Efficient Mapping of Task Graphs onto Reconfigurable Hardware Using Architectural Variants

Miaoqing Huang, *Member, IEEE*,
Vikram K. Narayana, *Member, IEEE*,
Mohamed Bakhouya, Jaafar Gaber, *Member, IEEE*, and
Tarek El-Ghazawi, *Fellow, IEEE*

Abstract—High-performance reconfigurable computing involves acceleration of significant portions of an application using reconfigurable hardware. Mapping application task graphs onto reconfigurable hardware is therefore of rising attention. In this work, we approach the mapping problem by incorporating multiple architectural variants for each hardware task; the variants reflect tradeoffs between the logic resources consumed and the task execution throughput. We propose a mapping approach based on genetic algorithm, and show its effectiveness for random task graphs as well as an *N*-body simulation application, demonstrating improvements of up to 78.6% in the execution time compared with choosing a fixed implementation variant for all tasks. We then validate our methodology through experiments on real hardware, an SRC-6 reconfigurable computer.

Index Terms—Hardware Task Mapping, Genetic Algorithm, Reconfigurable Computing.

1 Introduction

Advances in reconfigurable computing technology have led to the development of high-performance reconfigurable computers (HPRC), comprised of commodity microprocessors coupled with reconfigurable hardware. Applications that are executed on HPRCs can therefore be accelerated by mapping significant parts of them onto reconfigurable hardware. With the availability of field-programmable gate arrays (FPGA) of high capacity, HPRCs show a great performance potential, with several orders of magnitude speedup possible over purely software implementations [1].

The architecture of a typical HPRC is shown in Fig. 1. The FPGA device, coupled to the microprocessor through a high-speed interconnect, serves as a reconfigurable co-processor. In order to exploit the performance potential of the coprocessor, an application needs to be carefully mapped to the system. For instance, when the application to be accelerated does not fit in the FPGA, the hardware mapping needs to be partitioned into multiple FPGA configurations, in a way that minimizes the total execution time. Partitioning gives rise to overheads, due to reconfiguration of the FPGA as well as transfer of intermediate data, as shown in Fig. 2 [∗].

Manuscript received Jan. 2011, revised July 2011.

M. Huang is with the Department of Computer Science and Computer Engineering, University of Arkansas, Fayetteville, AR, 72701, USA, email: mqhuang@uark.edu.
V. K. Narayana and T. El-Ghazawi are with the NSF Center for High-Performance Reconfigurable Computing (CHREC), Electrical and Computer Engineering Department, The George Washington University, Washington, DC 20052, USA, e-mail: {vikram,tarek}@gwu.edu.
M. Bakhouya and J. Gaber are with the Computer Science Department, Technical University of Belfort Montbeliard, 90010 Belfort Cedex, France, email: {mohamed.bakhouya,gaber}@utbm.fr

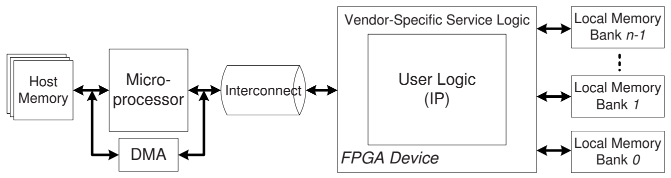


Fig. 1. General Architecture of a High-Performance Reconfigurable Computer

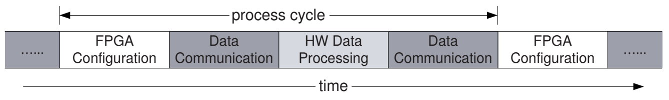


Fig. 2. Basic Execution Model for Hardware Tasks on an FPGA

The problem of mapping and partitioning an application has been studied by many researchers, and is generally approached by decomposing the application into its constituent tasks. These tasks might correspond to computation kernels and hardware modules that are part of a hardware implementation library for the particular HPRC platform [2]. After identifying the constituent tasks, a task graph is constructed to describe the data flow, which is then partitioned and scheduled into multiple configurations. For carrying out the required task scheduling, the usual practice is to consider the availability of one hardware implementation for each task. The parameters used for scheduling would include the logic resource consumed for each task (as a fraction of the total FPGA resource), as well as the processing throughput for that particular hardware implementation. Scheduling algorithms try to ensure that highly communicating tasks are retained within the same configuration as far as possible, in order to minimize overheads due to off-chip data transfer. In addition, the number of configurations is also reduced to the extent possible, in order to minimize the reconfiguration overhead.

