Mapping Multi-Domain Applications onto Coarse-Grained Reconfigurable Architectures

Ganghee Lee, Kiyoung Choi, *Senior Member, IEEE,* and Nikil D. Dutt, *Fellow, IEEE*

***Abstract*—Coarse-grained reconfigurable architectures (CGRAs) have drawn increasing attention due to their performance and flexibility. However, their applications have been restricted to domains based on integer arithmetic since typical CGRAs support only integer arithmetic or logical operations. This paper introduces approaches to mapping applications onto CGRAs supporting both integer and floating-point arithmetic. After presenting an optimal formulation using integer linear programming, we present a fast heuristic mapping algorithm. Our experiments on randomly generated examples generate optimal mapping results using our heuristic algorithm for 97% of the examples within a few seconds. We observe similar results for practical examples from multimedia and 3-D graphics benchmarks. The applications mapped on a CGRA show up to 120 times performance improvement compared to software implementations, demonstrating the potential for application acceleration on CGRAs supporting floating-point operations.**

***Index Terms*—Design automation, high-level synthesis, parallelizing compiler, reconfigurable architecture.**

I. Introduction

WITH THE INCREASING requirements for more flexibility and higher performance in embedded systems design, reconfigurable computing is becoming more and more popular. Various coarse-grained reconfigurable architectures (CGRAs) have been proposed in recent years [1]–[3], with different target domains of applications and different tradeoffs between flexibility and performance. Typically, they consist of a reconfigurable array of processing elements (PEs) and a host [mostly reduced instruction set computing (RISC)] processor. The computation intensive kernels of the applications—typically loops—are mapped to the reconfigurable array while the remaining code is executed by the processor. However, it is not easy to map an application onto the reconfigurable array because of the high complexity of the problem that requires compiling the application on a dynamically reconfigurable parallel architecture, with additional complexity of dealing with complex routing resources. The problem of mapping an application onto a CGRA to minimize the number of resources giving best performance has been shown to be NP-complete [16].

Few automatic mapping/compiling/synthesis tools have been developed to exploit the parallelism found in the applications and extensive computation resources of CGRAs. Some researchers [1], [4] have used structure-based or graphical user interface-based design tools to manually generate a mapping, which would have a difficulty in handling big designs. Some researchers [5], [6] have only focused on instruction-level parallelism, failing to fully utilize the resources in CGRAs, which is possible by exploiting loop-level parallelism. Some researchers [7], [8], [17] have introduced a compiler to exploit the parallelism in the CGRA provided by the abundant resources. However, their approaches use shared registers to solve the mapping problem. While these shared registers simplify the mapping process, they can increase the critical path delay or latency. We show in this paper that the shared registers can be eliminated if routing resources are considered explicitly during the mapping process.

More recently, routing-aware mapping algorithms have been introduced [9]–[11]. However, they rely on step-by-step approaches, where scheduling, binding, and routing algorithms are performed sequentially. Thus, it tends to fall into a local optimum. Our unified approach considers scheduling, binding, and routing at the same time, thereby generating better optimized solutions. It also explicitly considers incorporating Steiner points for more efficient routing (some previous approaches use a kind of maze routing algorithm that can incorporate Steiner points although it is not clear whether they are actually incorporated). In [7], they have also presented a unified approach based on the simulated annealing algorithm. However, it takes too much time to get a solution. Typically, the approach in [7] takes hundreds of minutes whereas our approach takes only a few minutes. Furthermore, all previous mapping/compiling/synthesis tools have been restricted to integer-type application domains, whereas ours extends the coverage to floating-point-type application domains.

As floating-point applications such as 3-D graphics become more prevalent, acceleration of floating-point operations become more important. In [22], they have introduced floating-point behavioral synthesis to provide an optimized floating-point functional library giving tradeoffs between performance

Manuscript received April 22, 2010; revised August 12, 2010 and November 19, 2010; accepted November 27, 2010. Date of current version April 20, 2011. This work was supported by the Korea Science and Engineering Foundation, under NRL Program Grant R0A-2008-000-20126-0 funded by the Ministry of Education Science and Technology, Korea. A preliminary version of this paper was presented at the 6th International Symposium on Applied Reconfigurable Computing [20]. This paper was recommended by Associate Editor Y. Xie.

G. Lee is with the SoC Platform Development Team, System-LSI Division, Samsung Electronics Company, Gyeonggi 445-701, Korea (e-mail: ghi.lee@samsung.com).

K. Choi is with the School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-600, Korea (e-mail: kchoi@snu.ac.kr).

N. D. Dutt is with the Departments of Computer Science and Electrical Engineering, University of California, Irvine, CA 92697 USA (e-mail: dutt@uci.edu).

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.2010.2098571

and area. For the reconfigurable architectures, most of the previous researchers [23], [24] have focused on fine-grained architectures such as field programmable gate array. Thus, little work has examined the problem of mapping floating-point applications onto CGRAs. Currently, there is no existing compiler known to support floating-point operations for CGRAs [25].

This paper presents two mapping approaches: 1) an optimal approach using integer linear programming (ILP), and 2) a fast heuristic approach using quantum-inspired evolutionary algorithm (QEA). Both approaches support integer-type applications as well as floating-point-type applications. Our mapping algorithms adopt high-level synthesis (HLS) techniques that handle loop-level parallelism by applying loop unrolling and pipelining techniques.

The remainder of this paper is organized as follows. Section II introduces our CGRA supporting both integer-type application domains and floating-point-type application domains. Section III explains overall design flow for application mapping onto the CGRA. Section IV formulates the application mapping problem with ILP to find the optimal solution. Section V introduces a fast heuristic algorithm for application mapping based on HLS techniques. Section VI demonstrates the effectiveness of our approach. Finally, Section VII concludes this paper.

II. Target Architecture

A. *Coarse-Grained Reconfigurable Architecture*

Our target architecture consists of a reconfigurable computing module (RCM) for executing loop kernel code segments and a general-purpose processor for controlling the RCM, and these units are connected with a shared bus. The RCM used in our platform consists of an array of PEs, several sets of data memory, and a configuration cache memory [13]. Fig. 1 shows our CGRA containing a 4 × 4 reconfigurable array of PEs and internal structure of the PEs. It is connected with the nearest neighboring PEs—top, bottom, left, and right. The size of the array can be optimized to a specific application domain [13]. In Fig. 1, the area-critical functional units (such as multipliers) are located outside the PEs and shared among a set of PEs [13]. Each area-critical functional unit is pipelined to curtail the critical path delay, and its execution is initiated by scheduling the area-critical operation on one of the PEs that share this area-critical resource. Thus, each PE can be dynamically reconfigured either to perform arithmetic and logical operations with its own arithmetic logic unit (ALU) in one clock cycle, or to perform multiply or division operations using the shared functional unit in several clock cycles with pipelining. Resource pipelining further improves loop pipelining execution by allowing multiple operations to execute simultaneously on one pipelined resource. Furthermore, pipelining together with resource sharing further increases the utilization of these area-critical units.

The data memory in Fig. 1 is used for storing data that can be accessed by the PEs. There are two sets of memory, each of which consists of three banks: one connected to the write bus and the other two connected to the read buses. These read/write buses are also shared by the PEs like shared functional units. These two sets of memory are used for double buffering.

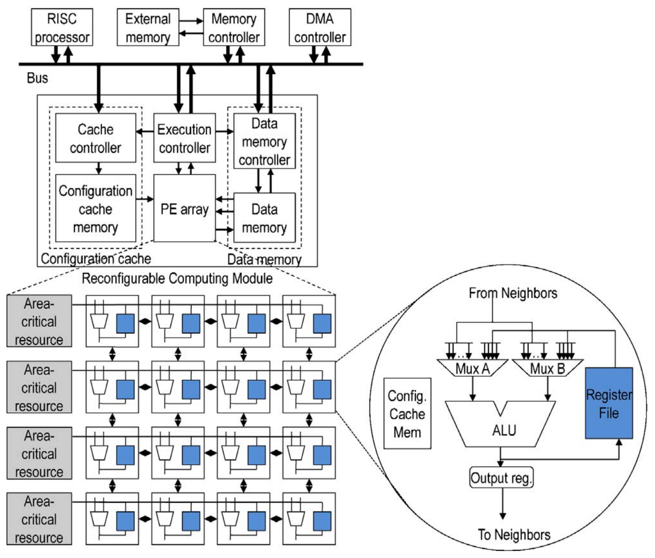


Fig. 1. CGRA containing a 4 × 4 reconfigurable array of PEs.

The configuration cache consists of cache elements (CEs) in the form of an array of the same size as the array of PEs, i.e., it has an M (number of PEs in a column) by N (number of PEs in a row) array of CEs. Each CE has several layers, so the corresponding PE can be reconfigured independently with different contexts. Note that the area-critical resources shared by the PEs in the same row are activated through the individual PEs and, thus, need not be explicitly considered for the modeling of the CEs.

B. *Architecture Extension for Floating-Point Operations*

A pair of PEs in the PE array is able to compute floating-point operations according to its configuration [21]. While a PE in the pair computes the mantissa part of the floating point operations, the other PE handles the exponent part. Since the data path of a PE is 16 bits wide, the floating-point operations do not support the single precision IEEE-754 standard, but support only reduced mantissa of 15 bits. However, experiments show that the precision is good enough for hand-held embedded systems [21].

Since adjacent PEs are paired for floating-point operations [Fig. 2(c)], the total number of floating-point units that can be constructed is half the number of integer PEs in the PE array as shown in Fig. 2(a) and (b). The PE array has enough interconnections among the PEs so that it makes use of the interconnections for the data exchange between the PEs required in floating-point operations.

