High Throughput Data Mapping for Coarse-Grained Reconfigurable Architectures

Yongjoo Kim, Jongeun Lee, *Member, IEEE,* Aviral Shrivastava, *Member, IEEE,* Jonghee W. Yoon, Doosan Cho, *Member, IEEE,* Yunheung Paek, *Member, IEEE*

***Abstract*—Coarse-grained reconfigurable arrays (CGRAs) are a very promising platform, providing both up to 10–100 MOps/mW of power efficiency and software programmability. However, this promise of CGRAs critically hinges on the effectiveness of application mapping onto CGRA platforms. While previous solutions have greatly improved the computation speed, they have largely ignored the impact of the local memory architecture on the achievable power and performance. This paper motivates the need for memory-aware application mapping for CGRAs, and proposes an effective solution for application mapping that considers the effects of various memory architecture parameters including the number of banks, local memory size, and the communication bandwidth between the local memory and the external main memory. Further we propose efficient methods to handle dependent data on a double-buffering local memory, which is necessary for recurrent loops. Our proposed solution achieves 59% reduction in the energy-delay product, which factors into about 47% and 22% reduction in the energy consumption and runtime, respectively, as compared to memory-unaware mapping for realistic local memory architectures. We also show that our scheme scales across a range of applications and memory parameters, and the runtime overhead of handling recurrent loops by our proposed methods can be less than 1%.**

***Index Terms*—Array mapping, bank conflict, coarse-grained reconfigurable architecture, compilation, multi-bank memory.**

Manuscript received April 28, 2011; accepted June 3, 2011. Date of current version October 19, 2011. This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science, and Technology (MEST), under Grant 2010-0011534, in part by the Korea Science and Engineering Foundation (KOSEF) NRL Program grant funded by the MEST, under Grant 2011-0018609, in part by the Engineering Research Center of Excellence Program of MEST/KOSEF, under Grant 2011-0000975, in part by the Sunchon National University Research Fund in 2011, in part by the Basic Science Research Program, under Grant 2010-0024529, through the National Research Foundation of Korea (NRF) funded by the MEST, and the IDEC, and in part by funding from the National Science Foundation, under Grants CCF-0916652 and CCF-1055094 (CAREER), in part by the NSF I/UCRC for Embedded Systems, under Grant IIP-0856090, Raytheon, Intel, Microsoft Research, SFAz, and Stardust Foundation. This paper was recommended by Associate Editor M. Hutton.

Y. Kim, J. W. Yoon, and Y. Paek are with the Department of Electrical Engineering and Computer Science, Seoul National University, Seoul 151 742, Korea.

J. Lee is with the School of Electrical and Computer Engineering, Ulsan National Institute of Science and Technology, Ulsan 689-805, Korea (e-mail: jlee@unist.ac.kr).

A. Shrivastava is with the Department of Computer Science and Engineering, School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Phoenix, AZ 85051 USA.

D. Cho is with the Department of Electronic Engineering, Sunchon National University, Sunchon 540-742, Korea.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2011.2161217

I. Introduction

COARSE-GRAINED reconfigurable arrays, or CGRAs, promise up to 10–100 MOps/mW of energy efficiency [1] while still retaining programmability. Being essentially an array of processing elements (PEs), where each PE can be programmed to execute word-level operations, such as arithmetic and logic operations, CGRAs are especially suited for multimedia and compute-intensive applications, whereas field-programmable gate arrays (FPGAs) can be more appropriate for applications involving complex logic and bit manipulation. Another benefit of the coarser granularity is that programming CGRAs with *configuration bitstream*, or simply *configuration*, can be done much more quickly (and even during runtime) than programming FPGAs. Generating configurations for a given application is called *application mapping*, or *compilation*, and it is one of the biggest challenges for CGRAs.

Compilation for CGRAs has traditionally focused on two issues [2]–[11]: 1) placing *operations* (such as arithmetic/logic, multiplication, and load/store) of a loop kernel onto the PE array, and 2) guaranteeing the data flow, or *communication*, between operations using the existing interconnection resources. Then the loop is essentially turned into a "pipeline" on a CGRA, completing one iteration every cycle, or every nth cycle in general, where n is the initiation interval (II) of the pipeline [12]. The input and output data are stored in the local memory of the CGRA, which is a small on-chip SRAM with very high bandwidth to the PE array. To sustain fast computation rate on the PE array, managing the input and output data on the CGRA's local memory becomes a critical issue.

There are two aspects to data management. The first is how to place data, typically large array variables, in the limited local memory of a CGRA. We refer to the problem of placing arrays on the CGRA local memory as *data mapping*, whereas placing operations onto the PE array is called *computation mapping*. Data mapping may not be an issue for a simple local memory architecture consisting of a single large bank,[1] or in general for a uniform memory access (UMA) architecture. However, for more realistic local memory architectures with

[1]Alternatively, one can use a dynamic hardware support such as a crossbar switch to provide uniform access to multiple banks [13], which may have disadvantages such as limited scalability and higher cost. On the other hand, using multiple banks with static mapping limits the applicability to loops with sequential memory accesses.

multiple ports and multiple banks, or in general for nonuniform memory access (NUMA) architectures such as those used in architecture for dynamically reconfigurable embedded system (ADRES) [13] and MorphoSys [14], data mapping can affect the quality of computation mapping and even limit the performance of the overall mapping.

The other aspect is how to make data transfers in and out of the local memory very efficient, since CGRAs are typically used as coprocessors to main processor and therefore the input data for a CGRA must eventually come from the main memory over the system bus, which may take long. To hide the long data transfer latency as well as to overcome the limited local memory size, data transfer is often done simultaneously along with computation in an overlapping fashion on a *double-buffered* local memory, as is the case with MorphoSys [14]. Consequently, the overall throughput on such CGRAs is *not* determined by computation rate only, but by the *minimum* of computation rate and data transfer rate. This means that we can increase the overall performance by sacrificing computation mapping a little if that buys us improved data mapping and data transfer rate—provided that the computation rate was initially higher than the data transfer rate, or in other words the loop was initially memory-bound. Another complication that arises from hardware double buffering on CGRAs is that it creates for application mapping a nontrivial problem of correctly and efficiently handling dependent data for loops with inter-iteration dependence, such as infinite impulse response (IIR) filters. To the best of the authors' knowledge this problem of handling recurrent loops on CGRAs with double-buffered local memories is not previously addressed.

In this paper, we present an application mapping flow for CGRAs that considers both computation and data aspects of the mapping so as to maximize the overall performance, as opposed to maximizing computation rate. Our proposal introduces new costs such as data reuse opportunity cost (DROC) and bank balancing cost (BBC) to steer the mapping process to be more aware of the architectural peculiarities. In addition, for CGRAs with double-buffered local memories, we present methods to correctly handle inter-iteration dependent data of recurrent loops. Our experimental results indicate that not only is our proposed mapping heuristic able to achieve near-optimal results as compared to single-bank memory mapping but it can also achieve 59% reduction in the energy-delay product as compared to memory-unaware mapping for multi-bank memory, which factors to 47% and 22% reductions in the energy consumption and runtime, respectively. We also demonstrate that our scheme scales across a range of applications, and memory parameters.

The rest of this paper is organized as follows. In Section II, we review the related work. After describing the target architecture in Section III, we explain in Section IV the challenges of considering computation and data mapping together. In Section V, we present our memory-aware heuristic that can be applied to any modular scheduling algorithm. In Section VI, we present novel methods to handle recurrent loops correctly on a double-buffering local memory. In Section VII, we present our experimental results and conclude this paper in Section VIII.

II. Related Work

Earlier research on CGRAs was mostly about architecture design [15], but with the recognition that application mapping is the bottleneck, recent work increasingly focuses on application mapping techniques.