An algorithm that takes care of both these aspects, i.e., data transfer and reconfiguration time, was proposed by us earlier as the Reduced Data Movement Scheduling (RDMS) algorithm [3]. This algorithm, similar to many other algorithms in the literature, considers only one architecture variant available for the hardware implementation of each task. If we incorporate multiple architectural variants for hardware tasks, it is possible to choose variants such that imbalances between the processing throughput of interacting tasks are minimized [4]. This presents an opportunity for optimization during the mapping process. We therefore propose a methodology for mapping task graphs with the added dimension of multiple variants, based on the use of a genetic algorithm. It will be demonstrated that by using

∗. In many cases the data communication and data processing can be overlapped in order to achieve high throughput. We intentionally distinguish these two times in this work for the sake of simplicity.

the proposed approach, a significant improvement in execution time can be achieved.

The rest of the paper is organized as follows. The next section gives an overview of related work, along with a background on the RDMS algorithm. Section 3 describes the proposed application mapping approach, starting with the problem description followed by the formulation of the genetic algorithm solution. Subsequently, the results are presented in Section 4. Finally, Section 5 concludes the paper.

2 Related Work

2.1 Multiple Task Variants and Genetic Algorithms

Use of multiple task variants for FPGAs has been considered earlier in the literature, although the applicability is different. For example, in [5], [6], multiple kernel/task variants are considered, to carry out a run-time binding of a kernel/task with an implementation variant. This is done more within the context of operating systems for reconfigurable systems or embedded systems, whereas the problem considered in this paper is the static mapping of task graphs containing dependence constraints. In [7], an application mapping approach termed as parallelism granularity selection (PARLGRAN) is proposed, to dynamically adjust the task granularity (the number of task instances) to balance the throughput among tasks. However, they consider it in the context of tasks chains, and apply it for the case when the device supports partial run-time reconfiguration.

Genetic algorithm (GA) has been used earlier by academic researchers for mapping task graphs to FPGAs [8], [9], [10]. In [8], task graphs are mapped to a single FPGA configuration composed of Xilinx MicroBlaze processors and hardware implementations of tasks. However, reconfiguration, multiple variants, and communication overheads are not considered. The work in [9] uses GA to map task graphs onto a partially reconfigurable FPGA, by modeling it as a tiled resource with multiple configuration controllers operating in parallel. Again, multiple task variants and communication overheads are ignored. In [10], the authors use a genetic algorithm for HW-SW partitioning in partially reconfigurable systems that execute multiple periodic task graphs. As part of the GA approach for determining the best partitioning, different individuals are evaluated by using the execution time and deadline violations obtained by using a modified list scheduling and placement algorithm. This work does not consider multiple variants of hardware tasks; nevertheless, it bears the most similarity to our approach, since we also use a scheduler as a subroutine in the genetic algorithm.

2.2 Background on RDMS Algorithm

Given the hardware task graph of one application and the hardware implementation of each task, the RDMS algorithm schedules the hardware tasks into a series of FPGA configurations in a way that minimizes the total hardware execution time, by restricting the number of FPGA configurations and the transfer of intermediate data between FPGA and host memory. In order to achieve these desired objectives, the RDMS algorithm takes three factors into account during the scheduling process, namely, (a) the task data dependency, (b) the hardware resource requirement of each task, and (c) the inter-task data communication. Details of the RDMS algorithm are given in [3].