Mapping a floating-point operation on the PE array with integer operations may take many layers of cache. If a kernel consists of a multitude of floating-point operations, then mapping it on the array easily runs out of the cache layers, causing costly fetch of additional context words from the main memory. Instead of using multiple cache layers to perform such a complex operation, we add some control logic to the PEs so that the operation can be completed in multiple cycles but without requiring multiple cache layers. The control logic

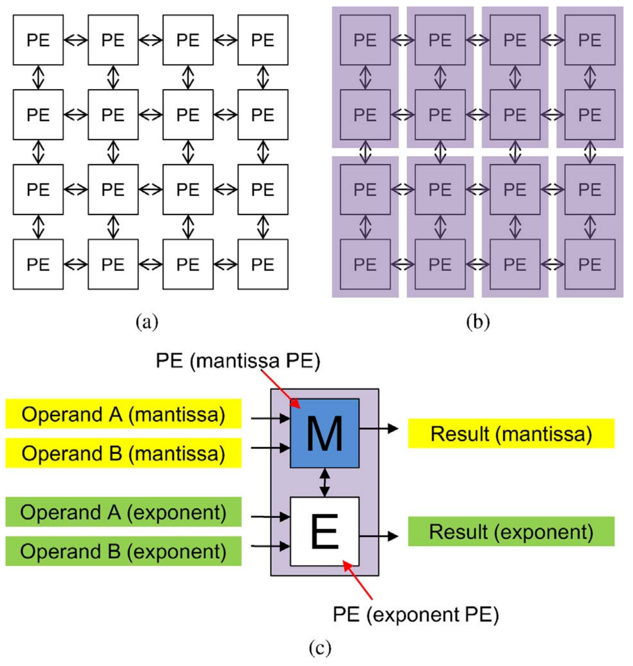


Fig. 2. Diagram of 4×4 PE array: combination of a pair of PEs for floating-point operation in the same integer PE array. (a) Integer PE array (4×4). (b) Floating-point PE array (2×4). (c) Pair of integer PEs executes floating-point operation.

can be implemented with a small finite state machine (FSM) that controls the PE's existing datapath for a fixed number of cycles.

C. Mapping Strategies for CGRA

When mapping kernels onto the reconfigurable architecture, we can consider two different strategies: spatial mapping and temporal mapping. Fig. 3 shows the difference between the two mapping strategies. In case of spatial mapping [Fig. 3(a)], each PE executes a fixed operation with a static configuration. Thus, each operation in a loop body is spatially mapped to a dedicated PE. The main advantage of spatial mapping is that each PE may not need reconfiguration during execution of a loop because of its fixed functionality. However, it has a disadvantage that spreading all operations of the loop body over the reconfigurable array may require a very large array size. Moreover, the limited interconnect resources among the PEs should be allocated to the data dependencies with great care. Otherwise, the deficit in the interconnect resources will decrease the utilization of the PEs, thus requiring even bigger array size.

In case of temporal mapping, a PE executes multiple different operations by changing the configuration dynamically within a loop. Therefore, complex loops having many operations with heavy data dependencies can be mapped better in temporal fashion, provided that the configuration cache has sufficient number of layers to execute the whole loop body.

Fig. 3(b) shows an example of temporal mapping. In the first cycle, the first column executes the first configuration (*Load*). In the second cycle, the second column fetches the same configuration (*Load*) while the first column fetches the second configuration (*Exec1*). Four cycles later, when the pipeline is

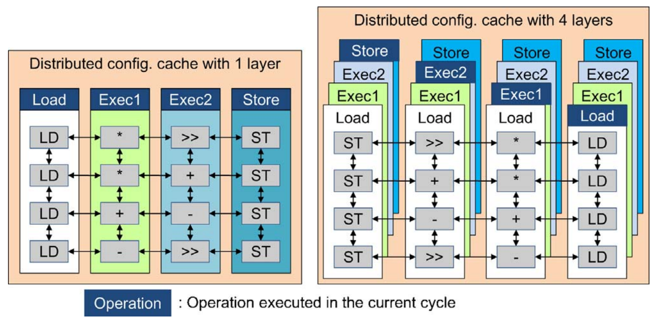

Distributed config. cache with 1 layer
Load
Exec1
Exec2
Store
LD
*
>>
ST
+
-
Distributed config. cache with 4 layers
Operation
: Operation executed in the current cycle

Fig. 3. Two different mapping strategies. (a) Spatial mapping. (b) Temporal mapping.

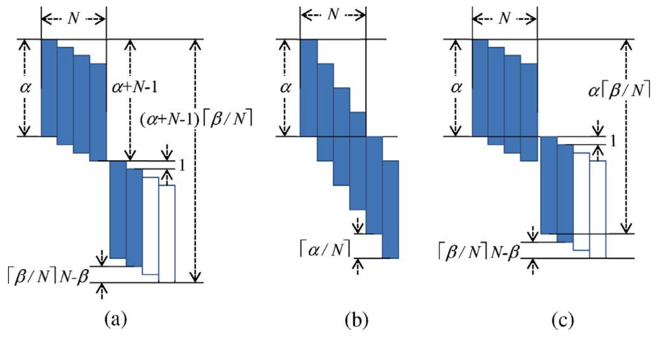


Fig. 4. Scheduling of loop iterations. (a) S1. (b) S2. (c) S3.

full, the configuration of the array will look like the one in Fig. 3(b). Assuming that there is no loop-carried dependency or the initiation interval is one, the total latency of the loop execution is determined by

$$T = \alpha + (\beta - 1) \tag{1}$$

where N is the number of columns in the array, α is the latency of one loop iteration obtained by scheduling, and β is the number of iterations. This is true when there are enough columns (i.e., $N \geq \beta$) so that each iteration can be executed on a dedicated column.

When $N < \beta$, we can implement the loop on the array by executing N iterations at a time (S1). Thus, we need to repeat the execution $\lceil \beta/N \rceil$ times to complete the entire loop iterations. The first column waits for the completion of the last column and then repeats execution. Fig. 4(a) shows how each iteration is scheduled for execution. In this case, the total latency becomes

$$T = (\alpha + N - 1) \cdot \lceil \beta/N \rceil - (\lceil \beta/N \rceil N - \beta) \tag{2}$$

where the second term takes care of possible early termination of the last stage.

Alternatively, we can implement the loop on the array by extending the initiation interval to $\lceil \alpha/N \rceil$ (S2). Again, we need to repeat the execution $\lceil \beta/N \rceil$ times to complete the entire loop iterations. However, the first column does not wait for the completion of the last column. Fig. 4(b) shows how each iteration is scheduled for execution. In this case, the total latency becomes

$$T = \alpha + (\beta - 1) \cdot \lceil \alpha/N \rceil . \tag{3}$$

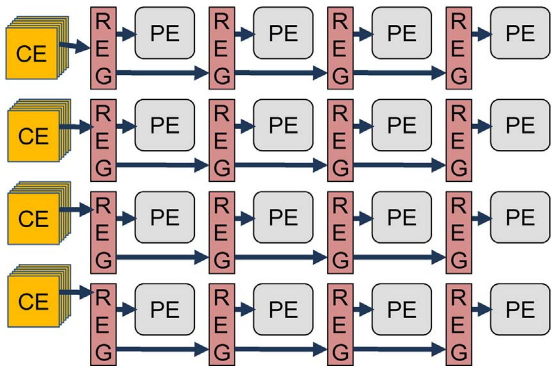


Fig. 5. Configuration cache structure for temporal mapping.

We can implement the loop more aggressively as shown in Fig. 4(c) (S3). In this case, the total latency becomes

$$T = \alpha \cdot \lceil \beta/N \rceil + (N - 1) - (\lceil \beta/N \rceil N - \beta). \tag{4}$$

In terms of latency, S3 gives the best result. One can easily show that S1 is better than S2, if $\alpha > \beta$, and S2 is better, otherwise. We have taken the approach of S1 in this paper for simplicity in implementation, but S2 or S3 can also be used.

When there is a loop-carried dependency, we have to resolve it under resource constraints (e.g., number of fetch units and interconnect topology between columns) and it may increase the initiation interval. Once the initiation interval (*II*) is determined, the total latency given by (1) and (2) are modified, respectively, to

$$\begin{aligned} T &= \alpha + II \cdot (\beta - 1), \text{ and} \\ T &= (\alpha + II \cdot (N - 1)) \cdot \lceil \beta/N \rceil - II \cdot (\lceil \beta/N \rceil - \beta). \end{aligned} \tag{5}$$

In the CGRA, since the PEs are connected with the nearest neighbors, the data for the loop-carried dependency can be transferred through the interconnections between columns. The details are explained in Section IV.

Even though temporal mapping is more efficient in mapping complex loops onto the reconfigurable array, it requires many configuration cache layers as well as many power-consuming accesses to the configuration cache. To reduce the area cost and power consumption, we have only one column of configuration cache for temporal mapping, which consists of M (the number of PEs in a column) by H (the number of layers) array of CEs as shown in Fig. 5. Once the configuration is fetched from this configuration cache, the context word is handed over from the left to the right neighbor to achieve loop-level pipelining [14].[1]

III. Design Flow

A. Overall Design Flow

Fig. 6 describes the overall design flow, which integrates the processes of mapping of kernels onto the reconfigurable array.

[1]To make this scheme work for pipelining loops with initiation interval greater than one, we employ adjustable-depth FIFO buffers for pipelining the configuration at the same rate. However, if the initiation interval is too large, the cache for spatial mapping can be diverted to temporal mapping, although it is less efficient.

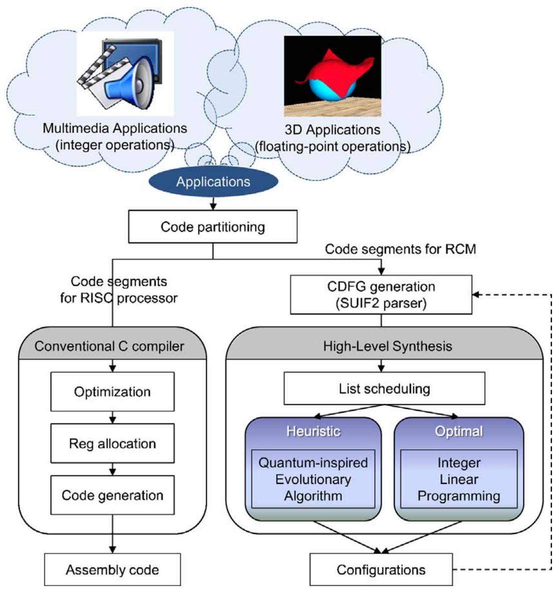


Fig. 6. Overall design flow for application mapping onto CGRA.

With our hardware/software partitioning tool [12], we first partition the application into two parts that execute, respectively, on the RISC processor and the RCM. Unlike other application mapping approaches for CGRAs, ours is not restricted to integer-type application domains, but also covers applications such as 3-D graphics that require intensive floating-point operations. The result of partitioning is two sets of code segments written in C. For the code segments for the RISC processor, we statically schedule them and generate assembly code with a conventional compiler. For the code segments for the RCM (generally loop kernels), we generate a control data flow graph (CDFG) using the SUIF2 [15] parser. During this process, we perform loop unrolling to maximize the utilization of the PEs. We then perform HLS to get the schedule and binding information for one column of PEs. Since we have multiple columns, we let each column of the CGRA execute its own iteration of the loop to implement loop pipelining. In other words, our approach adopts the temporal mapping strategy.

