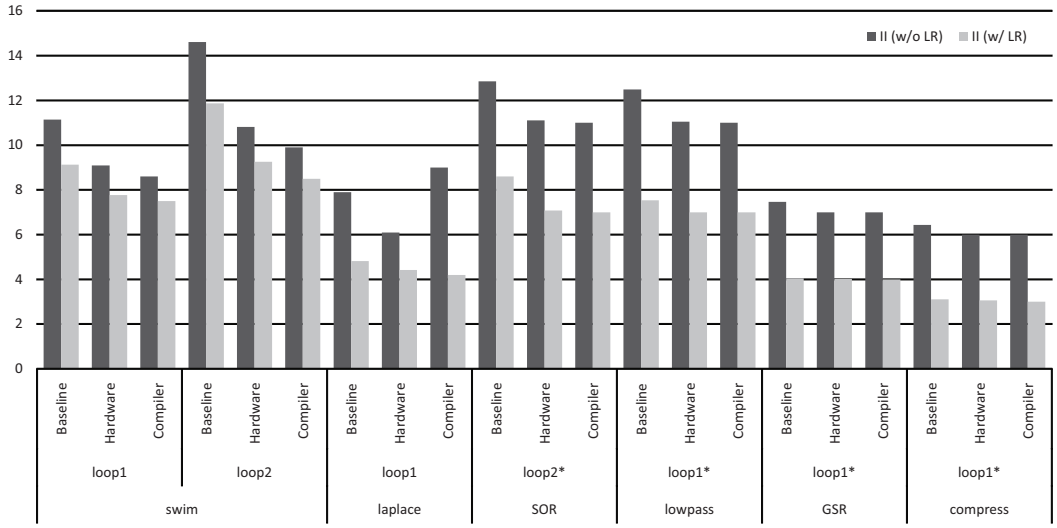


Fig. 10. Comparing runtimes, with vs. without LR optimization, for the three approaches; The asterisk indicates a recurrent loops. The y-axis represents averaged II.

7.3. Effectiveness of Load Reduction

Next, we evaluate the effectiveness of our load reduction (LR) optimization. We apply LR optimization to all the loops with a data reuse opportunity. Table IV compares DFG statistics for those loops, with and without LR optimization. First, for all the loops with a data reuse opportunity, our LR optimization can consistently reduce the number of memory operations by 34%, on average, and up to 60%. While this reduction in the number of memory operations can certainly contribute to reducing bank conflicts, and thereby improving the performance of CGRA, the impact of this reduction alone on overall DFG scheduling is rather small, since memory operations account for only a small portion (about 20%) of the entire number of operations. Second, our LR optimization can remove non-memory operations as well, usually many more than memory operations. This helps reduce the total number of DFG operations by 21% on average, and up to 34%. As we show in the next graph (Figure 10), this contributes to significant performance improvement. Third, the biggest difference due to our LR optimization is observed in recMII. The recMII in this analysis assumes a MBA architecture with a 3-cycle load latency. Even then we can observe an outstanding decrease in recMII due to our LR optimization. Lastly, the last column lists the largest reuse distance we exploit in our optimization, which is very small. This means that the overhead of handling reuse edges during mapping will not be large.

Table V. Comparing II Results, With and Without RAW

Kernel	RAR only	RAR and RAW	II reduction
SOR L2	11	7	36.4%
Lowpass L1	11	7	36.4%
GSR L1	7	4	42.9%
Compress L1	6	3	50.0%
Average	–	–	41.4%