3 Proposed Mapping Approach

3.1 Problem Statement

The application to be implemented on hardware is considered to be represented by a function-level directed acyclic graph (DAG), with tasks represented by the symbols F_i, $i \in [1, N]$ where N is the number of tasks in the task graph. A hardware library is available, comprising of hardware modules used for implementing the corresponding task nodes in the DAG. Within the hardware library, every hardware module consists of multiple implementations, reflecting trade-offs between resource requirement and performance. The implementation variant number j for a task F_i is represented by the symbol $H_{i,j}$. If there are J implementation variants for each task, then $j = 1, \ldots, J$.

Given a function-level DAG and a hardware library of the corresponding hardware modules, optimization techniques are needed to (1) select the proper implementation variant $H_{i,j}$ for each task F_i, which can be considered as a mapping between the hardware tasks and the available implementations, and (2) schedule the hardware tasks efficiently across multiple FPGA configurations $C_1, \ldots, C_k, \ldots, C_K$, to maximize the performance. Although step (2) can conceptually be carried out by using the RDMS algorithm, the two steps (1) and (2) need to be carried out together in order to obtain the best mapping. Since an exhaustive search for the suitable choice of implementation variant for each task is infeasible, we propose an approach based on genetic algorithm.

3.2 Genetic Algorithm - Overview and Formulation

Genetic algorithms [11] rely on starting with an initial population of randomly chosen solutions, which are successively refined through generations using crossover and mutation operations. Every individual in the population is represented using a bit string known as a chromosome. Each chromosome consists of genes. In our formulation, we choose to use a gene for each of the N tasks in the task graph; each gene represents the choice of a particular implementation variant for the task. For example, if there are J possible implementation variants for each task, then every gene would require $\lceil \log_2 J \rceil$ bits. Correspondingly, a chromosome consists of N genes, or $N\lceil \log_2 J \rceil$ bits. Every chromosome represents one possible selection of variants for all tasks. A

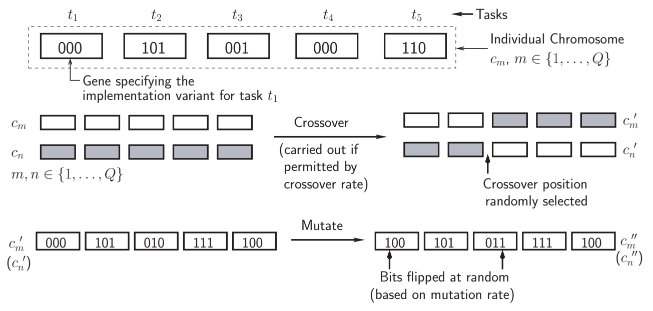

t_1 t_2 t_3 t_4 t_5
Tasks
000 101 001 000 110
Individual Chromosome
c_m, $m \in \{1, \ldots, Q\}$
Gene specifying the implementation variant for task t_1
c_m
c_n
$m, n \in \{1, \ldots, Q\}$
Crossover
(carried out if permitted by crossover rate)
c_m'
c_n'
Crossover position randomly selected
c_m' (c_n')
000 101 010 111 100
Mutate
100 101 011 111 100
c_m'' (c_n'')
Bits flipped at random (based on mutation rate)

Fig. 3. Chromosome Representation and Illustration of Crossover and Mutation Operations

```
Input: A hardware task graph, with each task $F_i$ having $J$
       implementation variants $H_{i,j}$, $j = 1, \ldots, J$.
Output: A reasonable initial target hardware execution time
        $T_{thwe}$
1 $T_{thwe} = \infty$;
2 for $j$ = 1 to $J$ do
3     Apply RDMS algorithm on the task graph in which task
      $F_i$ is mapped to implementation $H_{i,j}$, calculate $T_{hwe}$;
4     if $T_{thwe} > T_{hwe}$ then
5         $T_{thwe} = T_{hwe}$;
6 return $T_{thwe}$;
```