B. Loop Unrolling

Since we have many PEs in a column of the CGRA, there are chances that we can benefit from unrolling the loop. To get a proper unroll factor, we first run as soon as possible (ASAP) scheduling over one iteration of the loop without resource constraints. If the maximum resource usage is lower than the given resource constraint (the number of PEs in a column), we increase the unroll factor, which is repeated while it meets the resource constraint. With this initial value of unroll factor, we unroll the loop and run the mapping algorithm to obtain the total latency of the entire loop execution. Then we try to increase the unroll factor further to see if we can find a better solution. This iterative improvement continues until the increased unroll factor does not help reduce the total latency.

Fig. 7(a) shows the CDFG representation of a simple loop body ($Z = A * B + C * D$) and Fig. 7(b) shows the loop-unrolled CDFG representation. Assume that multiplication takes two clock cycles and addition takes one clock cycle.

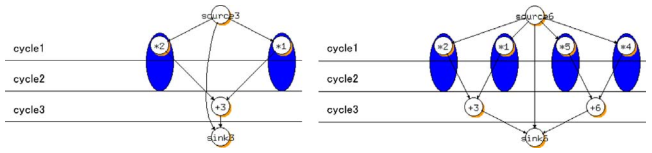


Fig. 7. CDFG view and loop unrolling with unroll factor 2. (a) CDFG example. (b) Loop unrolling.

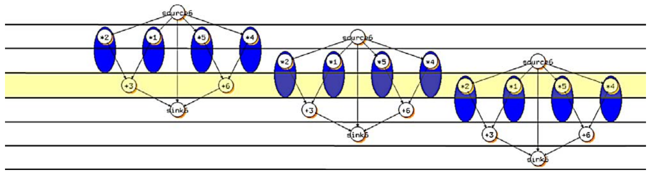


Fig. 8. CDFG view after loop pipelining with initiation interval 1.

Also assume that each column has four PEs. Each PE can perform ALU operations, load/store operations, or multiplication/division operations. Each row has one multiplier, one divider, and one load/store unit shared among the four PEs in the row.

C. HLS with Loop Pipelining

With multiple columns in the CGRA, we perform loop pipelining so that each column can undertake one iteration of the loop. Fig. 8 shows the resulting schedule obtained by our approach on the simple example from Fig. 7. With loop unrolling and pipelining, the total latency of whole loop kernel is reduced significantly. Assuming that the number of iterations of this loop kernel is 6, in this example, the total latency is reduced from $4 * 6 = 24$ to $4 + 2 = 6$ cycles.

We perform HLS to map the unrolled CDFG onto the RCM considering loop pipelining. With a given resource constraint (e.g., number of adders, multipliers, or buses), a typical HLS produces schedule and binding information for each operation. The novelty of our mapping algorithm is that it performs HLS with the number of PEs in each column as the resource constraint and simultaneously deals with the routing problem by explicitly representing the routing resources (which is not considered in typical HLS). Furthermore, we consider deploying Steiner trees for solving the routing problem, since a Steiner tree may give a better result than a Spanning tree in terms of performance and power [20].

As shown in Fig. 6, our mapping algorithm has two paths: 1) optimal but slow path based on ILP, and 2) fast heuristic path based on QEA. Both paths start from the list scheduling result, since it reduces the time taken to obtain a solution.

D. Mapping Operations

In our mapping flow, we map floating-point operations as well as integer operations. Fig. 9 shows an example of mapping a kernel of 3-D physics engine consisting mostly of floating-point operations. Unlike normal operations, floating-point operations are multicycle operations executed on a pair of integer PEs.

Normally, each PE fetches a new context word from the configuration cache every cycle to execute the operation corresponding to the context word. However, if the fetched context

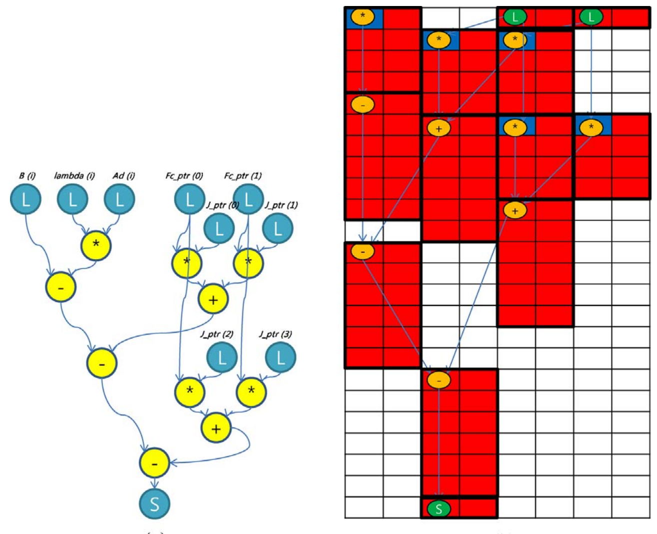


Fig. 9. (a) Kernel part of 3-D physics engine. (b) Mapping of the floating-point operations onto CGRA. A floating-point operation is treated as a multicycle operation which uses a pair of integer PEs.

word is for a multicycle operation such as floating-point operation, the control is passed over to the FSM. A context word for such multicycle operation contains information about how long the operation will be executed, so that the cache control unit can wait until the current multicycle operation is finished. During the multicycle operation, the cache control unit does not send the next context words to the PEs but resumes sending the context words right after the multicycle operation is finished. The operations that a PE (or a pair of PEs) in our CGRA can execute are classified into the following three groups.

1) *Arithmetic/logical operations:* a PE can execute ALU operations in one clock cycle.
2) *Multiply/divide/load/store operations:* a PE can execute multiply/divide/load/store operations in several clock cycles. These operations are executed by dedicated functional resources located outside the PE array. Since the functional resources are pipelined, they can start a new computation every clock cycle.
3) *Floating-point operations:* a pair of PEs can execute floating-point operations taking several clock cycles. These operations are also multicycle operations like multiply/divide/load/store operations. However, they cannot be pipelined since they are executed directly on the PEs (which have no pipelining). Among the floating-point operations, however, some operations such as floating-point multiplication and division utilize the dedicated outside integer multiplier or divider. Both operands of a floating-point operation must be of floating-point type since we do not support mixed-type inputs or type-casting in our current CGRA implementation.

IV. Problem Formulation

A. Notations and Definitions

The main objective of the problem is to map a given loop kernel to the CGRA such that the total latency is minimized while satisfying several constraints. In this section, we define the notations that are used throughout this paper, and formulate our problem using ILP.

1) *Loop Kernel:* The loop kernel is represented by a CDFG, $K = (V, E, D)$, where V is the set of vertices, E is the set of data-dependency edges, and D is the set of loop-carried dependency edges. The vertices in V represent the operations in the loop, and an edge $e = (u, v) \in E$ exists for a pair of vertices, $u, v \in V$, iff operation v is data-dependent on operation u, and an edge $d = (u, v) \in D$ exists for a pair of vertices, $u, v \in V$, iff operation v is loop-carried dependent on operation u.

2) *CGRA:* A column of the CGRA which has M PEs with H configuration cache layers can be represented by a graph $C = (P, L)$, where P is the set of vertices and L is the set of edges. For a given column of the CGRA, vertex $p_{hm} \in P, 1 \leq h \leq H, 1 \leq m \leq M$ represents the mth PE that is configured by the hth layer of the configuration cache. For any two elements $p, q \in P$, an interconnection link $l = (p, q) \in L$ exists, if there is an interconnection between PEs p and q.

3) *Application Mapping:* The application mapping problem is formulated as follows. Given a kernel graph $K = (V, E, D)$ and a CGRA graph $C = (P, L)$, find a mapping of K onto C with the objective of minimizing total latency under resource constraints. Note that the problem is not just finding a subgraph of C that is isomorphic to K, since we should also consider the case where a PE sends the output data to the destination indirectly using other PEs as routing resources.

B. ILP Formulation

The problem of application mapping onto a CGRA has been proven to be NP-complete [16], even in the special case where the application is represented by a complete binary tree and the CGRA consists of a 2-D grid with just the neighboring connections. In the absence of a polynomial-time exact algorithm, we formulate the problem as an ILP problem to find optimal solutions.

1) *Floating-Point Vertex:* A floating-point vertex v_i in the kernel graph should be mapped as a multicycle operation on a pair of PEs. To handle this, we add as many extra vertices $f_{ib} \in F^{vi}$ as necessary to represent the pair of PEs and the number of cycles required, where F^{vi} is the set of added vertices for the floating-point vertex v_i. Fig. 10 shows the transformation of floating-point vertices to the group of normal vertices. These extra vertices f_{ib} are tightly coupled with the original floating-point vertex v_i, such that all of them (f_{ib} and v_i) are mapped together as a group. For example, if the floating-point vertex v_i is mapped onto p_{hm} and takes four cycles to execute the floating-point operation, then f_{i1}, f_{i2}, f_{i3}, f_{i4}, f_{i5}, f_{i6}, and f_{i7} must be mapped onto $p_{h(m+1)}$, $p_{(h+1)m}$, $p_{(h+1)(m+1)}$, $p_{(h+2)m}$, $p_{(h+2)(m+1)}$, $p_{(h+3)m}$, and $p_{(h+3)(m+1)}$, respectively. In Fig. 10, only the dependencies $e(v_1, v_3)$ and $e(v_2, v_3)$ among the floating-point vertices are represented by the corresponding edges in the transformed graph, and all the other dependencies internal to a group are hidden since they are taken care of by the control logic explained in Section II-B.

2) *Routing Vertices and Routing PEs:* When two vertices are mapped, respectively, to two PEs that have no interconnections between them, we add extra dummy move vertices for data forwarding. Such vertices are called *routing vertices*, and the corresponding PEs are called *routing PEs*. To accommodate the routing PEs, we insert extra routing vertices into each edge of K. Since the exact number of routing PEs required for an edge $e_j \in E$ is not known until the mapping is completed, we insert the maximum possible number of routing vertices. Let R^{ej} be the set of routing vertices inserted into e_j, and let $R = \cup_{\forall i} R^{ej}$ be the set of all routing vertices. An upper bound of the number of routing PEs for edge e_j can be given by $|R^{ej}| \leq T_j^w - 1$, where T_j^w is obtained by subtracting ASAP schedule time of the tail vertex of edge e_j from as late as possible (ALAP) schedule time of the head vertex. To obtain the worst-case latency used in the ALAP scheduling, we perform list scheduling (details of the list scheduling are explained in Section V).