A. Architecture Focus

Data transfer architectures between local memory and PEs can be classified into implicit load-store and explicit load-store architecture. Implicit load-store CGRA architectures, e.g., MorphoSys [14], do not have explicit load and store instructions. Data has to be pre-arranged in the local memory, organized like a queue, and the topmost element is broadcast to the CGRA every cycle. On the other hand, in explicit load-store CGRAs, e.g., ADRES [16], PEs can explicitly compute the address of the memory location that they intend to access, and read/write to that location. While the implicit load-store architectures are potentially much more power efficient, they are more challenging to program, and also incur penalties relating to the efforts required to arrange the data in a very specific order in the local store.

Local memory can be designed as single-bank or multi-bank. Single-bank memory makes programming much easier; however, it is very difficult to provide all the necessary ports for the PE array with just one bank. One solution is to use multi-port static random-access memory cells, which are however extremely expensive in terms of area, power, and speed [17]. With multi-bank memory, it is the responsibility of the programmer/compiler to make sure that the data a PE accesses is present in a bank that it has access to. Alternatively, one can use hardware arbitration to make every bank accessible to any PE [13], which makes the local memory design more complicated with higher power, area, and possibly cycle time compared to multi-bank memory without hardware arbitration. Our work provides a software solution rather than a hardware solution to the problem of managing multi-bank memory.

Hardware double buffering, e.g., MorphoSys [14] and resource sharing and pipelining in coarse-grained reconfigurable architecture (RSPA) [18], can speed up the data transfer between the system memory and the CGRA local memory, while some architectures, e.g., ADRES [13], opt for a single large buffer. Double buffering becomes more useful if the local memory size is smaller, or the loops and arrays of the applications are larger. We assume explicit load-store, multi-bank, and double-buffered local memory in this paper.

Our partial shutdown exploration requires the resources of a CGRA to be split and put into different modes of operation. A similar idea and detailed mechanism are presented in [19], which applies the technique to running multiple loops simultaneously on the same CGRA.

B. Compilation Focus

Most previous work on application mapping for CGRA [2]–[11] does not explicitly consider the local memory architecture or data placement. They assume that all the required data is already present in the local memory, and every load-store PE can access that data whenever they need to. Even with such

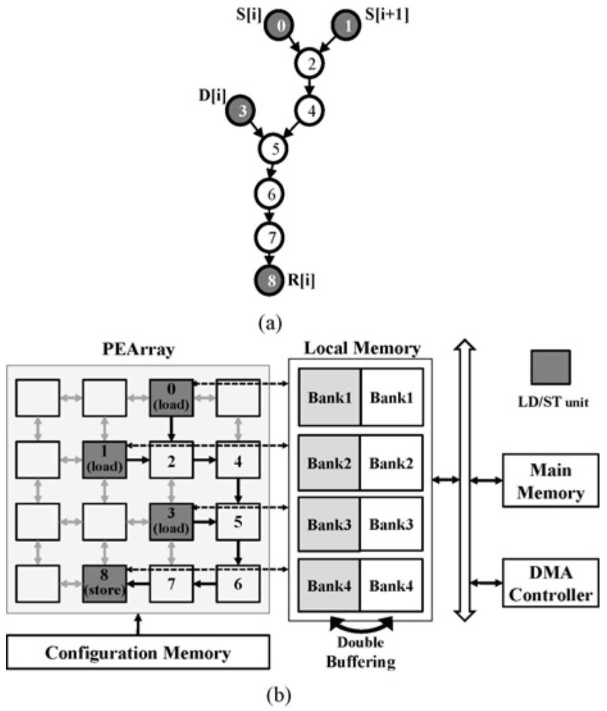


Fig. 1. CGRA architecture and application mapping. (a) DFG of a loop from MPEG2. (b) Mapping the DFG onto CGRA. Dark nodes in (a) and dark PEs in (b) represent memory operations and load-store PEs. Though a single configuration is enough to map the DFG of (a), in general, multiple configurations may be used.

a simplification, the application mapping problem for CGRA is shown to be very hard [7], having to deal with operation placement on a 2-D array considering the communication between them (*spatial mapping*) [8], as well as possibly changing configurations every cycle (*temporal mapping*) [4], [9].

One exception to this is [20], which assumes a hierarchical memory architecture, where the PEs are connected to a L0 local memory, which connects to the external main memory through an L1 local memory. Since both these local memories are scratchpads, and therefore statically scheduled, their main interest is in improving the reuse between the L0 and L1 local memories. An early work [21] on CGRA presents a methodology to evaluate memory architectures for CGRA mapping; however, it lacks a detailed mapping algorithm. Reference [22] also considers memory architecture for mapping, and is therefore most closely related to our paper. However, their mapping assumes multi-bank memory with arbitration logic and single buffering, and therefore is not applicable to our target architecture while we explore the impact of partitioned, or multi-banked, memory architecture and also explore the impact of limited memory bandwidth on the mapping. The idea of balancing computation rate and data transfer rate on CGRAs with double buffering was presented in [23], which is extended in this paper to handle recurrent loops by efficiently managing dependent data on buffer switches.

III. Target Architecture

CGRA is essentially an array of PEs connected through a mesh-like interconnects, as illustrated in Fig. 1(b). The local memory is an important part of CGRA, and is the only memory that can be directly accessed by a PE. The set of operations of a PE and the interconnections between PEs can vary in different CGRA designs, but basic operations such as arithmetic/logic operations are typically performed by any PE whereas costly operations such as memory operations and multiplications can be performed only by some PEs. The PEs that can perform memory operations are called *load-store PEs*. In a multi-bank local memory architecture, there may be a one-to-one mapping between a row of PEs and a bank, such that a bank may be accessed only by the PEs in the corresponding row. In the ADRES architecture, for instance, there is only one load-store PE per row in a 4×4 PE array (either at one end or in the middle), and consequently each load-store PE has its own bank, which can be accessed only by the corresponding PE.

The local memory is limited in size. Hence hardware support to manage data on the local memory is critical to achieve high performance. First, direct memory access (DMA) is used to transfer data between the local memory and main memory, at the command of main processor. Second, hardware double buffering is used to allow simultaneous accesses by both DMA and the PE array (i.e., load-store PEs). Double buffering works as follows, with two buffers denoted as A and B. While the PE array uses buffer A to access the array data for the current loop, DMA uses buffer B to write back the output arrays of the previous loop and to bring in the input arrays for the next loop. Then in the next loop, the other buffer is used for each (called *buffer switching*). If the arrays for a loop do not fit in one buffer, the loop can be broken into smaller ones—executing only the first T iterations on one buffer followed by the next T iterations to be executed on the alternate buffer, and so on. Note that from the compiler point of view, this is equivalent to *tiling* the loop with *tile size* of T. Due to such restrictions, and because we do not assume dynamic hardware such as a crossbar switch between load-store PEs and bank memories [13], only loops with sequential memory accesses can be supported by our static mapping.

IV. Problem Description

We first explain the basics of application mapping and discuss the challenges of our problem.

1) *Application Mapping:* CGRAs are typically used to accelerate the innermost loops of applications, thereby saving runtime and energy. The innermost loop of a perfectly nested loop can be represented as a data flow graph (DFG) [Fig. 1(a)], in which the nodes represent micro-operations (arithmetic and logic operations, multiplication, and load/store), and the edges represent the data dependency between the operations. While not for this loop, the data dependency can be in general loop-carried. The task of mapping an application onto a CGRA traditionally comprises of: 1) mapping the nodes of the DFG onto the PE array of the CGRA, and 2) mapping the edges onto the connections between the PEs. Since the mesh-like interconnection can be restrictive for application mapping, most CGRAs allow PEs to be used for routing of data (*routing PE*); the routing PE does not perform any operation, but just transfers one of the inputs to its output. This flexibility can

be exploited by allowing the edges in the DFG to be mapped onto paths in the CGRA.