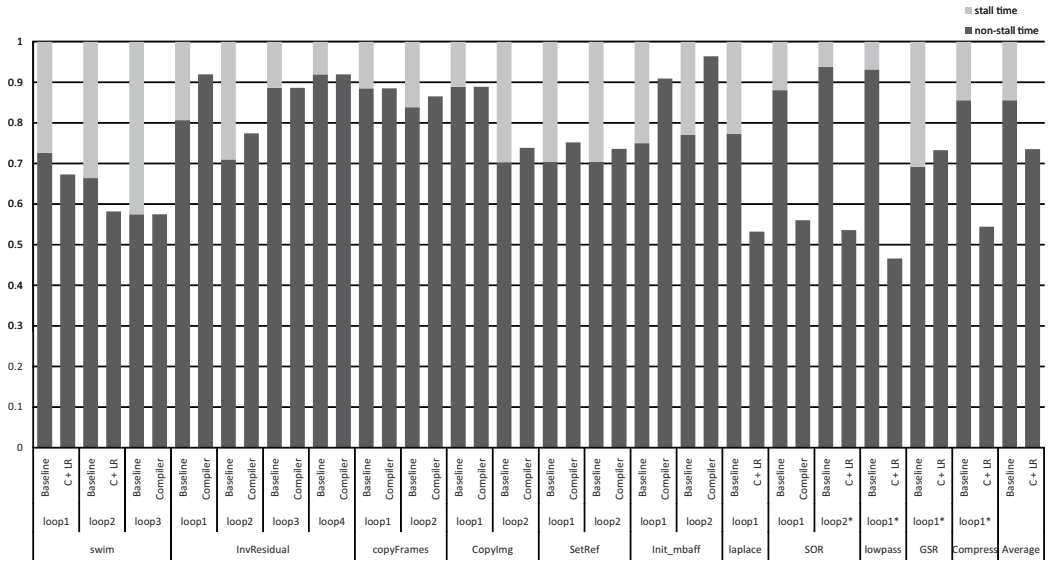


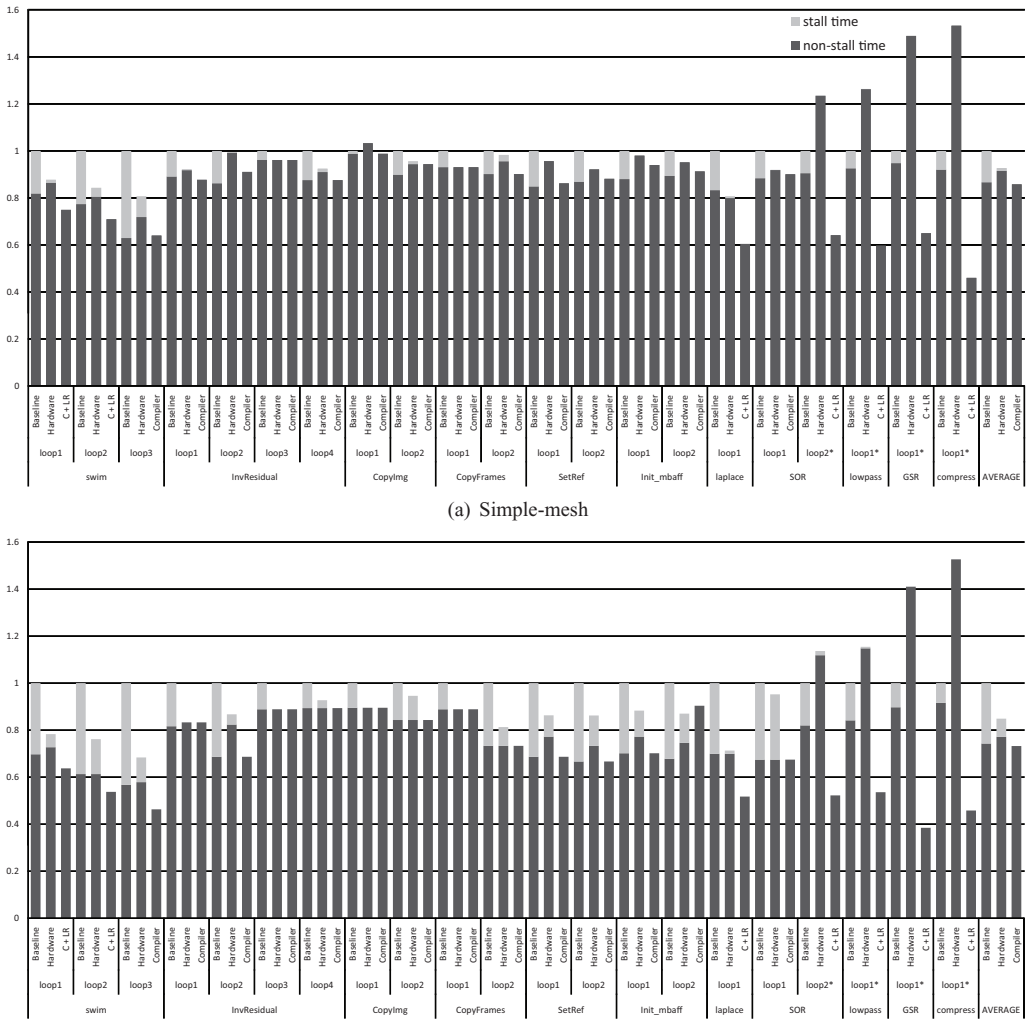
Fig. 11. Comparing runtimes for baseline vs. compiler + LR; The asterisk indicates recurrent loop; The y-axis represents runtime normalized to that of baseline.