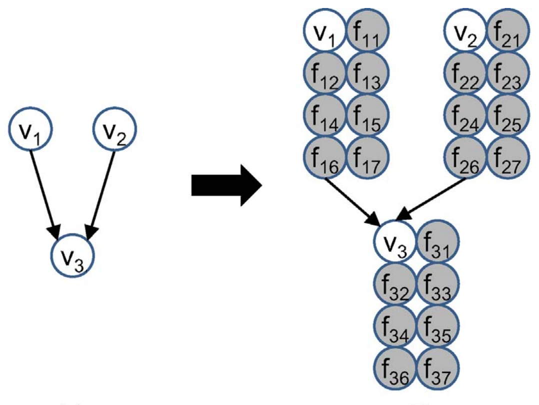
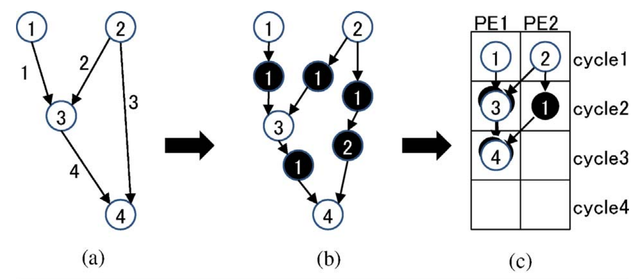

v_1
v_2
v_3
f_{11} f_{12} f_{13} f_{14} f_{15} f_{16} f_{17}
f_{21} f_{22} f_{23} f_{24} f_{25} f_{26} f_{27}
f_{31} f_{32} f_{33} f_{34} f_{35} f_{36} f_{37}
(a)
(b)

Fig. 10. Floating-point vertices are transformed to the group of several normal vertices. (a) Floating-point vertices. (b) Transformed to normal vertices.

For every loop-carried dependency edge $d_j = (v_p, v_q) \in D$, we also add extra dummy move vertices that can be mapped later to routing PEs for data forwarding. The number of dummy move vertices can be upper-bounded as $|R^{dj}| \leq T_j^{qALAP} - 1$, where T_j^{qALAP} is ALAP schedule time of the head vertex v_q. Then we take transformation $K\ K' = (V', E', D')$ by inserting $|R^{ej}|$ vertices into each edge $e_j \in E$ and $|R^{dj}$ vertices into each loop-carried dependency edge $d_j \in D$. Fig. 11 shows the original kernel K [Fig. 11(a)] transformed into K' by inserting the candidates of routing vertices [dark vertices in Fig. 11(b)] into every data-dependency edge in K. Then the problem of mapping K to C becomes the problem of mapping K' to C. Unlike normal vertices $v \in V$, the candidates of routing vertices $r \in R$ can be mapped to the same PE $p \in P$ of the CGRA, including the PE on which a normal vertex $v \in V$ is mapped. Now we define our ILP formulation for the transformed CDFG K'.

3) *Boolean Decision Variables:* For $1 \leq i \leq |V|, 1 \leq j \leq |E|, 1 \leq a \leq |R^e|, 1 \leq b \leq F^v|, 1 \leq h \leq H$, and $1 \leq m \leq M$ (these ranges of i, j, a, b, h, and m are used throughout this paper, unless otherwise specified) the ranges are as follows.

a) x_{ihm} is 1 if ith vertex $v_i \in V$ is mapped onto $p_{hm} \in P$.
b) y_{ibhm} is 1 if bth floating-point vertex $f_{ib} \in F^{vi}$ of ith vertex $v_i \in V$ is mapped onto $p_{hm} \in P$.



$$|V| = 4,\ |E| = 4,\ M = 2,\ WCET = 4,$$
$$T^{w}{}_{3} = ALAP(v_4) - ASAP(v_2) = 4 - 1 = 3,\ |R^{e3}| = T^{w}{}_{3} - 1 = 2,$$
$$x_{111} = 1,\ x_{112} = 0,\ \cdots,\ x_{212} = 1,\ x_{321} = 1,\ x_{431} = 1,\ \cdots$$
$$z_{1111} = 0,\ z_{1121} = 1,\ \cdots,\ z_{2121} = 1,\ z_{3122} = 1,\ z_{3231} = 1,\ z_{4131} = 1,\ \cdots$$
$$q_{1111} = 0,\ q_{1121} = 0,\ \cdots,\ q_{2121} = 0,\ q_{3122} = 1,\ q_{3231} = 0,\ q_{4131} = 0,\ \cdots$$

(d)

Fig. 11. Example of ILP formulation for routing vertices. (a) Kernel K. (b) K'. (c) Mapping result. (d) Variables in ILP.

c) z_{jahm} is 1 if ath routing vertex r_{ja}, $r_{ja} \in R^{ej}$ for $e_j \in E$ or $r_{ja} \in R^{dj}$ for $d_j \in D$, is mapped onto $p_{hm} \in P$.

d) q_{jahm} is 1 if ath routing vertex r_{ja}, $r_{ja} \in R^{ej}$ for $e_j \in E$ or $r_{ja} \in R^{dj}$ for $d_j \in D$, is mapped onto $p_{hm} \in P$, but no normal vertex is mapped onto p_{hm}, which means p_{hm} is used as an actual routing PE for the routing vertex. We call such a routing vertex an *actual routing vertex*.

e) The variables get 0, otherwise.

4) *Objective Function:* The objective to be minimized is the total latency when we map K' onto C. For this, we add a sink vertex v_s to K' and calculate the total latency T as follows:

$$T = \alpha + II \cdot (\lceil \beta/U \rceil - 1), \text{ when } \beta/U \le N \tag{5a}$$

$$(\alpha + II \cdot (N-1)) \cdot \lceil \beta/U/N \rceil, \text{ when } \beta/U > N \tag{5b}$$

where $\alpha = \sum_h \sum_m h \cdot x_{shm} - 1$ is the latency, i.e., (schedule time of v_s) $- 1$, II is the initiation interval, and U is the unroll factor.

5) *Constraints:* The constraints are given below.

a) Unique location

$$\sum_h \sum_m x_{ihm} = 1 \quad \forall i \tag{6}$$

$$\sum_h \sum_m z_{jahm} = 1 \quad \forall j, a \tag{7}$$

$$\sum_h \sum_m y_{ibhm} = 1 \quad \forall i, b. \tag{8}$$

b) Floating-point vertices are tightly coupled

$$x_{ihm} = y_{i0h(m+1)} = y_{i1(h+1)m} = y_{i2(h+1)(m+1)} = \cdots \quad \forall h, m. \tag{9}$$

c) Single mapping to a PE at a time

$$\sum_m x_{ihm} + \sum_i \sum_b y_{ibhm} \le 1 \quad \forall h, m. \tag{10}$$

d) Single mapping to a PE at a time, including routing vertices, except for the case of mapping a routing vertex together with the head vertex of the corresponding edge onto the same PE (note that multiple routing vertices can still be mapped to a PE)

$$z_{jahm} + \sum_{i: v_i \neq (\text{head vertex of } e_j)} x_{ihm} + \sum_i \sum_b y_{ibhm} \le 1 \ \forall h, m, j, a. \tag{11}$$

e) Single mapping to a PE at a time, including actual routing vertices (multiple actual routing vertices cannot be mapped to the same PE)

$$\sum_a q_{jahm} + \sum_a q_{j'ahm} + \sum_i x_{ihm} + \sum_i \sum_b y_{ibhm} \le 1 \quad \forall h, m, j, j' \tag{12}$$

except for the case where j and j' share a same tail vertex (this is to allow Steiner tree routing).

f) For an edge, the set of actual routing vertices is a subset of the set of routing vertices. Therefore, if $q_{jahm} = 1$, then $z_{jahm} = 1$

$$q_{jahm} - z_{jahm} \le 0 \quad \forall h, m, j, a. \tag{13}$$

g) Actual routing vertex (z=1 but no v is mapped) should set the corresponding variable q

$$q_{jahm} - z_{jahm} + \sum_i x_{ihm} \ge 0 \quad \forall h, m, j, a. \tag{14}$$

h) Shared resource

$$\sum_{i:(\text{type of } v_i)=t} \sum_{p=0}^{N-1} x_{i(h+p \cdot II)m} \le S^t, \forall t, m, h, 1 \le h \le H-(N-1) \cdot II \tag{15}$$

where S^t is the number of shared functional units of type t available on each row.

i) Data dependency in E

$$\sum_h \sum_m k \cdot x_{qhm} - \sum_h \sum_m k \cdot x_{phm} \ge T^p, \forall e = (v_p, v_q) \in E \tag{16}$$

where T^p is the latency of vertex v_p.

j) Loop-carried dependency in D when $|R^{dj}| = 0$

$$\forall d = (v_p, v_q) \in D',$$
$$\sum_h \sum_m h \cdot x_{phm} - \sum_h \sum_m h \cdot x_{qhm} = II - T^p \tag{17a}$$
$$\sum_h \sum_m m \cdot x_{phm} = \sum_h \sum_m m \cdot x_{qhm} \tag{17b}$$

when $|R^{dj}| > 0$

$$\forall d = (v_p, r_{j1}) \in D'$$

$$\sum_h \sum_m h \cdot x_{phm} - \sum_h \sum_m h \cdot z_{j1hm} = II - T^p \tag{18a}$$

$$\sum_h \sum_m m \cdot x_{phm} = \sum_h \sum_m m \cdot z_{j1hm}. \tag{18b}$$

Constraints (17a) and (18a) imply that v_q (or r_{j1}), which consumes loop-carried data, must be located one cycle after the schedule time of v_p, which generates the loop-carried data. These constraints are necessary for the architecture having distributed register files, since the consumer must fetch the data before the data on the output register of the producer is overwritten. Constraints (17b) and (18b) imply that v_p and v_q (or r_{j1}) must be mapped to PEs on the same row. These constraints are necessary for the architecture having interconnection between PEs only on the same row (no diagonal connections).