2) *Software Pipelining:* Pipelining is explicit in the CGRA, in the sense that the result of computation inside one PE can be used by the neighboring PEs in the next cycle. For effective application mapping the compiler must software-pipeline the loop before mapping it onto the the PEs. Modulo scheduling [12], one of the most effective algorithms to perform software pipelining, tries to find a valid schedule for a given target II (initiation interval) by using "modulo resource table," which can easily keep track of modulo resource constraints generated as scheduling progresses. If no valid schedule is found, the target II is incremented, and the whole procedure is repeated.

3) *Challenges of the Problem:* Thus in addition to the problem of expressing the application in terms of the functionality of PEs, a CGRA compiler must explicitly perform resource allocation, software pipelining, and routing of data dependencies on the CGRA. It is for these reasons that the problem of application mapping on CGRA is challenging. Furthermore, to consider data placement during CGRA mapping we must maximize *both* computation rate and data transfer rate at the same time. Simply maximizing data transfer rate is trivial; for instance, fixing the placement of all the arrays beforehand will do, but it may decrease computation rate excessively. Considering computation rate only is what has been typically done in previous approaches, which may fail to maximize the overall throughput. Moreover, data mapping should be emphasized only if the application is memory-bound, which adds to the complexity of our problem. Thus, our CGRA mapping problem considering both computation and data mapping is more complicated than the traditional CGRA mapping problem considering computation only, which is already NP-hard [7]. Hence, we propose a heuristic in this paper. We will also demonstrate through our experiments that our heuristic can achieve near-optimal results for many loops.

V. Simultaneous Data and Computation Mapping

We now present our approach to simultaneous data and computation mapping. Our heuristic considers: 1) minimizing duplicate arrays (or maximizing data reuse); 2) balancing bank utilization; and 3) balancing computation and data transfer rates. A unique feature of our heuristic is that it merely defines some cost functions for those memory-related considerations, rather than prescribing a whole new algorithm, so that our heuristic can be easily integrated with other existing memory-unaware mapping algorithms. While our technique is generally applicable to any modular scheduling algorithm considering one operation at a time such as [4] and [9]; for the sake of the discussion, we use the edge-centric modulo scheduling (EMS) algorithm [9] in this paper as it is one of the best known.

A. Balancing Computation and Data Transfer

To balance optimization effort for computation and data parts we first perform *performance bottleneck analysis*. Performance bottleneck analysis determines whether it is computation or data transfer that limits the overall performance. We define the data-transfer-to-computation time ratio (DCR) as $DCR = t_d/t_c$. For this we generate an initial, memory-unaware

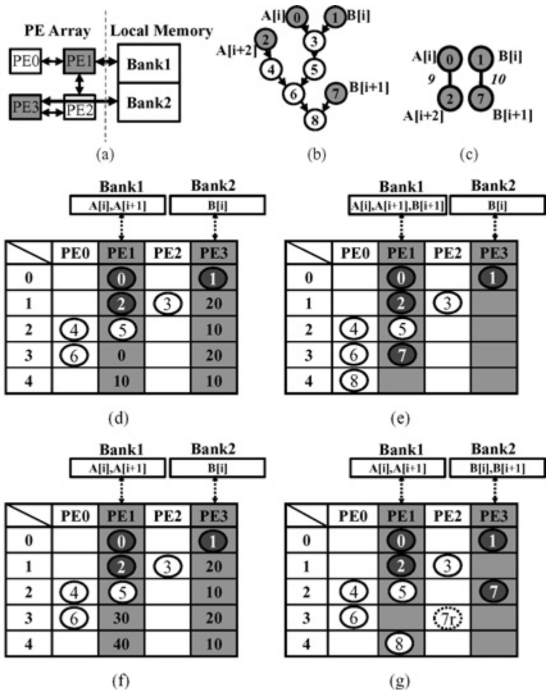


Fig. 2. Data reuse example. Mapping operation 7 to PE3 allows the reuse of array B between operations 1 and 7. Assuming: base routing cost = 10, DCR = 3. (a) CGRA architecture. (b) DFG. (c) DRG. (d) Cost values for operation 7 by EMS. (e) Completed mapping by EMS. (f) Cost values for operation 7 by ours. (g) Completed mapping by ours.

mapping, and t_d is calculated as the time to transfer the input and output data of the loop between the local memory and main memory through DMA and $t_c = II_u \cdot N_{iter}$, where II_u is the II of the memory-unaware mapping and N_{iter} is the number of iterations of the loop. A loop is memory-bound if $DCR > 1$, and roughly represents the optimization opportunity for our memory-aware mapping.

B. Maximizing Data Reuse

Temporal reuse of data, or the use of the same data or array elements in different iterations of a loop, is frequently found in many loop kernels. Temporal as well as spatial reuse is automatically exploited by data caches for general purpose processors; however, for CGRAs everything must be explicitly controlled by compilers. Traditional compilation flows for CGRA, which are memory unaware, do not treat specially arrays with reuse. As a result, two load operations, even if they read from the same array, will typically be mapped to different rows. Note that this is not an issue of functional correctness, but of performance in NUMA CGRAs, since duplicating the arrays in multiple banks solves the correctness problem. An alternative approach is to realize reuse by mapping to the same row all the load operations accessing the same array, which we call *reuse through the local memory*.[2]

[2]When there is data reuse between two memory operations, the reuse can be realized by routing the data through either a distributed register file (assuming it is rotating), a series of routing PEs, or the local memory. Routing data through either a distributed register file or routing PEs can be wasteful, since the involved PEs cannot perform any other operation during routing. In addition, the number of wasted PEs to route data using these schemes is proportional to II, which can be large. Therefore we realize all data reuse through the local memory.

Reuse through the local memory has the benefit of lowering the local memory pressure, but at the cost of constraining the computation mapping. If computation is the bottleneck, it may be better simply not to realize data reuse. Therefore, whether and how much reuse to realize should be decided carefully for optimal results. To guide the decision we introduce DROC. DROC is defined for a memory operation and a load-store PE, and measures the goodness of a reuse opportunity which will be forfeited if the operation is mapped to the PE. Intuitively, if two load operations have a reuse relation (i.e., they load from the same array variable), placing the operations on the same row has merit (because they can access the same array variable), which is forfeited if they are placed to PEs on different rows. Notice that in the latter case the same array variable has to be duplicated to multiple banks. This reuse opportunity is what DROC tries to quantify.

1) *Data Reuse Analysis:* Data reuse analysis finds the amount of potential data reuse between every pair of memory operations. Our data reuse analysis first creates a data reuse graph (DRG) from the DFG of a loop. DRG is an undirected graph, where nodes correspond to memory operations and edge weights approximate the amount of reuse between two memory operations. The amount of reuse is maximized when two memory operations access the same array with the same index expression, or access function. We define the weight for such a case to be T, which is the number of iterations to execute between buffer switches (switching due to double buffering). If two memory access functions are affine forms that differ only in the constant (with the difference divisible by the coefficient of the affine functions), the weight is $T - d_r$, where d_r is the constant difference divided by the coefficient (i.e., reuse distance). Otherwise, we assume that there is no data reuse we can exploit, and edges with zero weight are omitted. Fig. 2(c) illustrates the DRG generated from Fig. 2(b). T is assumed to be 11 in this example. d_r between nodes 0 and 2 is 2, since the constant difference between them is 2 and their coefficient is 1. Therefore the edge weight (of the DRG) between the nodes is 9. In the same way, the edge weight between nodes 1 and 7 is 10.