Figure 10 shows the advantage of using our LR optimization in terms of expected II (taking stalls into account), for the three approaches compared (i.e., baseline, hardware, and compiler). The graph indicates that, as expected, our LR optimization can improve performance in all cases where there is reuse opportunity, and its effectiveness is orthogonal to the underlying mapping approach.

One difference between our load reduction optimization and other similar techniques such as that of Dimitroulakos et al. [2009] is that ours can handle flow dependence, (RAW) as well as input data reuse (RAR). To see the importance of considering flow dependence, we evaluate our technique with and without it. Table V compares the two cases considering RAR only vs. both RAR and RAW, in terms of II for all recurrent loops. (For nonrecurrent loops the two cases become identical, since only recurrent loops have RAW dependence.) As can be seen from the table, considering RAW dependence for recurrent loops in addition can be quite rewarding, reducing II by $35 \sim 50\%$, compared to considering input data reuse only.

7.4. Effectiveness of Our Combined Compiler Approach

Figure 11 compares our combined compiler approach (conflict avoidance + load reduction) against the baseline approach in terms of runtime. Our approach uses the same target architecture as the baseline. However, our approach can completely eliminate stalls, and generate much higher performance throughout all applications. The runtime improvement by our techniques is more pronounced in the case of recurrent loops, but is significant in nonrecurrent loops as well. Overall, the runtime improvement is



(a) Simple-mesh

(b) Simple-mesh + diagonal + 1-hop

Fig. 12. Runtime comparison between baseline vs. compiler + LR for multiple CGRA architectures (differing in the interconnection); the y-axis represents the runtime of the compiler + LR case, normalized to that of the baseline.

26% on average, and up to 53%. The runtime improvement compared to the hardware approach is 24.9%, on average.

Figure 12 compares the three approaches on different CGRA architectures. Whereas the interconnect for Figure 11 includes simple-mesh and diagonal only, for Figure 12(a) it is varied to a less efficient one (simple-mesh only), and for Figure 12(b) to a more efficient one (simple-mesh plus diagonal plus one-hop). For those two new interconnect architectures, the runtime improvement of our compiler approach compared to the baseline is 14% and 27%, on average, while, it is 26%, on average, for our initial interconnect architecture. Thus, a trend can be observed, that the relative benefit of our approach over the baseline increases as the interconnect becomes more efficient, which, however, seems to end after the simple-mesh plus diagonal. This trend can be explained by the fact that memory becomes more of a performance bottleneck as the