k) Interconnection and data dependency in E' (or D') when $|R^{ej}|$(or $|R^{dj}$) = 0

$$\forall e = (v_p, v_q) \in E'(or D'),$$
$$w_{h_1m_1h_2m_2} - x_{ph_1m_1} - x_{qh_2m_2} \geq -1 \quad \forall h_1, m_1, h_2, m_2 \quad (19)$$

when $|R^{ej}| > 0$

$$\forall e = (v_p, r_{j1}) \in E'(or D'), w_{h_1m_1h_2m_2} - x_{ph_1m_1} - z_{j1h_2m_2} \geq -1 \quad \forall h_1, m_1, h_2, m_2 \quad (20a)$$

$$\sum_h \sum_m h \cdot z_{j1hm} - \sum_h \sum_m h \cdot x_{phm} > 0 \quad (20b)$$

$$\forall e = (r_{ja}, r_{j(a+1)}) \in E'(or D'), 1 \leq a \leq \left|R^{ej}\right| (or \left|R^{dj}\right|) - 1$$
$$w_{h_1m_1h_2m_2} - z_{jah_1m_1} - z_{j(a+1)h_2m_2} \geq -1 \quad \forall h_1, m_1, h_2, m_2 \quad (21a)$$

$$\sum_h \sum_m h \cdot z_{j(a+1)hm} - \sum_h \sum_m h \cdot z_{jahm} \geq 0 \quad (21b)$$

$$\forall e = (r_{j\left|R^{ej}\right|(or\left|R^{dj}\right|)}, v_q) \in E'(or D'), w_{h_1m_1h_2m_2} - z_{j\left|R^{ej}\right|(or\left|R^{dj}\right|)h_1m_1} - x_{qh_2m_2} \geq -1 \quad \forall h_1, m_1, h_2, m_2 \quad (22a)$$

$$\sum_h \sum_m h \cdot x_{qhm} - \sum_h \sum_m h \cdot z_{j\left|R^{ej}\right|(or\left|R^{dj}\right|)hm} \geq 0 \quad (22b)$$

where w is a 4-D adjacency matrix that represents the interconnection between the PEs determined as follows.

i) $w_{h1m1h2m2}$ is 1 if p_{h1m1} is directly connected to p_{h2m2} by interconnect resources or if the two PEs are same (i.e., $h_1 = h_2$ and $m_1 = m_2$).

ii) $w_{h1m1h2m2}$ is 0 if $m_1 \neq m_2$ (different PEs) and there is no direct connection between p_{h1m1} and p_{h2m2}.

iii) Otherwise ($m_1 = m_2$ and $h_1 \neq h_2$), $w_{h1m1h2m2}$ is 1 if p_{h1m1} is connected to p_{h2m2} (i.e., the output of p_{h1m1} is transferred to p_{h2m2}) by a local register and is 0 otherwise (23).

Constraints (20a), (21a), and (22a) imply that there should be interconnection links in C that implement the paths in K'. Constraints (20b), (21b), and (22b) represent the dependencies among v_p, r_{ja}, and v_q of each edge $e = (v_p, v_q) \in E'$ and loop-carry dependency edge $d = (v_p, v_q) \in D'$.

l) Number of local registers in each PE

$$\sum_{h_1:h_1 \leq h} \sum_{h_2:h_2 > h} w_{h_1mh_2m} \leq L \quad \forall h, m \quad (23)$$

where L is the total number of local registers in each PE.

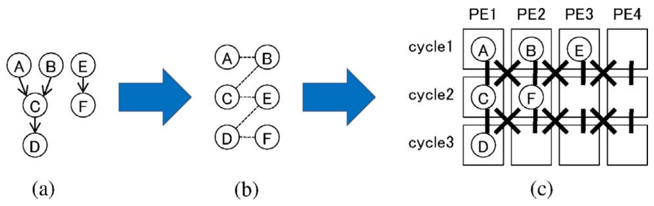


Fig. 12. Application mapping on CGRA using list scheduling and temporal mapping techniques. (a) CDFG. (b) Topologically sorted list. (c) Mapping.

V. Heuristic Algorithm

ILP-based application mapping yields an optimal solution. However, it takes an unreasonably long execution time to find a solution, making it unsuitable for large designs, as well as for fast design space exploration of CGRAs. Here, we present a fast heuristic mapping algorithm considering Steiner points. It is based on a mixture of two algorithms: list scheduling and QEA.

A. *List Scheduling*

Over the unrolled CDFG, we run list scheduling with the given resource constraint in order to obtain an initial solution. As mentioned in Section III-C, the initial solution is used for the two paths: QEA and ILP. For QEA, the initial solution is used as the starting point of the evolutionary algorithm. For ILP, the latency given by the initial solution is used for setting the worst-latency in the ILP formulation.

The list scheduling algorithm first topologically sorts the vertices from the sink to the source. If a vertex has a longer path to the sink, then it gets a higher priority. Fig. 12(a) shows a CDFG example and Fig. 12(b) shows the topologically sorted list of the vertices. In Fig. 12(c), the squares represent the PEs of the CGRA, and bold lines represent interconnections between PEs. We assume mesh connectivity of 16 (4 × 4) PEs in an array. In the temporal mapping, four PEs in each column perform the operations in one iteration of a loop in *cycle1*, *cycle2*, *cycle3*, and so on, as shown in Fig. 12(c). Since data transfer between neighbor PEs takes one cycle, the data is actually transferred to the PEs in the next cycle. So the communication pattern of a column of the CGRA looks like the one in Fig. 12(c).

From the sorted list, the algorithm selects and schedules the vertex with the highest priority if all the predecessor vertices have been scheduled and the selected vertex is reachable from all the scheduled predecessor vertices through the interconnections available in the CGRA. If the vertex is a floating-point vertex, the algorithm checks to see if neighbor PEs are busy, since executing a floating-point operation requires a pair of PEs for several cycles.

When mapping a vertex onto a PE, we consider interconnect constraint and shared resource constraint. If there is no direct connection available for implementing a data dependency between two PEs, we try to find a shortest path consisting of unused PEs which work as routers. We should also consider the constraint set by sharing area-critical functional units [note that (11) in the ILP formulation represents the same constraint]. For example, if we have only one multiplier shared among the PEs in a row, we cannot schedule two multiply

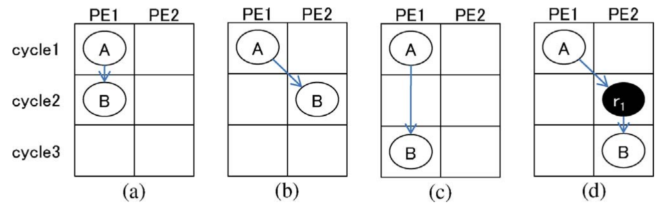


Fig. 13. Several data transfer cases. For some cases, we have to use a routing PE. (a) Forwarding. (b) Interconnect. (c) Local register. (d) Routing PE.

operations successively but should wait for N (number of PEs in a row) cycles after scheduling one multiply operation, since the multiplier must be used by PEs in other columns for loop pipelining. In this case, the second multiply operation may need to wait with proper routing of the input data.

Fig. 13 shows different ways of data transfer according to the scheduling and binding result. In the cases of Fig. 13(a)–(c), we can use internal forwarding, interconnection between neighbors, and a local register, respectively, for the data transfer. However, in case of Fig. 13(d), we have to use a routing PE for data transfer. In this case, we try to find a shortest path with unused remaining PEs between v_A and v_B. For finding the shortest path, we first make a graph with the remaining PEs which have not been used yet. The remaining PEs and their interconnections are represented, respectively, by vertices and edges of the graph. Then we use the Dijkstra's algorithm to find a shortest path; the vertices r_i in this shortest path become routing vertices.

One important thing to be considered is that as the found paths are implemented, the number of remaining PEs will be decreased, which in turn decreases the chance to find a solution in a later phase. Therefore, we have to take care in ordering the edges to be routed. In our list scheduling, the order of edges is fixed by the topological sorting of the vertices. This ordering problem will be addressed in the QEA formulation as described next.

B. *Quantum-Inspired Evolutionary Algorithm*

The QEA is an evolutionary algorithm that is known to be very efficient compared to other evolutionary algorithms [18]. It represents the solution space by encoding each solution with a set of Q-bits. A Q-bit is the smallest unit of information in QEA, which is defined with a pair of numbers (α, β) where $|\alpha|^2 + |\beta|^2 = 1$. $|\alpha|^2$ gives the probability of the Q-bit to be found in the "0" state and $|\beta|^2$ gives the probability of the Q-bit to be found in the "1" state. A Q-bit may be in the "1" state, in the "0" state, or in a linear superposition of the two and this is how QEA maintains many potential solutions in a compact way, thereby enabling much faster design space exploration than any other currently known evolutionary algorithm. The concepts of Q-bit and superposition of states originate from quantum computing and that is why the algorithm is called QEA. It has been first used for optimizing electronic designs in [12] and also successfully used by others [26]–[28].

1) *Q-Bit Encoding:* To solve the problem of mapping an application onto the CGRA with QEA, we use a 2-D array ($H \times M$) of Q-bits for each operation in the CDFG. In the Q-bit array, each row represents binding and each column represents scheduling. Thus, if a Q-bit entry in the array at location (h, m), $1 \leq h \leq H$, and $1 \leq m \leq M$, is evaluated to 1, that means the operation is bound to mth PE and scheduled at control step h. Since an operation should be scheduled at only one control step and mapped to only one PE, we adopt one-hot encoding. Thus only one entry in the array should be evaluated to "1" and all the others should be evaluated to "0."

2) *Q-Bit Generation and Evaluation:* The QEA, like genetic algorithms, generates tens or hundreds of possible cases (through rotation instead of crossover operation) and evaluates each case to choose the best solution. After generating a given number of possible cases in a generation, we observe the array of Q-bits, choose only one Q-bit with highest probability (i.e., largest $|\beta|^2$) among the Q-bits that satisfy constraints on data dependency, and then set its state to "1."

The fitness function that we use for the QEA is the performance, which is the inverse of the total latency given by (5a) and (5b) in the ILP formulation. The solution is then used to produce the next generation. This iterative improvement continues until it finds a solution that has the same latency as the ASAP scheduling result, or there is no improvement during a given time interval, T_{termi}. We stop the iteration if it reaches the latency of the ASAP solution since it is a lower bound of optimal latency.

The QEA starts from the list scheduling result as a seed and attempts to further reduce the total latency. Since the QEA starts with a relatively good initial solution, it tends to reach a better solution sooner than starting with a random seed.

3) *Routing Path Finding:* When the schedule and binding of each vertex are determined, it tries to find the routing path among the vertices with unused remaining PEs to see if these schedule and binding results violate the interconnect constraint. In this routing phase, we consider the order of edges to be routed as mentioned in the previous section. The priority used for the ordering is determined as follows.

a) Edges located in the critical path get higher priority.
b) Among the edges located in the critical path, edges that have smaller slack (shorter distance) get higher priority.
c) If a set of edges have the same tail vertex, then the set of edges becomes a group and the priority of this group is determined by the highest priority among the group members.