Once the DRG is constructed, computing DROC is easy. Given scheduling context information such as what operations have been already scheduled and which operation is about to be scheduled, we first find the set of edges, called *frontier edge set*. For a memory operation u that is about to be scheduled, the frontier edge set of u consists of all the edges in the DRG connecting u to a memory operation v that is already scheduled. Then for such an edge $e = \{u, v\}$ in the frontier edge set, we compute its reuse opportunity as $ro_e = w_e \cdot DCR$, where w_e is the nonzero weight of edge e in the DRG, and DCR is the DCR of the loop. Finally, the reuse opportunity of each edge e induces DROC of the same amount, for all the load-store PEs other than the PE to which v is mapped. DROC induced by all reuse opportunities are averaged if the frontier edge set is larger than one. DROC is zero if the frontier edge set is empty.

2) *Example:* Consider mapping the DFG shown in Fig. 2(b) (dark nodes are memory load operations) onto the 2×2 CGRA shown in Fig. 2(a) (dark PEs are load-store PEs).

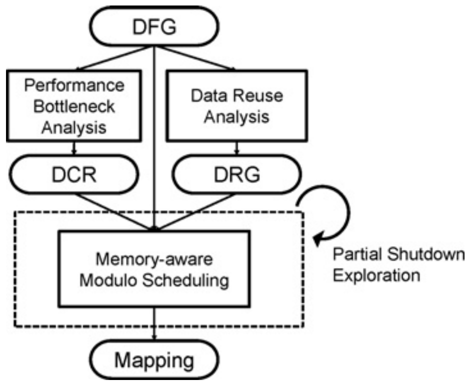

DFG
Performance Bottleneck Analysis
Data Reuse Analysis
DCR
DRG
Memory-aware Modulo Scheduling
Partial Shutdown Exploration
Mapping

Fig. 3. Application mapping flow. Note: DFG, DCR, and DRG.

The DRG for the DFG is shown in Fig. 2(c). Fig. 2(d)–(g) illustrate the mapping results in a tabular format, where the vertical direction represents time in cycles. Suppose that we are about to schedule the edge connecting operations 7 and 8 after having scheduled operations 0 through 6 as shown in Fig. 2(d). Operation 7 is a load operation $B[i+1]$, and operation 8 is an arithmetic operation.

The EMS algorithm works as follows. First the routing costs for each open PE slot where the memory operation can be scheduled are updated as in Fig. 2(d). Routing cost is calculated by multiplying the unit routing cost (which is assumed to be 10) by the number of routing PEs needed to map the edge. In this example, if we schedule operation 7 in time slot 1 of PE3, at least two routing operations are needed to map operation 8. Thus, routing cost in the time slot 1 of PE3 is 20. Considering these costs, operation 7 will be mapped onto the time slot 3 of PE1, which has the minimum cost. The final solution generated by EMS is shown in Fig. 2(e). However, this mapping requires array B to be duplicated in two banks.

DROC helps avoid duplicating reused arrays. In the same example, the DROC cost induced by the reuse relation between operations 1 and 7 is 30, assuming that the DCR parameter is 3. This DROC cost is added to all the load-store PEs except for PE3, which forces operation 7 to be scheduled onto the time slot 2 of PE3, as shown in Fig. 2(g). Though this new mapping results in the use of an extra PE as a routing PE, it increases the utilization of array B, which may reduce the overall execution time.

C. Balancing Bank Utilization

The next important issue in application mapping onto a NUMA CGRA is that, if the scheduler is not careful, it can skew the distribution of the data in the memory banks. For example, the solution can result in mapping all the data to just one bank, and not utilizing the other banks. This can happen, if the application mapping is unaware of the banked memory architecture, but also if we apply our data reuse optimization too aggressively and map all the arrays to the same bank. Such a mapping can reduce the performance, since it decreases the effective local memory size, results in smaller tiling factor for the loop, and may cause very frequent buffer switching for hardware buffering. One desirable shape of the data placement is uniform distribution of the data among the banks. This can

```
for ( $i = 0$; $i < N$; $i$++ ) do
    $Y[i+2] = a_1 Y[i+1] + a_2 Y[i] + bX[i]$
end for
```

Fig. 4. Recurrent loop, i.e., one with loop-carried data dependence.

be rather easily solved by adding an additional cost to the PEs to which load/store operations have been mapped, called *BBC*. We define the BBC for a PE p, as $BBC(p) = b \cdot m(p)$, where b is a design parameter called the base balancing cost, and $m(p)$ is the number of memory operations already mapped onto PE p.

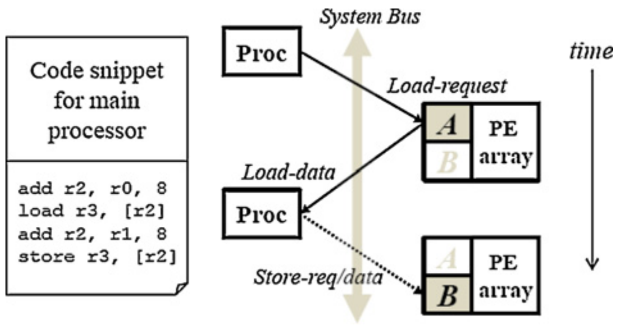
Fig. 3 illustrates our compilation flow. The two analyses, performance bottleneck analysis and data reuse analysis, are performed before time-consuming modulo scheduling. Memory-aware modulo scheduling refers to the EMS algorithm extended by adding DROC and BBC to the existing cost function, which does not significantly increase the complexity of the mapping algorithm. The partial shutdown exploration is explained in Section VII-C.

VI. Handling Recurrent Loops

A. Problem

The fundamental restrictions of CGRAs such as limited local memory size and substantial data transfer latency are the driving force behind the double-buffering architecture in the local memory. The local memory has two buffers, or two sets of banks physically, with one set connected to the PE array and the other to the main memory through the system bus. Let us call one buffer A and the other B, and if buffer A is used for computation (i.e., accessed by the PE array), buffer B is used for data transfer (i.e., accessed by DMA). What we have so far considered as the local memory is then just one set of banks that is connected to the PE array at the moment. While double-buffering can effectively ease the two restrictions of CGRA accelerators, it complicates handling of dependent data for large loops if the data cannot entirely fit in the local memory.

The situation is most complicated when there is loop-carried dependence. Fig. 4 illustrates an IIR filter, which is a recurrent loop. Suppose that the number of iterations, N, is so large that the arrays X and Y cannot be contained in one buffer of the local memory. Then the loop must be tiled; that is, the loop is executed only for some iterations (T, called *tile*) on one buffer, and then the next iterations, of up to T, are executed on the other buffer, and so on. In this scheme, it is not a problem to bring the input array X into the local memory buffers. However, managing Y is, since Y is both input and output of the computation, and therefore some elements of Y are generated in one tile and used in the next tile while the buffers available to the two tiles must be different due to double-buffering. In the above example, $Y[T+1]$ would be such a case of being generated and used in two different tiles. Thus it becomes necessary to perform some data copy operation on every buffer switch, without which the execution cannot be correct for recurrent loops. The amount of data copy is proportional to the distance between the dependent



```
add r2, r0, 8
load r3, [r2]
add r2, r1, 8
store r3, [r2]
```

Fig. 5. Data copy performed by main processor. The two `add` instructions on the left represent address calculation. The diagram on the right illustrates that the main performance bottleneck is caused by the load latency, which has to cross the system bus twice. Store instructions can be buffered, which then has little impact on the delay.

write-read reference pair. However, the runtime overhead can vary significantly depending on how the copy operation is performed.