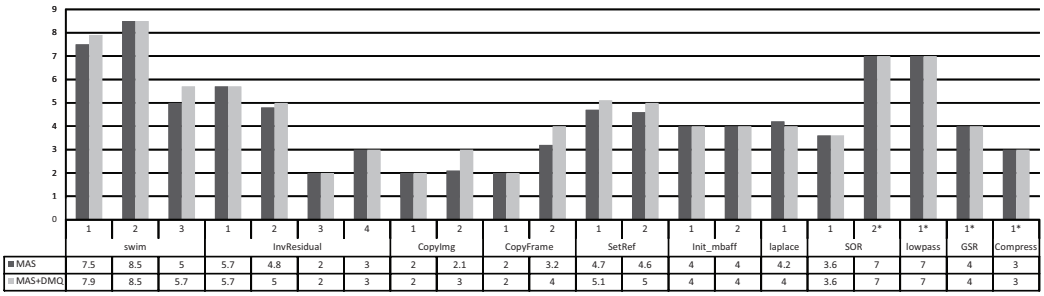


Fig. 13. Our combined compiler approach, with and without DMQ; y-axis represents II.

PE array becomes more efficient due to complex interconnects, in which case memory-aware mapping such as ours has an advantage. This experiment also reinforces the importance of considering memory during CGRA mapping, especially for CGRAs with more efficient interconnect architectures.

7.5. Effect of Using DMQ with Our Approach

Since DMQ can reduce bank conflicts and increase performance when used with a conventional memory-unaware scheduler, it is interesting to see how effective it is with our memory-aware compilation flow. Figure 13 compares the II (average of ten trials) by our memory-aware compiler with/without DMQ. Here, we apply our LR optimization in both cases. 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-aware mapping. Mostly-the II is the same, and in some cases the II is actually increased 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.

8. CONCLUSION

We presented a data-mapping aspect of CGRA compilation. Our first focus is 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 in performance. To reduce or eliminate bank conflicts, either the hardware or compiler approach can be used. We define the mapping problem for the compiler approach, and also propose a heuristic approach, since the problem is computationally intractable, like many problems in compilation. Second, we also propose load reduction (LR) optimization. Our LR optimization can eliminate dependent load operations from the DFG by providing the required data through routing, which can also enable elimination of related non-memory operations and contribute to a large improvement in performance. Our experiments demonstrate that our memory-aware compilation approach can generate mappings that are up to 53% better in performance (on average, 26%) compared to the memory-unaware scheduler for the same architecture. Even when compared to the hardware approach using DMQ, our memory-aware approach is on average 24.9% better, and can be a good alternative to the hardware solution. Moreover, our compiler guarantees that all the mappings are free of bank conflicts, which can be used to eliminate conflict-resolution hardware, which is another advantage of our approach.

REFERENCES

AHN, M., YOON, J. W., PAEK, Y., KIM, Y., KIEMB, M., AND CHOI, K. 2006. A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'06)*. European Design and Automation Association, Leuven, Belgium, 363–368.

BOUGARD, B., DE SUTTER, B., VERKEST, D., VAN DER PERRE, L., AND LAUWEREINS, R. 2008. A coarse-grained array accelerator for software-defined radio baseband processing. *IEEE Micro 28*, 4, 41–50.

BOUWENS, F. 2006. Power and performance optimization for Adres. M.S. dissertation, Delft University of Technology.

DIMITROULAKOS, G., GEORGIOPOULOS, S., GALANIS, M. D., AND GOUTIS, C. E. 2009. Resource aware mapping on coarse-grained reconfigurable arrays. *Microprocess. Microsyst. 33*, 2, 91–105.

DIMITROULAKOS, G., GALANIS, M.D., AND GOUTIS, C.E. 2005. Alleviating the data memory bandwidth bottleneck in coarse-grained reconfigurable arrays. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05)*. IEEE, Los Alamitos, CA, 161–168.

HATANAKA, A. AND BAGHERZADEH, N. 2007. A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In *Proceedings of the Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, Los Alamitos, CA, 1–8.

KIM, Y., LEE, J., SHRIVASTAVA, A., AND PAEK, Y. 2010a. Operation and data mapping for CGRAs with multi-bank memory. *ACM SIGPLAN Not. 45*, 4, 17–26.

KIM, Y., LEE, J., SHRIVASTAVA, A., YOON, J. W., AND PAEK, Y. 2010b. Memory-aware application mapping on coarse-grained reconfigurable arrays. In *Proceedings of HiPEAC'10*. Lecture Notes in Computer Science, vol. 5952, Springer, Berlin, 171–185.