According to the above priority, we make a list of candidate edges and find a shortest path for each edge in the order of priority with the Dijkstra's algorithm. In this routing phase, we consider a Steiner tree (instead of a spanning tree) for multiple writes from a single source. Our heuristic algorithm for finding a Steiner tree tries to find a path individually for each outgoing edge from the source. If some paths use the same routing PE, it becomes a Steiner point. Although this approach may not always find an optimal path, it gives good solutions in most of the cases if not all. Indeed, our experimental results presented in the next section show that the approach finds optimal solutions for 97% of 2700 randomly generated examples.

VI. Experiments

A. Experimental Setup

To demonstrate the effectiveness of our approach, we have performed three different experiments: 1) experiment with an illustrative example to show that our algorithm based on Steiner tree gives a better solution than spanning tree; 2) experiment with a random set of CDFG examples for both integer operations and floating-point operations; and 3) experiment with several examples from standard benchmarks and real applications. For the integer type applications, we have chosen a collection of standard benchmarks from the DSPStone, Livermore loops, and Mediabench. For the floating-point type applications, we have chosen a collection of DSPStone benchmarks for floating-point and physics engine of 3-D graphics.

The second experiment attempts to exercise our approach with a variety of random graphs, for which we have devised a random kernel CDFG generator. Our CDFG generator randomly generates 100 integer type CDFGs for each value of node cardinalities from 5 to 20 nodes (1600 in total), and 100 floating-point type CDFGs for each value of node cardinalities from 5 to 15 nodes (1100 in total). Each CDFG is generated according to the following steps. Once the number of nodes is given, the operations which are supported by a CGRA PE (including shared functional units) are randomly assigned to each node. Then edges are inserted such that each node has exactly the same number of incoming edges as the number of operands for the corresponding operation type and should have at least one outgoing edge. All experiments have been done on Pentium4, 3.0 GHz dual processor machine with 2 GB RAM. We have used glpk 4.38 [19] for solving the ILP formulation.

We have designed the CGRA at the RT-level in VHDL, and synthesized the gate-level circuit with Design Compiler [29] using 0.13 μm technology. The synthesis result shows that the CGRA runs at 200 MHz. It is about the same clock frequency of a typical ARM9 processor in a similar synthesis environment. In our experiments, we have used the same clock for all components in the system. Specifically, the processor and the reconfigurable array have been synchronized within one clock domain.

B. Experimental Results

1) *Illustrative Example:* Table I shows the experimental result of the butterfly addition example shown in Fig. 14. We consider four PEs in a column that has mesh connectivity. The approach using Steiner tree gives better solution than the one using spanning tree as shown in Fig. 15. Both *ILP* and *Heuristic* approaches give optimal solutions. However, the *Heuristic* approach is two orders of magnitude faster than the *ILP* approach.

With the help of the automatic generation of configurations for the CGRA, architecture variations can be easily derived. We consider three different interconnect topologies: mesh, mesh-plus, and full. Fig. 16 shows the three different interconnect topologies of 4 × 4 array of PEs. In the mesh-plus connectivity, interconnects of two-hop distance are added to the mesh interconnects. Table II shows the effect of three different interconnect topologies on the performance and mapping time of the butterfly addition example. The *Heuristic* approach always gives optimal solutions for the three different architecture instances. As expected, the more interconnect resources are provided between PEs, the less mapping time is needed, and the less cycles are needed to run the application. It is due to the fact that adding more interconnect resources renders better routability between PEs, and therefore the operations can be easily scheduled without spending much time to find routing paths between the operations. However, adding more interconnect resources may lead to more wires and wider multiplexors, which incurs more area, longer delay, and more power consumption.

2) *Randomly Generated CDFGs:* We experiment with randomly generated CDFGs for integer type as well as floating-point type mapped on the architecture instance having mesh-plus connectivity. Fig. 17(a) shows the mesh-plus connectivity of 8 PEs in a column. When our CGRA is used for floating-point operations, the interconnect topology becomes mesh connectivity as shown in Fig. 17(b). Fig. 18(a) shows the

TABLE I
Experimental Result of Butterfly Addition Example

		Latency (Cycle)	Mapping Time (s)
ILP	Spanning tree	5	1022
	Steiner tree	4	965
Heuristic	Spanning tree	5	13
	Steiner tree	4	9

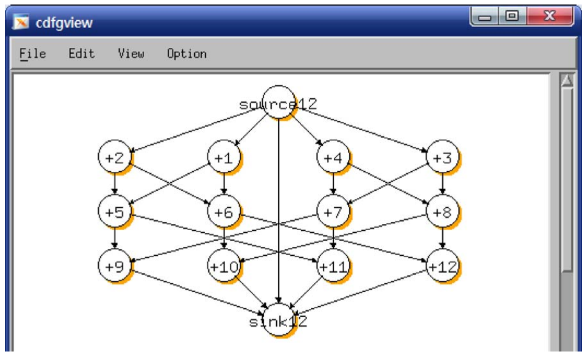


Fig. 14. CDFG view of a butterfly example.

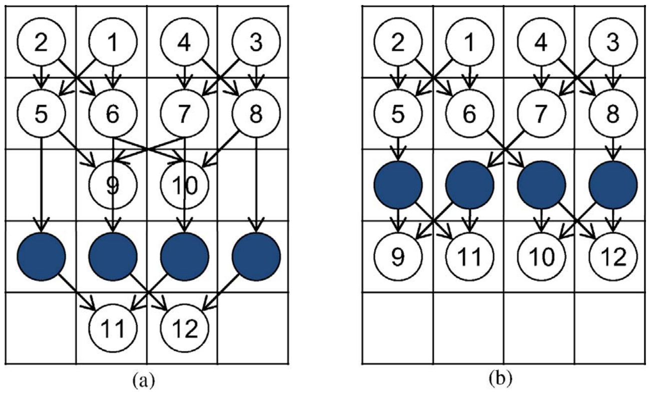


Fig. 15. Mapping result of the butterfly example. Dark nodes represent routing nodes. (a) Routing with spanning tree. (b) Routing with Steiner tree.

TABLE II
EXPERIMENTAL RESULT OF BUTTERFLY ADDITION EXAMPLE ACCORDING TO THE THREE DIFFERENT INTERCONNECT TOPOLOGIES

	Mesh	Mesh-Plus	Full
Latency (cycle)	4	3	3
Mapping time (s)	9	<1	<1

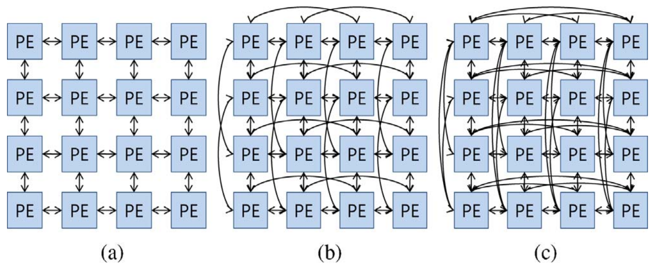


Fig. 16. Three different interconnect topologies. (a) Mesh. (b) Mesh-plus. (c) Full.

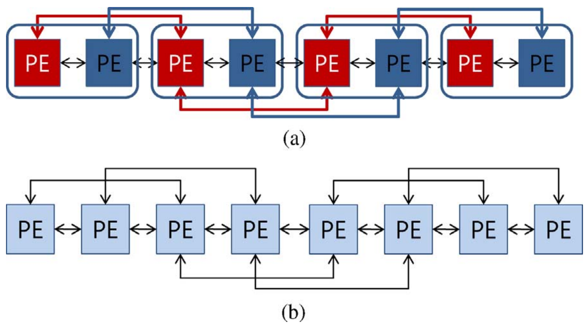
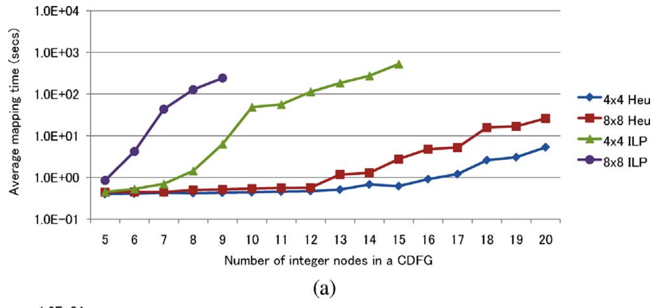


Fig. 17. Interconnect topology for the PEs in a column. (a) Mesh-plus connectivity used for integer operations. (b) Mesh (bold line) connectivity used for floating-point operations.

experimental result of 100 randomly generated integer type examples for each problem size (from 5 integer nodes to 20 integer nodes), and Fig. 18(b) shows the experimental result of 100 randomly generated floating-point type examples for each problem size (from 5 floating-point nodes to 15 floating-point nodes). The x-axis represents the number of nodes that each example contains, and the y-axis shows the log scaled average mapping time for *ILP* and *Heuristic* approaches. In this experiment, we consider two different architecture instances: 4×4 PE array and 8×8 PE array. Since *ILP* takes a lot of time for large input graphs, we stop the experiment of *ILP* when it fails to find a solution within 3600 s.

For both integer and floating-point experiments, the *Heuristic* approach is about two orders of magnitude faster than the *ILP* approach in average. From Fig. 18, we see that, generally, problems with more PEs take more time to get a solution (since the algorithm has to handle more variables). However, for the floating-point experiment in Fig. 18(b), the *Heuristic* approach takes more time on 4×4 PE array than on 8×8 PE array. The reason is as follows. In the *Heuristic* approach, we set the QEA to terminate early when it finds a solution having the same performance as the ASAP solution. However, since floating-point examples have big granularity [each floating-point operation requires two PEs and multiple cycles, see Fig. 9(b)] and interconnect resources are scarcer



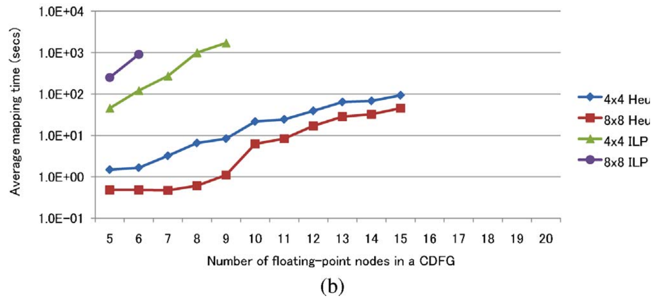
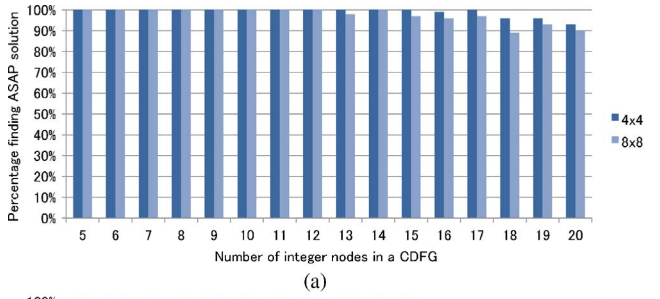


Fig. 18. Average mapping time of randomly generated CDFG examples. (a) Integer-type random examples. (b) Floating-point-type random examples.