B. Proposed Methods

We propose three methods to perform the necessary data copy operation between buffers on a buffer switch. First note that copying data between the buffers of a local memory is an extended feature of a memory-aware CGRA. By default the main processor or the PE array can access only one buffer at any time. We relax the restriction such that during buffer switching, both the buffers become accessible by either the main processor (Method 1) or the PE array (Methods 2 and 3).[3]

1) *Serial Mem-Copy by Main Processor:* The main processor repeats a pair of load-store instructions for every word of data to be copied, as illustrated in Fig. 5. Since during the buffer switching time PEs cannot do any useful work, the time it takes for the main processor to finish the memory load/store instructions is directly reflected on the overhead of this method. Most of the delay is caused by load instructions, since every memory read requires crossing the system bus twice. In contrast, store instructions, or memory write operations, can be buffered and have negligible effect on the delay. Overall, the overhead of this method can be formulated as $P + LN + Q$, where P is the number of cycles for the address calculation operations for the first load instruction,[4] L is the number of cycles taken until a pair of load-store instructions finish (from the processor's point of view), N is the total number of data words to copy, and Q is the additional number of cycles for the last store value to be written out in the local memory buffer.

This method requires that the main processor be able to access both buffers of the local memory of CGRA during buffer switching time.

[3]Otherwise the only way to do data copy between the buffers would be to transfer the data using DMA from the source buffer to the main memory, and again from the main memory to the target buffer, which would be extremely inefficient, since the amount of data to be copied is very small compared to DMA setup time.

[4]Address calculation can be quite simple, since load and store locations in the buffers are statically known.

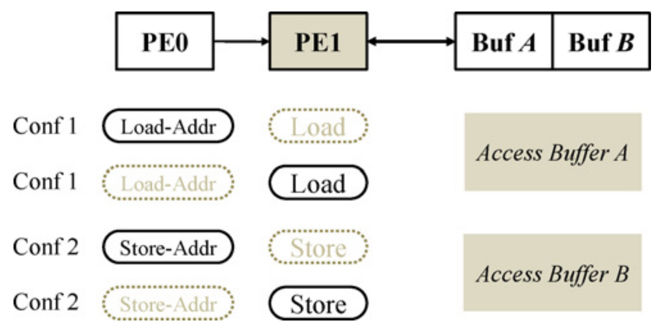


Fig. 6. Data copy performed by PEs. Here, address calculation and load/store operations are assumed to take only one cycle each.

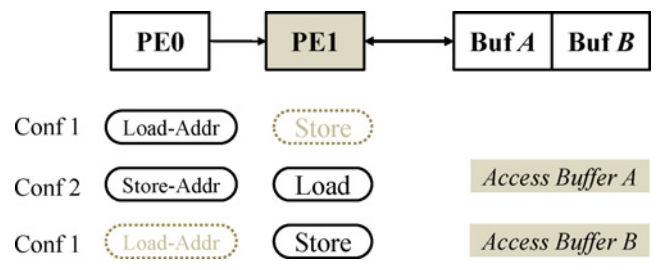


Fig. 7. Data copy performed by PEs with pipelining.

2) *Parallel Load/Store by PE Array:* The disadvantages of the first method are twofold: 1) it takes much longer for the main processor to access the CGRA local memory because the data transfer takes place across the system bus, and 2) even if the data to be copied are distributed over multiple banks, the data copy operation cannot be done in parallel by the main processor. In contrast, the PE array can access its local memory much more quickly, and multiple load-store PEs can work on different data if they are in different banks. To let the PEs do the data copy, we must provide appropriate configurations in advance, as illustrated in Fig. 6. We need separate load and store configurations, which are executed alternately, and each configuration consists of two parts, i.e., the address calculation part and the load/store part, each of which is assumed to take one cycle in Fig. 6.

This simple scheme does not utilize pipelining, and its overhead is $2(p+l)d$ cycles, where p is the height of the tree for the address calculation expression (either load or store), l is the load/store latency from the PE, and d is the largest number of data words to copy for a bank.

This method requires that the load-store PEs of a CGRA have access to both buffers of the local memory during buffer switching time.

3) *Pipelined Parallel Load/Store by PE Array:* This method has the same architectural requirement as Method 2 but improves it by pipelining the PE operations, as illustrated in Fig. 7. Pipelining can reduce the overhead to $p+2d \cdot l$ cycles. This is assuming that all the data copy operations for a bank can be pipelined as if they are from a single reference pair, which is typically the case. The difference, $(2d-1)p$, can be significant if d, the distance between the dependent reference pair, is large.

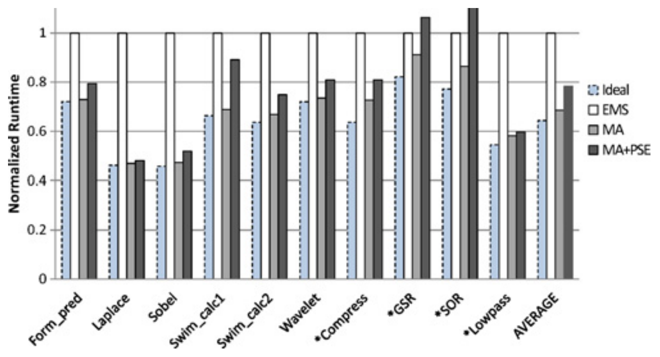


Fig. 8. Runtime comparison. Refer to Section VII-C for PSE.

TABLE I
BENCHMARK CHARACTERISTICS

	#Ns	#Es	#memNs	Length	recMII	resMII
Form_pred	10	9	4	8	–	1
Laplace	23	22	10	10	–	2
Sobel	29	34	9	10	–	2
Swim_calc1	39	44	14	8	–	3
Swim_calc2	46	48	20	8	–	3
Wavelet	20	22	6	6	–	2
Compress	9	8	5	6	3	1
GSR	12	11	6	7	4	1
SOR	25	26	12	11	8	2
Lowpass	27	26	10	11	8	2

N: operation; E: edge; memN: memory operation; Length: length of the longest path.

VII. EXPERIMENTS

A. Setup

To evaluate the effectiveness of our memory-aware compilation heuristic, we use a set of kernels from the MiBench benchmark suite [24], multimedia benchmarks, and SPEC 2000. Table I lists characteristics of benchmarks. Our target architecture is a 4×4 architecture, as illustrated in Fig. 1(b), with load-store units alternating in the two middle columns. The 4×4 configuration is the basic unit in many CGRA architectures including ADRES (4×4 tiles) and MorphoSys (4×4 quadrants), and also frequently used to evaluate various mapping algorithms (e.g., [5], [8], [9]). For the PE array, we assume that a PE is connected to its four neighbors and four diagonal ones. We also assume that each PE has an arithmetic logic unit (ALU) and a multiplier unit, so that other than memory operations, any operation can be mapped to any PE.

The local memory architecture has four banks, each connected to a different row (i.e., to the load-store unit of the corresponding row). The details of the local memory architecture, such as width, size, and latency are modeled after the RSPA architecture [18]. The local memory is double buffered in hardware and the buffers can be switched in one cycle. The size of each buffer is 768 bytes, or 384 16-bit words, and is connected to the system memory through a high-performance 16-bit pipelined bus. The system memory is assumed to operate at half the frequency of the CGRA coprocessor, giving the memory bandwidth is 16 bits per two CGRA cycles. The clock frequency of the main processor is assumed to be three times that of CGRA coprocessor.

In the literature, mapping algorithms are often compared in terms of II, which is valid, since CGRA processors are under a complete compile-time control; it is like a very long instruction word processor without pipeline stall. However, II captures the quality of the computation mapping only, and cannot capture the possible delay due to the memory bottleneck. We therefore use the CGRA runtime, which is computed by adding up tile execution times, where tile execution time is the maximum of computation II multiplied by the tile size, and the memory access time for the tile. If an array is duplicated in multiple banks with different offsets, we assume that the array is loaded twice from the system memory, which is the most straightforward way to load them, used in RSPA [18] and MorphoSys [14]. However, for read-after-write dependent references in recurrent loops we: 1) place the dependent operations on the same row, and 2) copy the dependent data on every buffer switch using Method 1 (copy by main processor) as described in Section VI.