Fig. 5. Algorithm to Initialize Target Hardware Execution Time, T_{thwe}

```
Input: Random initial population of $Q$ chromosomes. Each
       chromosome contains a randomly selected
       implementation variant for every task in the
       application
Output: New population after genetic evolution. The best
        chromosome within this population is expected to be
        the desired mapping of the implementation variants
        for each task
1  repeat
2      For each chromosome in the population comprising the
       current generation, evaluate $T_{hwe}$ using RDMS
       algorithm;
3      Calculate the fitness of each chromosome using the
       fitness function in Eq (1);
4      repeat
5          Select two chromosomes from the current population
           based on fitness;
6          Crossover based on crossover rate, generate two
           offsprings;
7          Step through all the bits in the offsprings, flip them
           based on mutation rate;
8      until new generation created with Q chromosomes ;
9      Update $T_{thwe} \leftarrow$ Min exec time from all prev generations
10 until $K$ generations have been evaluated ;
```

Fig. 4. Genetic Algorithm for Task Mapping

sample chromosome is shown in Fig. 3, for $N = 5$ tasks (or, 5 genes) and $J = 8$ (3 bits per gene).

With this encoding of the scheduling problem, we use the genetic algorithm shown in Fig. 4. The algorithm begins with initializing a population with Q individuals having randomly assigned mappings, using Q chromosomes. Random initialization for task graph scheduling is normally used when the task dependencies are not encoded within the chromosome [12]–[14], which is the case here. Dependencies between tasks are taken care of by the RDMS algorithm during the evaluation of the fitness of the mapping. T_{hwe} denotes the hardware execution time obtained using the RDMS algorithm.

Within each iteration of the algorithm, new chromosomes are generated through crossover and mutation, giving rise to a new generation that replaces the existing population. The crossover and mutation operations are depicted in Fig. 3. The process repeats until a termination criterion is reached, which in our case corresponds to an upper limit on the number of iterations.

The selection of chromosomes for crossover and mutation is based on their fitness, evaluated in Line 3 of the algorithm. Since the GA tries to maximize the fitness, we use a reciprocal function of the execution time as part of the fitness function, similar to [12]:

$$\text{fitness}^{(i)} = \begin{cases} \frac{1}{T_{hwe}^{(i)} - T_{thwe}}, & \text{if } T_{hwe}^{(i)} > T_{thwe}, \\ \kappa & \text{otherwise.} \end{cases} \tag{1}$$

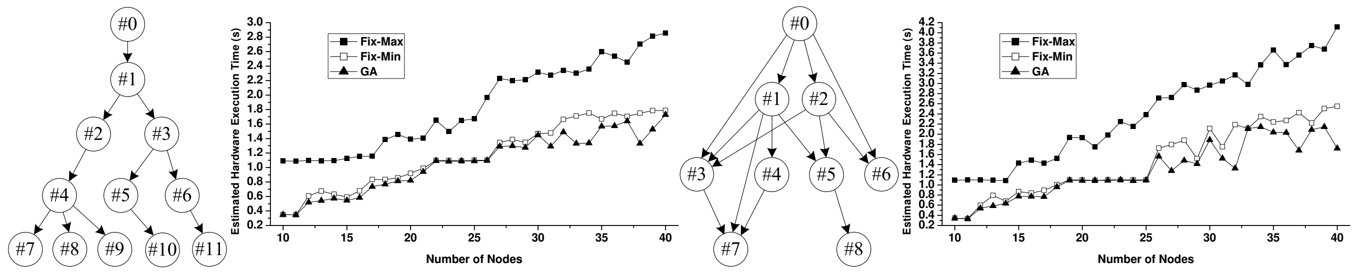
where *fitness*$^{(i)}$ is the fitness of chromosome i, and $T_{hwe}^{(i)}$ is the execution time of the task graph using the task variants based on the genes in chromosome i. The parameter T_{thwe} is a constant within an iteration (or generation), and needs to be initialized to a reasonable value. Based on the fact that the minimum execution time will at most be equal to that for the case when all tasks have the same implementation variant, an algorithm to choose T_{thwe} is given in Fig. 5. As mentioned earlier, selection of chromosomes for crossover is based on the fitness function (Line 5 of Fig. 4). We adopt Roulette wheel selection method for choosing the chromosome [15].