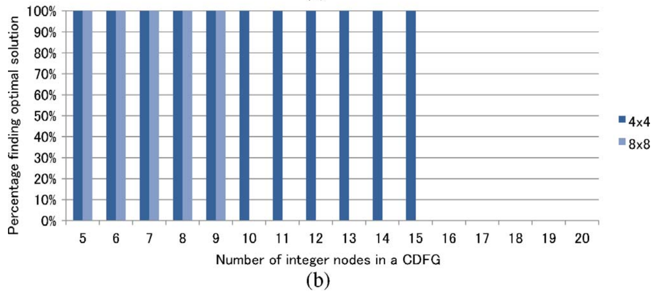
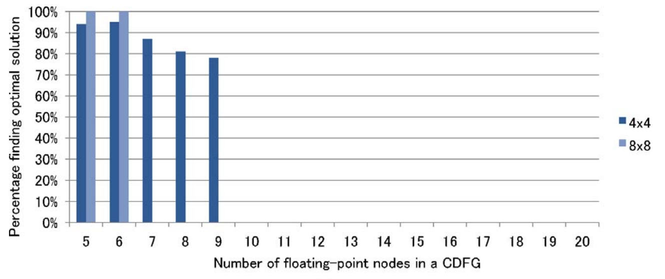


Fig. 19. Percentage of finding an optimal solution with the *Heuristic* approach. (a) Integer-type random examples. (b) Floating-point-type random examples.

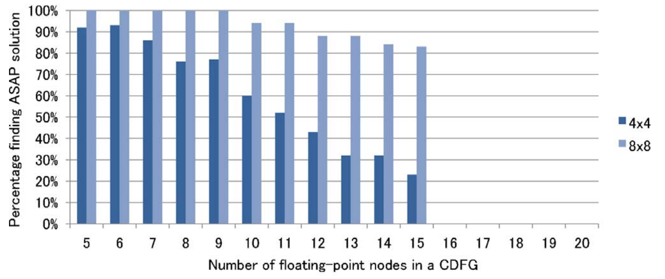
than for integer operations [integer operations get mesh-plus interconnectivity but floating-point operations get mesh interconnectivity, see Fig. 17(b)], it is hard and thus takes longer time to find an optimal solution for fewer PEs.

Fig. 19 shows the percentage of finding an optimal solution (same performance as ILP solutions) with the *Heuristic* approach for 100 integer examples [Fig. 19(a)] and 100 floating-point examples [Fig. 19(b)] for each problem size. The *Heuristic* approach finds an optimal solution for 100% of the integer cases and 87% of the floating-point cases.

SINCE it takes too much time for large-sized problems to get an optimal solution with the *ILP* approach (in some cases, we cannot find the optimal solutions within 24 h), we also compare the *Heuristic* approach with the ASAP scheduling in

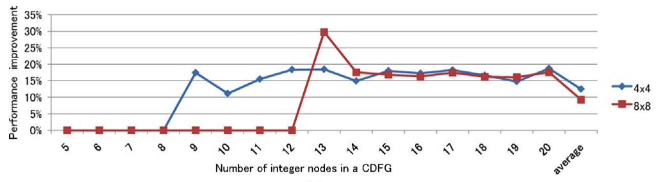


(a)

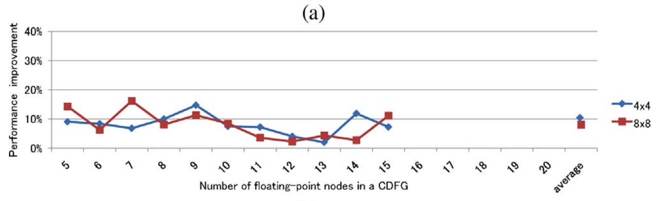


(b)

Fig. 20. Percentage of finding ASAP solution with the *Heuristic* approach. (a) Integer type random examples. (b) Floating-point-type random examples.



(a)

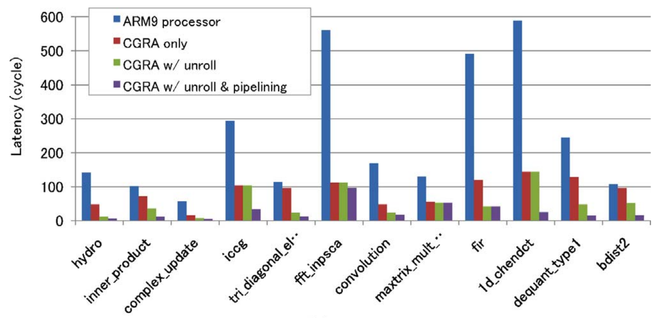


(b)

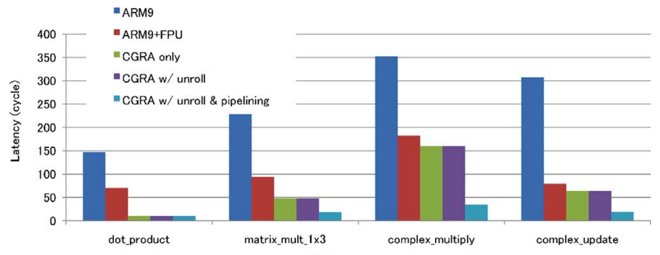
Fig. 21. Performance comparison between spanning tree and Steiner tree approaches of random examples. (a) Integer type random examples. (b) Floating-point-type random examples.

Fig. 20. Note that an ASAP result gives a lower bound of the optimal latency. Therefore, even if the *Heuristic* approach did not achieve the performance of ASAP solution for some cases, it could have achieved optimality. When compared with the ASAP solution, the *Heuristic* approach finds an optimal solution for 98% of the integer cases and 77% of the floating-point cases. Unfortunately the percentage of finding an ASAP-quality solution for the floating-point cases with the 4×4 PE array decreases rapidly as the number of nodes in the CDFG increases. From Figs. 18(b) and 20(b), we see that it is important to give enough resources for large-sized problems with floating-point operations.

Fig. 21 shows the average performance improvement obtained by the *Steiner tree* approach compared to the result by the *spanning tree* approach for both integer type [Fig. 21(a)] and floating-point type [Fig. 21(b)] random examples. Our approach using *Steiner tree* gives 11% and 8% average performance improvement over the one using *spanning tree* for integer cases and floating-point cases, respectively.



(a)



(b)

Fig. 22. Performance comparison between ARM9 processor and CGRA for benchmark examples. CGRA gives performance improvement by adopting loop level optimizations such as loop unrolling and pipelining. (a) Integer type benchmark examples. (b) Floating-point-type benchmark examples.

3) *Standard Benchmarks and 3-D Graphics:* To demonstrate the effectiveness and usefulness of our algorithm, we experiment with a set of benchmarks collected from the Livermore loops, Mediabench, and DSPStone benchmark suites for integer domain application. We assume eight PEs in a column with mesh-plus interconnectivity. Fig. 22(a) shows the total latencies obtained on the CGRA by our *Heuristic* approach together with those obtained by software implementation on ARM9 processor. For the experiments with ARM9 processor, we assume that all used data are already stored in the processor's local memory that takes one clock cycle for access. In other words, the memory access overheads of both ARM9 and RCM are the same.

Table III shows the characteristics of the examples and the experimental results. The examples with zero *recMII* (ideal minimum recurrence initiation interval) have no loop-carried dependency and all other examples have loop-carried dependencies. In some cases, the *ILP* approach does not find solutions within 24 h (86 400 s). Thus, we compare the latency for single iteration [i.e., α in (1)–(7)] with the ASAP solution and also compare the resulting *II* with the *recMII*. The ASAP solution is obtained by scheduling the operations while considering the data-dependency but without any constraints on the number of resources and interconnect topology. The *recMII* is obtained by analyzing the loop-carried dependency on the ASAP solution. Note that the ASAP solution and the *recMII* together provide a lower bound of the optimal solution. In Table III, the iccg example gives three times worse *II* result than the *recMII*. The example has complicated data dependency than the others, which causes the increase of the *II* mainly due to the limited interconnect resources. In the matrix_mult_4 $\times$ 4 example, the ASAP solution maximally exploits the unlimited resources (PEs as well as interconnects) to achieve five cycles latency. In reality, however, since we

TABLE III
CHARACTERISTICS AND EXPERIMENTAL RESULT OF INTEGER BENCHMARK EXAMPLES

	# of Nodes	# of Iter.	Unroll Factor	Mapping Time (s)		Latency (Cycle)			Initiation Interval (Cycle)		
				Heu.	ILP	Heu.	ILP	ASAP	Heu.	ILP	recMII
hydro (k = 4)	20	8	4	1	1	6	6	6	1	1	0
inner_product	14	8	2	3	140	5	5	5	1	1	0
complex_update	16	4	2	1	46	4	4	4	1	1	0
iccg (k=8)	16	8	1	108	>86 400	13	–	13	3	–	1
tri_diagonal_elimination (k = 4)	36	8	4	1	1	12	12	12	1	1	0
fft_inpsca	12	16	1	15	>86 400	8	–	5	7	–	4
convolution	8	16	4	1	2	6	6	6	4	4	2
matrix_mult_4 × 4	145	4	4	124	>86 400	60	–	5	1	–	0
fir (k = 40)	240	40	40	228	>86 400	45	–	42	1	–	0
1d_chendct (jpeg_encoder)	60	8	1	221	>86 400	18	–	11	1	–	0
dequant_type1 (mpeg4_decoder)	40	16	4	2	>86 400	12	–	8	1	–	0
bdist2 (mpeg2_encoder)	64	8	2	3	10	12	12	12	3	3	3

TABLE IV
CHARACTERISTICS AND EXPERIMENTAL RESULT OF FLOATING-POINT BENCHMARK EXAMPLES

	# of Nodes	# of Iterations	Unroll Factor	Mapping Time (s)		Latency (Cycle)	
				Heuristic	ILP	Heuristic	ILP
dot_product	3	1	1	1	1	10	10
matrix_mult_1 × 3	5	3	1	1	235	18	18
complex_multiply	6	16	1	1	1	34	34
complex_update	8	4	1	1	581	19	19

have only 8 PEs in a column of the CGRA, the actual mapping takes 60 cycles.