For the energy model of the CGRA, we consider both the dynamic power and the leakage power of PEs and memory banks. For dynamic power a PE is assumed to be in one of the three power states depending on its operation: ALU operation (including load/store), multiplication, and routing. The power numbers for those states are obtained from a detailed implementation of RSPA [25] based on a 180 nm technology from DongbuAnam Semiconductor. The dynamic power of a memory bank is obtained from CACTI 5.1 [17]. Following the dynamic versus leakage power ratios published in the literature [26], we assume that a PE dissipates as leakage 20% of its dynamic power at full operation (i.e., multiplication plus ALU operation, since every PE has a multiplier), and a memory bank 20% of its read-operation dynamic power, unless the PE or the memory bank is shut down. Note that the leakage power is constant regardless of the operation; therefore, the leakage energy is proportional to runtime while the dynamic energy is not.

B. *Efficiency of Our Memory-Aware Mapping*

Though our memory-aware mapping may balance computation rate and data transfer rate, the computation rate will be maximized in the case of traditional memory-unaware mappings such as EMS. The maximum computation rate could be realized if single-bank memory were used, although it seems likely to have other negative effects such as increased cycle time, power, and area, and may cancel out the benefit. Thus, we compare three cases: *Ideal* (single-bank + EMS), *EMS* (multi-bank + EMS), and *MA* (multi-bank + our memory-aware extension of EMS). For a realistic multi-bank local memory, the *Ideal* single-bank performance only serves as the upper limit that a realistic multi-bank mapping could achieve. We compare the three cases in terms of CGRA cycle count. In the case of *Ideal*, the possible cycle time increase is not taken into account, nor is the memory bandwidth restriction (hence the name). In the case of *EMS*, the array placement is determined in a straightforward manner after computation mapping is done.

Fig. 8 compares the runtime of the three cases (in cycle count), normalized to that of *EMS*. Comparing *Ideal* and

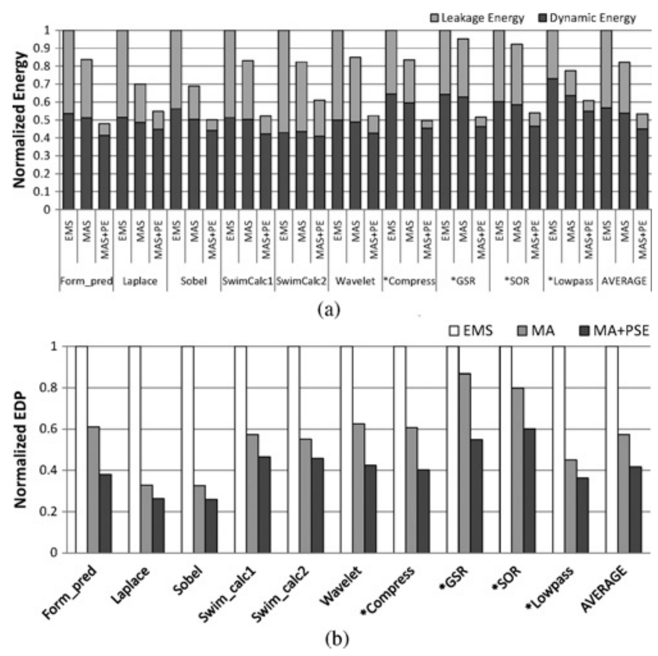


Fig. 9. Energy efficiency comparison. (a) Energy consumption. (b) EDP.

EMS indicates that for memory-bound loops, the cost of not considering array placement early in the compilation flow is quite high. By sequentially mapping computations and arrays, the runtime can increase by more than 55% on average compared to the Ideal case for memory-bound loops. On the contrary, if data mapping is considered proactively along with computation mapping as in our heuristic, the runtime increase can be very effectively suppressed. Compared to the *EMS*, our heuristic can reduce the runtime by as much as 31% on average for memory-bound loops. This strongly motivates the use of less expensive multi-bank memories for CGRAs rather than the more expensive and more power-dissipating single-bank memories.

Reduced runtime by our heuristic also translates into reduced energy consumption on the CGRA. Fig. 9(a) compares the energy consumption by the base EMS versus our heuristic, broken down into static and dynamic components. While our heuristic may sometimes generate less-efficient computation mappings compared to the base EMS (due to the increased use of routing PEs, for instance), our heuristic can effectively reduce the leakage energy by reducing the runtime, which leads to significant energy reduction by our heuristic. Accordingly, the EDP, or the energy-delay product, is also reduced significantly by our heuristic, as indicated in Fig. 9(b).

C. *Partial Shutdown Exploration*

For a memory-bound loop, the performance is often limited by the memory bandwidth rather than by computation, which will be increasingly the case as the number of PEs increases. For such a case, we can dramatically reduce the energy consumption of CGRA by shutting down some of the rows of PEs and the memory banks, effectively balancing computation and memory access. While this kind of optimization could be applied with any mapping algorithm, it becomes more interesting with our memory-aware mapping heuristic, as both our heuristic and partial shutdown try to exploit the same opportunity existing in memory-bound loops; one by reducing the memory access load, the other by reducing the computation resources. We assume that partial shutdown is performed on a per-loop basis, and if a memory bank is shut down, any output data on it are written back to the main memory before shutdown. Write-back can be done rather cheaply on a double-buffering architecture (which is our target architecture), since input/output data are loaded/stored even during loop execution. The time overhead of entering a partial shutdown mode is ignored, as it can be done while the main processor is doing some useful work.

We explore all the partial shutdown combinations on the PE rows and the memory banks, to find the best configuration that gives the minimum EDP. The design space is not large, with only 16 configurations to explore as there are four rows and four banks. The results are summarized in Figs. 8 and 9 (the last bars), with the best configurations summarized in Table II. The results suggest that the partial shutdown optimization can considerably reduce the energy consumption and the EDP, by more than 27% on average, even after our memory-aware heuristic is applied. Compared to the previous memory-unaware technique without partial shutdown optimization, our technique can achieve 59% reduction in the energy-delay product, which factors into about 47% reduction in the energy consumption and 22% reduction in the runtime. Our partial shutdown exploration gives further justification for the multi-bank memory architecture, as it is more amenable to partially shutdown than the single-bank memory architecture. And it also reinforces the importance of developing memory-aware mapping techniques for multi-bank or NUMA memory architectures, such as ours.

D. Effect of Memory Bandwidth and Register Files

To see the effect of memory bandwidth, we repeat the same experiments doubling the bandwidth between the CGRA local memory and the main memory (one word per cycle). Fig. 10(a) plots the average runtime, energy, and EDP values by different application mapping approaches, normalized to the results of EMS for the lower bandwidth case. The light gray bars are for the lower memory bandwidth and the dark ones are for the higher memory bandwidth. First, it is quite clear that the increased memory bandwidth can greatly enhance the execution profiles of the benchmarks regardless of the mapping approach. This is because the loops that we used are memory-bound rather than computation-bound, and therefore any alleviation of memory bottleneck can increase the performance. The amount of enhancement is greatest in the case of EMS, which is memory-unaware, but is significant also in the case of our memory-aware approaches. The only exception is that the energy consumption by our memory-aware mapping plus partial shutdown exploration is no further reduced by the increased bandwidth. This is because our partial shutdown exploration already balances computation and data transfer rates to minimize the energy, and therefore increasing one will only increase the other, resulting in no significant change in the overall energy consumption. The EDP reductions very closely follow the multiplication of runtime reduction and energy