KIM, Y., KIEMB, M., PARK, C., JUNG, J., AND CHOI, K. 2005. Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'05)*. IEEE, Los Alamitos, CA, 12–17.

LEE, J.-E., CHOI, K., AND DUTT, N. D. 2008. Evaluating memory architectures for media applications on coarse-grained reconfigurable architectures. *IJES 3*, 3, 119–127.

LEE, J.-E., CHOI, K., AND DUTT, N. D. 2003a. An algorithm for mapping loops onto coarse-grained reconfigurable architectures. In *Proceedings of the ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems (LCTES'03)*. ACM, New York, 183–188.

LEE, J.-E., CHOI, K., AND DUTT, N. D. 2003b. Compilation approach for coarse-grained reconfigurable architectures. *IEEE Des. Test 20*, 1, 26–33.

LI, S., LAI, E. M.-K., AND ABSAR, M. J. 2003. Analysis of cooperation semantics for transaction processing with full and partial aborts in active database. In *Proceedings of the 4th IEEE International Pacific Rim Conference on Multimedia, Information, Communication and Signal Processing (ICICS/PCM'03)*. IEEE, Los Alamitos, CA.

MEI, B., VERNALDE, S., VERKEST, D., DE MAN, H., AND LAUWEREINS, R. 2003. Adres: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Field Programmable Logic and Application,* Lecture Notes in Computer Science, vol. 2778, 61–70.

MEI, B., VERNALDE, S., VERKEST, D., DE MAN, H., AND LAUWEREINS, R. 2002. DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*. IEEE, Los Alamitos, CA, 166–173.

OH, T., EGGER, B., PARK, H., AND MAHLKE, S. 2009. Recurrence cycle-aware modulo scheduling for coarse-grained reconfigurable architectures. *SIGPLAN Not. 44*, 7, 21–30.

PARK, H., FAN, K., MAHLKE, S. A., OH, T., KIM, H., AND KIM, H.-S. 2008. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. ACM, New York, 166–176.

PARK, H., FAN, K., KUDLUR, M., AND MAHLKE, S. 2006. Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06)*. ACM, New York, 136–146.

SHIELDS., C. O., JR. 2001. Area efficient layouts of binary trees in grids. Ph.D. dissertation, University of Texas at Dallas.

SINGH, H., LU, G., FILHO, E., MAESTRE, R., LEE, M.-H., KURDAHI, F., AND BAGHERZADEH, N. 2000. Morphosys: Case study of a reconfigurable computing system targeting multimedia applications. In *Proceedings of the 37th Annual Design Automation Conference (DAC'00)*. ACM, New York, 573–578.

TAMIR, Y. AND FRAZIER, G. L. 1992. Dynamically-allocated multi-queue buffers for VLSI communication switches. *IEEE Trans. Comput. 41*, 6, 725–737.

VENKATARAMANI, G., NAJJAR, W., KURDAHI, F., BAGHERZADEH, N., AND BOHM, W. 2001. A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'01)*. ACM, New York, 116–125.

WANG, M., LIU, D., WANG, Y., AND SHAO, Z. 2009. Loop scheduling with memory access reduction under register constraints for DSP applications. In *Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE, Los Alamitos, CA, 139–144.

WOLF, M. E. AND LAM, M. S. 1991. A data locality optimizing algorithm. *ACM SIGPLAN Not. 26*, 6, 30–44.

XUE, C. J., SHA, E. H.-M., SHAO, Z., AND QIU, M. 2008. Effective loop partitioning and scheduling under memory and register dual constraints. In *Proceedings of the Conference on Design, Automation and Test In Europe (DATE'08)*. ACM, New York, 1202–1207.

YOON, J. W., SHRIVASTAVA, A., PARK, S., AHN, M., JEYAPAUL, R., AND PAEK, Y. 2008. SPKM: A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'08)*. IEEE, Los Alamitos, CA, 776–782.

Received May 2010; revised October 2010; accepted April 2011