For the floating-point domain applications, we experiment with a collection of DSPStone floating-point benchmark examples and physics engine examples from 3-D graphics. Again, we assume eight PEs in a column with mesh-plus interconnectivity (it enables four concurrent floating-point operations per column). Fig. 22(b) shows the total latencies obtained on the CGRA by our *Heuristic* approach together with those obtained by software implementation on an ARM9 processor for the DSPStone floating-point benchmark examples. The applications used in these floating-point experiments have no loop-carried dependency. Although floating-point applications with loop-carried dependency ($recMII > 0$) can also be mapped onto the CGRA by the same approach, it has not been included in the current implementation. Table IV shows the characteristics of the examples and the results.

Table V shows performance comparison between the software implementation on ARM9 processor and the CGRA implementation obtained by the *Heuristic* approach for physics engine kernels from 3-D graphics. In this experiment, we compare the performance only for a single iteration of each kernel loop. Implementation of multiple iterations with loop pipelining would provide much higher speedups. We cannot get the result of the *ILP* approach since it takes too much time (more than 24 h). However, the *Heuristic* approach takes a few minutes for all the cases. Speedups for the kernels are up to 119.9 times compared to the software only implementation.

To see the effect of the optimization by the QEA, we compare the performance result obtained by the QEA with that obtained by the list scheduling for the integer benchmark examples. Fig. 23 shows the performance improvement of

TABLE V
PERFORMANCE COMPARISON OF PHYSICS ENGINE KERNELS

	# of Nodes	ARM (Cycle)	CGRA (Cycle)	Speedup
Kernel1 (cloth)	48	5462	98	x55.7
Kernel2 (crash-wall)	26	10 673	89	x119.9
Kernel3 (crash-wall)	14	0673	48	x21.4

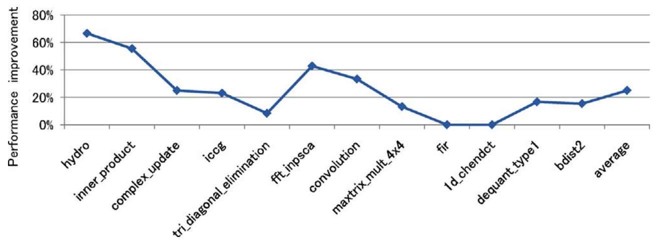


Fig. 23. Performance comparison between the QEA and the list scheduling for integer benchmark examples.

the QEA compared to the list scheduling result. The QEA gives 25% average performance improvement over the list scheduling result.

VII. CONCLUSION

In this paper, we presented a slow but optimal approach and fast heuristic approaches to mapping of applications from multiple domains onto a CGRA supporting both integer and floating point operations. In particular, we considered Steiner tree routing since it gives better result than spanning tree routing. After presenting an ILP formulation for an optimal solution, we presented a fast heuristic approach based on HLS techniques that performs loop unrolling and pipelining, which achieves drastic performance improvement. For randomly generated test examples, we showed that the proposed heuristic approach considering Steiner points finds 97% of the optimal solutions within a few seconds. Experiments on various benchmark examples and 3-D graphics examples have shown that our approach targeting a CGRA achieves up to 119.9 times performance improvement compared to software only implementation.

REFERENCES

[1] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. 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.

[2] *PACT XPP Technologies* [Online]. Available: http://www.pactxpp.com

[3] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Proc. FPLA*, 2003, pp. 61–70.

[4] *Chameleon Systems, Inc.* [Online]. Available: http://www.chameleon-systems.com

[5] T. J. Callahan and J. Wawrzynek, "Instruction-level parallelism for reconfigurable computing," in *Proc. IWFPL*, 1998, pp. 248–257.

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

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

[8] T. Toi, N. Nakamura, Y. Kato, T. Awashima, K. Wakabayashi, and L. Jing, "High-level synthesis challenges and solutions for a dynamically reconfigurable processor," in *Proc. ICCAD*, Nov. 2006, pp. 702–708.

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

[10] S. Friedman, A. Carroll, B. V. Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "SPR: An architecture-adaptive CGRA mapping tool," in *Proc. FPGA*, 2009, pp. 191–200.

[11] J. Yoon, A. Shrivastava, S. Park, M. Ahn, and Y. Paek, "A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architecture," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 17, no. 11, pp. 1565–1578, Jun. 2008.

[12] Y. Ahn, K. Han, G. Lee, H. Song, J. Yoo, and K. Choi, "SoCDAL: System-on-chip design accelerator," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 13, no. 1, pp. 171–176, Jan. 2008.

[13] 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*, Mar. 2005, pp. 12–17.

[14] Y. Kim, I. Park, K. Choi, and Y. Paek, "Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architecture," in *Proc. ISLPED*, Oct. 2006, pp. 310–315.

[15] *The SUIF Compiler System* [Online]. Available: http://suif.stanford.edu

[16] C. O. Shields, Jr., "Area efficient layouts of binary trees in grids," Ph.D. dissertation, Dept. Comput. Sci., Univ. Texas, Dallas, 2001.

[17] G. Lee, S. Lee, and K. Choi, "Automatic mapping of application to coarse-grained reconfigurable architecture based on high-level synthesis techniques," in *Proc. ISOCC*, Nov. 2008, pp. 395–398.

[18] K. Han and J. Kim, "Quantum-inspired evolutionary algorithms with a new termination criterion, Hε gate, and two phase scheme," *IEEE Trans. Evol. Computat.*, vol. 8, no. 2, pp. 156–169, Apr. 2004.

[19] *GNU Linear Programming Kit* [Online]. Available: http://www.gnu.org/software/glpk

[20] G. Lee, S. Lee, K. Choi, and N. Dutt, "Routing-aware application mapping considering Steiner point for coarse-grained reconfigurable architecture," in *Proc. ARC*, 2010, pp. 231–243.

[21] M. Jo, V. K. P. Arava, H. Yang, and K. Choi, "Implementation of floating-point operations for 3-D graphics on a coarse-grained reconfigurable architecture," in *Proc. IEEE-SOCC*, Sep. 2007, pp. 127–130.

[22] Z. Baidas, A. D. Brown, and A. C. Williams, "Floating-point behavioral synthesis," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 20, no. 7, pp. 828–839, Jul. 2001.

[23] J. L. Tripp, K. D. Peterson, C. Ahrens, J. D. Poznanovic, and M. B. Gokhale, "TRIDENT: An FPGA compiler framework for floating-point algorithms," in *Proc. FPL*, 2005, pp. 317–322.

[24] J. L. Rice, K. H. Abed, and G. R. Morris, "Design heuristics for mapping floating-point scientific computational kernels onto high performance reconfigurable computers," *J. Comput.*, vol. 4, no. 6, pp. 542–553, Jun. 2009.

[25] J. M. P. Cardoso and P. C. Diniz, *Compilation Techniques for Reconfigurable Architectures*. New York: Springer, 2009.

[26] S. Kim, C. Park, and S. Ha, "Architecture exploration of NAND flash-based multimedia card," in *Proc. DATE*, Mar. 2008, pp. 218–223.

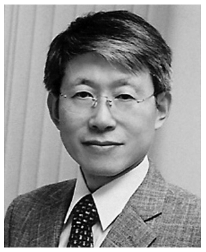
[27] Y. Joo, S. Kim, and S. Ha, "On-chip communication architecture exploration for processor-pool-based MPSoC," in *Proc. DATE*, Apr. 2009, pp. 466–471.

[28] H. Yang and S. Ha, "Pipelined data parallel task mapping/scheduling technique for MPSoC," in *Proc. DATE*, Apr. 2009, pp. 69–74.

[29] *Synopsys Corporation* [Online]. Available: http://www.synopsys.com

Ganghee Lee received the B.S. degree in electronics engineering from Ajou University, Seoul, Korea, in 2001, and the M.S. and Ph.D. degrees in electrical engineering and computer science from Seoul National University, Seoul, in 2003 and 2010, respectively.

Currently, he is a Senior Engineer with the SoC Platform Development Team, System-LSI Division, Samsung Electronics Company, Gyeonggi, Korea. His current research interests include communication architecture design and software synthesis for reconfigurable computing. He is also interested in high-level synthesis and multimedia architecture design.



Kiyoung Choi (M'88–SM'08) received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea, in 1978, the M.S. degree in electrical and electronics engineering from the Korea Advanced Institute of Science and Technology, Seoul, in 1980, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1989.

From 1989 to 1991, he was with Cadence Design Systems, Inc., Santa Clara, CA. In 1991, he joined the faculty of the School of Electrical Engineering and Computer Science, Seoul National University. His current research interests include various aspects of computer-aided electronic systems design, including embedded systems design, high-level synthesis, and low-power systems design. He is also interested in computer architecture and especially in configurable and reconfigurable computer architecture design.

Nikil D. Dutt (SM'96–F'08) received the B.E. (honors) degree in mechanical engineering from the Birla Institute of Technology and Science, Pilani, India, in 1980, the M.S. degree in computer science from Pennsylvania State University, University Park, in 1983, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign, Urbana, in 1989.

Currently, he is a Chancellor's Professor with the University of California, Irvine, with academic appointments in the Departments of Computer Science and Electrical Engineering. His current research interests include embedded systems, electronic design automation, computer architecture, optimizing compilers, system specification techniques, distributed embedded systems, formal methods, and brain-inspired architectures and computing.

Dr. Dutt received Best Paper Awards at CHDL89, CHDL91, VLSIDesign2003, CODES+ISSS 2003, CNCC 2006, ASPDAC-2006, and IJCNN 2009. He currently serves as an Associate Editor of the *ACM Transactions on Embedded Computer Systems* and the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATED SYSTEMS. He served as the Editor-in-Chief of the *ACM Transactions on Design Automation of Electronic Systems* from 2004 to 2008. He was an ACM SIGDA Distinguished Lecturer from 2001 to 2002, and an IEEE Computer Society Distinguished Visitor from 2003 to 2005. He has served on the steering, organizing, and program committees of several premier computer-aided design and embedded system design conferences and workshops, including DAC, DATE, ESWEEK, ICCAD, ISLPED, RTAS, and RTSS. He serves or has served on the advisory boards of ACM SIGBED and ACM SIGDA, the ACM Publications Board, and has previously served as the Vice-Chair of ACM SIGDA and IFIP WG 10.5. He is an ACM Distinguished Scientist and an IFIP Silver Core Awardee.