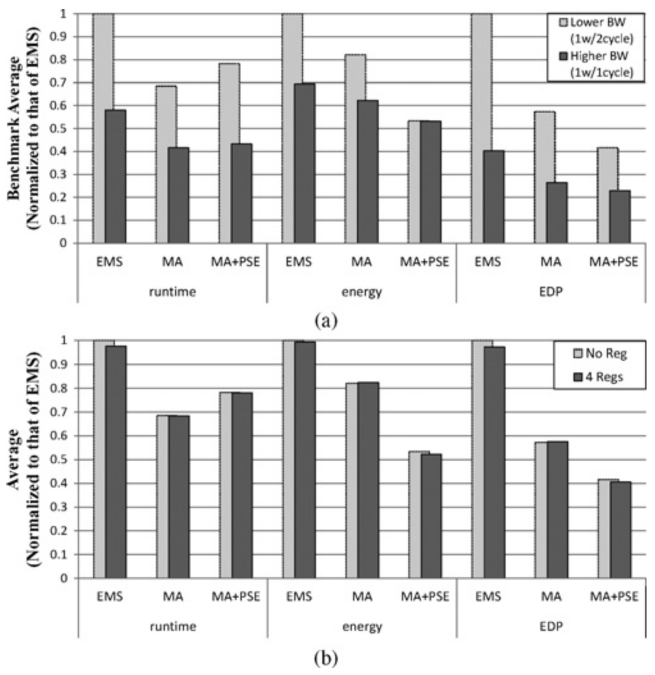


Fig. 10. Effect of memory bandwidth and local registers. (a) While a higher memory bandwidth helps all mapping approaches, it is more helpful to memory-unaware mappings. (b) For memory-bound loops, the effect of utilizing local registers is marginal.

reduction. As expected, the advantages of our memory-aware approaches over the memory-unaware one is reduced when the memory bandwidth becomes less of a bottleneck. In fact our mapping algorithm uses a measure of memory-boundness to determine how aggressively to apply our memory-related cost functions, and thus for fully computation-bound loops, our memory-aware mapping will become indistinguishable from memory-unaware mapping.

Another level of memory hierarchy in CGRA is local register files distributed throughout the PE array. The effect of distributed register files is worth investigating, since register files, being closely related to local memory in function, may reduce the necessity and effectiveness of local memory-aware optimizations such as ours. While so far our experiments have assumed no utilization of local registers for routing, in this experiment we allow up to four registers in each PE to be utilized so as to ease the data transfers among PEs. Fig. 10(b) compares, for different mapping approaches, the three execution profiles, averaged for all benchmarks and normalized to those of EMS with “no register” option. Interestingly, adding registers in each PE to aid in data routing does not translate to significant improvement by any metric. This is again due to the memory-boundness of the loops used in the experiments—if computation is not a bottleneck, improving it might not have any noticeable effect.

Table II lists the best configurations found by our partial shutdown exploration, for different combinations of memory bandwidth and register usage. As expected, the higher memory bandwidth tends to pull the best configurations toward the bigger corner. Though not as remarkable, allowing local registers has subtle effect of decreasing the number of rows of

TABLE II
BEST CONFIGURATIONS BY PARTIAL SHUTDOWN EXPLORATION (OPTIMIZED FOR ENERGY-DELAY PRODUCT "$ArBm$" REPRESENTS A ROWS AND B BANKS)

Mem BW	#Reg	Form_pred	Laplace	Sobel	Swim_calc1	Swim_calc2	Wavelet	*Compress	*GSR	*SOR	*Lowpass
1w/2cyc	0	1r1m	2r2m	3r1m	3r1m	3r2m	2r1m	1r1m	1r1m	2r1m	2r2m
1w/2cyc	4	1r1m	2r2m	2r1m	1r1m	2r2m	1r1m	1r1m	1r1m	1r1m	2r2m
1w/1cyc	0	2r2m	4r4m	4r4m	4r3m	4r3m	3r2m	2r2m	2r2m	2r2m	2r2m
1w/1cyc	4	2r2m	3r3m	4r4m	2r2m	2r2m	2r2m	2r2m	2r2m	2r2m	2r2m

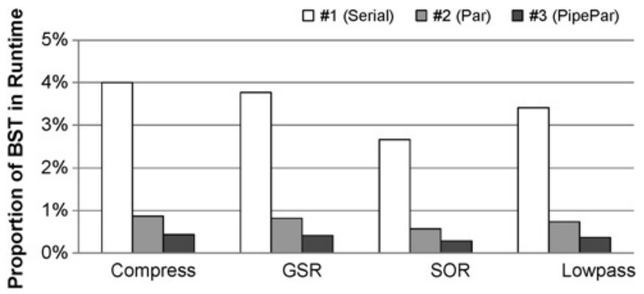


Fig. 11. Data copy overhead of the three proposed methods.

the best configurations. This means that the local registers are being used for data transfers within the PE array (thus speed up computation), but it has little effect on performance, since the loops are already memory-bound.

E. Recurrent Loop Overhead Reduction

Next, we compare the buffer switching overhead against the total runtime for all the recurrent loops used in our experiments. For this set of experiments, we use the original configuration, i.e., lower memory bandwidth and not using registers other than for storing constants. First, the exact mechanism of buffer switching for recurrent loops on a double-buffered local memory was not previously discussed in the literature. Therefore, this is the first paper quantitatively evaluating the buffer switching overhead. For architectural parameters related to data copy, we use the following: $P + Q = 9$, $L = 9$, $p = 1$, and $l = 1$, all in CGRA cycles. Here, $P+Q$ represents the initial address calculation on the main processor (six cycles) plus the last store completion (three cycles), the latter of which involves one system bus crossing and one CGRA local memory write. L accounts for two system bus crossings with synchronization, one CGRA local memory read, and two load/store instructions on the main processor.

The results shown in Fig. 11 suggest that the buffer switching itself, though very important for correct execution of these loops, does not have particularly impact on runtime, ranging from 2.7% to 4%, even by our simplest method of using the main processor as the agent. Second, our more advanced schemes of using the PE array for the data copy can bring down the buffer switching overhead to a minimum—to less than 1% for all the cases. This is because the advanced schemes copy the dependent data directly between buffers without ever leaving the local memory. The fact that our second method performs well enough (compared to the third) is very assuring, since the second method requires a much slower rate of changing PE-to-buffer connections—in our experiments, once in two cycles for Method 2 versus once per every cycle for Method 3—and because a slower rate requirement means that the corresponding hardware could be implemented more efficiently, i.e., with lower cost and/or power.

VIII. CONCLUSION

The promise of CGRAs providing very high power efficiency while being software programmable, critically hinges on the effectiveness of application mapping. While previous solutions have focused on improving the computation speed of the PE array, we motivate the need to balance computation rate and data transfer rate to achieve higher performance and energy efficiency for memory-bound loops on CGRAs. We presented an effective heuristic that can be easily integrated with existing modular scheduling-based algorithms. Further, we presented efficient methods to handle dependent data on a double-buffering local memory, which is necessary for recurrent loops. Our experimental results on memory-bound loops from MiBench, multimedia, and SPEC benchmarks demonstrate that not only is our proposed heuristic able to achieve near-optimal results as compared to single-bank memory mapping but it can also achieve 59% reduction in the energy-delay product as compared to memory-unaware mapping for multi-bank memory, which factors into 47% and 22% reductions in the energy consumption and runtime, respectively. Further, our extensive experiments showed that our scheme scales across a range of applications and memory parameters, and the runtime overhead of handling recurrent loops can be less than 1% with our proposed methods.

REFERENCES

[1] J. Bormans. (2006, Oct.–Nov.). Reconfigurable array processor satisfies multi-core platforms. *Chip Des. Mag.* [Online]. Available: http://chipdesignmag.com/display.php?articleId=950&issueId=19

[2] J. Lee, K. Choi, and N. Dutt, "Compilation approach for coarse-grained reconfigurable architectures," *IEEE Des. Test Comput.*, vol. 20, no. 1, pp. 26–33, Jan.–Feb. 2003.

[3] J. Lee, K. Choi, and N. Dutt, "An algorithm for mapping loops onto coarse-grained reconfigurable architectures," *ACM SIGPLAN Notices*, vol. 38, no. 7, pp. 183–188, 2003.

[4] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "DRESC: A retargetable compiler for coarse-grained reconfigurable architectures," in *Proc. IEEE Int. Conf. FPT*, Dec. 2002, pp. 166–173.

[5] H. Park, K. Fan, M. Kudlur, and S. Mahlke, "Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures," in *Proc. CASES*, 2006, pp. 136–146.

[6] A. Hatanaka and N. Bagherzadeh, "A modulo scheduling algorithm for a coarse-grain reconfigurable array template," in *Proc. IPDPS*, Mar. 2007, pp. 1–8.

[7] M. Ahn, J. Yoon, Y. Paek, Y. Kim, M. Kiemb, and K. Choi, "A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures," in *Proc. DATE*, 2006, pp. 363–368.

[8] J. 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 *Proc. ASP-DAC*, 2008, pp. 776–782.

[9] H. Park, K. Fan, S. Mahlke, T. Oh, H. Kim, and H. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proc. PACT*, 2008, pp. 166–176.

[10] 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 *Proc. CASES*, 2001, pp. 116–125.

[11] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe, "Space-time scheduling of instruction-level parallelism on a raw machine," in *Proc. ASPLOS-VIII*, 1998, pp. 46–57.

[12] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proc. 27th Annu. Int. Symp. Microarch. MICRO*, 1994, pp. 63–74.

[13] 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*, vol. 28, no. 4, pp. 41–50, Jul.–Aug. 2008.

[14] H. Singh, M.-H. Lee, G. Lu, N. Bagherzadeh, F. Kurdahi, and E. Filho, "MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Trans. Comput.*, vol. 49, no. 5, pp. 465–481, May 2000.

[15] R. Hartenstein, "A decade of reconfigurable computing: A visionary retrospective," in *Proc. DATE*, 2001, pp. 642–649.

[16] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins, "Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: A case study," in *Proc. DATE*, Feb. 2004, pp. 1224–1229.

[17] S. Thoziyoor, N. Muralimanohar, J. Ahn, and N. Jouppi, "Cacti 5.1," HP Lab., Palo Alto, CA, Tech. Rep. HPL-2008-20, 2008.

[18] 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 *Proc. DATE*, 2005, pp. 12–17.

[19] K. Wu, A. Kanstein, J. Madsen, and M. Berekovic, "MT-ADRES: Multi-threading on coarse-grained reconfigurable architecture," *Int. J. Electron.*, vol. 95, no. 7, pp. 761–776, 2008.

[20] G. Dimitroulakos, M. Galanis, and C. Goutis, "Alleviating the data memory bandwidth bottleneck in coarse-grained reconfigurable arrays," in *Proc. ASAP*, 2005, pp. 161–168.

[21] J. Lee, K. Choi, and N. Dutt, "Evaluating memory architectures for media applications on coarse-grained reconfigurable architectures," in *Proc. ASAP*, 2003, pp. 172–182.

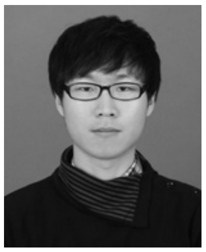
[22] G. Dimitroulakos, S. Georgiopoulos, M. Galanis, and C. Goutis, "Resource aware mapping on coarse grained reconfigurable arrays," *Microprocess. Microsyst.*, vol. 33, no. 2, pp. 91–105, 2009.

[23] Y. Kim, J. Lee, A. Shrivastava, J. W. Yoon, and Y. Paek, "Memory-aware application mapping on coarse-grained reconfigurable arrays," in *Proc. HiPEAC*, 2010, pp. 171–185.

[24] M. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. IWWC*, 2001, pp. 3–14.

[25] I. Park, Y. Kim, C. Park, J. Son, M. Jo, and K. Choi, "Chip implementation of a coarse-grained reconfigurable architecture," in *Proc. ISOCC*, 2006, pp. 628–629.

[26] L. Schuth and J. Binney, "Managing power in 45nm and 65nm designs," ARM and Synopsys, Mountain View, CA, Presentation Material [Online]. Available: http://www.dianzichan.com/anonymous/ic/arm07conf_mgm_pwr45nm.pdf



Yongjoo Kim received the B.S. degree in electrical engineering from Seoul National University, Seoul, Korea, in 2006, where he is currently pursuing the Ph.D. degree with the School of Electrical Engineering and Computer Science.

He was a Visiting Researcher with Compiler Micro-Architecture Laboratory of Arizona State University, Phoenix, in 2009. His current research interests include optimization techniques for coarse-grained reconfigurable architecture, embedded system, and multi-core and many-core system-on-chip.

Jongeun Lee (S'01–M'11) received the B.S. and M.S. degrees in electrical engineering, and the Ph.D. degree in electrical engineering and computer science, all from Seoul National University, Seoul, Korea, in 1997, 1999, and 2004, respectively.

He is currently an Assistant Professor with the School of Electrical and Computer Engineering, Ulsan National Institute of Science and Technology, Ulsan, Korea. His current research interests include energy-efficient computing, such as reconfigurable architectures, compilers for low power and reliability, and design and optimization of multi-core systems-on-chip.

Aviral Shrivastava (M'09) received the Bachelors degree in computer science and engineering from the Indian Institute of Technology, Delhi, and the Masters and Ph.D. degree in information and computer science from the University of California, Irvine.

He is currently an Assistant Professor with the Department of Computer Science and Engineering, Arizona State University, Phoenix, where he has established and heads the Compiler Micro-Architecture Laboratories. His current research interests include intersection of compilers and computer architecture, ranging from embedded systems to high performance computing.

Dr. Shrivastava was the recipient of the prestigious 2010 NSF CAREER Award. His research is funded by NSF and several industries including Intel, Nvidia, Microsoft, Raytheon Missile Systems, and others. He serves on the organizing and program committees of several premier embedded system conferences, including ISLPED, CODES+ISSS, CASES and LCTES, and NSF and DOE review panels.

Jonghee W. Yoon received the B.S. degree in electrical engineering from the Korea Advanced Institute of Science and Technology, Daejeon, Korea, in 2005. Currently, he is pursuing the Ph.D. degree with the School of Electrical Engineering and Computer Science, Seoul National University, Seoul, Korea.

He was a Visiting Researcher with the Compiler Micro-Architecture Laboratory, Arizona State University, Phoenix, in 2007. His current research interests include embedded software, reconfigurable array processors, and MPSoC.

Mr. Yoon received a Best Paper Award at WASP 2007.

Doosan Cho (M'06) received the B.S. degree in digital information engineering from the Hankuk University of Foreign Studies, Seoul, Korea, in 2001, and the Ph.D. degree in electrical engineering and computer science from Seoul National University, Seoul, in 2009.

He was the Director with a semiconductor company for three years. He is currently an Assistant Professor with the Department of Electronic Engineering, Sunchon National University, Suncheon, Korea. He has published more than 20 research articles in peer-reviewed conferences and journals and presented several tutorials in the area of embedded system design and optimization at leading conferences. His current research interests include intersection of compilers and computer architecture, with particular interest in embedded systems, multiprocessor system optimization, and compiler techniques for low power and high speed memory systems.

Yunheung Paek (M'99) received the B.S. and M.S. degrees in computer engineering from Seoul National University (SNU), Seoul, Korea, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign, Urbana.

In 2003, he joined the School of Electrical Engineering, SNU, as an Associate Professor. Before he joined SNU, he was with the Department of Electrical Engineering, Korea Advanced Institute of Science and Technology, Daejeon, Korea, as an Associate Professor for one year and an Assistant Professor for two and a half years. His current research interests include embedded software, embedded system development tools, retargetable compilers, and multiprocessor systems-on-chip.

Dr. Paek has served on the program committees and organizing committees for numerous conferences. He was an Associate Editor or Reviewer for various journals and transactions.