4 Results

4.1 Random Graph Simulation

In order to demonstrate the efficiency of GA-based mapping approach on a broad range of applications, we first apply it on randomly generated data flow graphs, which are of two types. The first type of graphs consists of outtrees, such as one shown in Fig. 6(a). Each node basically has one to three child nodes, and node 0 (which is the virtual node in RDMS) is only connected to one node. The second type consists of general random graphs, with dependencies across multiple levels. We refer to them as the cross-level graphs, Fig. 6(c). Each node level in the task graph consists of 1 to 5 nodes, and in addition to each node having one dependency on a node in previous level, there are 0 to 2 dependencies on nodes in any of the previous levels.

Given the task count, the graph is randomly generated in the sense that the task dependencies are randomly generated, and the resource requirements and corresponding processing times of the implementation variants are chosen as shown in Table 1, to test the algorithm for a wide range of values. For each generated task graph, the proposed approach is used for obtaining



(a) Out-Tree (b) Out-Tree Simulation (c) Cross-Level DAG (d) Cross-Level DAG Simulation

Fig. 6. Simulation of Randomly Generated Graphs (Node #0: Imaginary Root Node)

TABLE 1
The Range of Resource Requirement and Processing Time of Task Variants Used in Random Graph Simulation

Task Variant	Resource Requirement (%)	Processing Time (ms)
imp_1	25–50	0–20
imp_2	12.5–25	50–60
imp_3	6.25–12.5	200–250
imp_4	0–6.25	950–1000

TABLE 2
The Characteristics of Three RC Platforms

	SGI RC100	SRC-6	Cray XD1
FPGA Device	XC4VLX200	XC2V6000	XC2VP50
Number of Slices	89,088	33,792	23,616
Full Configuration Time*	966 ms	60 ms	1,824 ms
Interconnect Bandwidth	2.1 GB/s	1.4 GB/s	1.4 GB/s

∗. Device configuration time + vendor-introduced overhead

the appropriate task mappings. We also obtain mappings by choosing the same variant for every task (“fixed mapping”), which gives J possible fixed mappings if there are J variants for each task. The mapping with the maximum execution time among the fixed mappings is called ‘Fix-Max’, and the mapping with the minimum execution time among the J fixed mapping solutions is termed as ‘Fix-Min’. We compare Fix-Max, Fix-Min and the proposed GA-based approach for different task counts, using $J = 4$.

The task count for the two types of graphs is varied from 10 to 40 and the corresponding simulation results are shown in Fig. 6(b) and 6(d), respectively. Under all different scenarios, GA-based strategy is able to find the mapping that yields a hardware execution time which is equal to or smaller than the minimum hardware execution time using Fixed Mapping. Furthermore, the gap between the results of GA and Fixed Mapping becomes wider as the task count increases, particularly when the number of tasks is greater than 25.

4.2 *N*-Body Simulation Application

The proposed GA-based mapping approach is considered here for the N-body simulation application using the parameters from three real reconfigurable computers [1], Table 2. Details of the N-body application and its 18-node task graph are given in [3]. For testing the mapping algorithm, we set the parameter $\mathcal{N} = 16,000$, which means each node in the task graph will process 1,600,000 particles or data items.

4.2.1 Testbed and Hardware Library Setup

The N-body simulation task graph is mapped to three different RC platforms, SGI RC100, SRC-6, and Cray XD1 [1]. are used for obtaining the task mappings that are appropriate for the platforms. These parameters consist of the FPGA logic resources, full configuration time, and the sustained interconnect bandwidth, listed in Table 2. In this work, we consider only homogeneous logic resources/slices in the FPGA.

In this work, we assume that each module in the architectural variants hardware library consists of four implementations for each task, *imp_i*, i=1,. . . ,4. Among the four implementation variants, *imp_1* is the high performance version, which also consumes the most logic resources. The terminology “fully pipelined version” is used for referring to this variant. The other variants occupy less resources in decreasing order from *imp_2* to *imp_4*, with a corresponding reduction in the computation speed compared with the fully pipelined variant.

Primitive operators such as adder/subtractor, multiplier and divider, are used to construct the functionality of nodes in N-body task graph. In general, multiple primitive operators are needed to build a function node. For instance, the function node #11 needs 1 adder, 4 multipliers and 1 divider. Many researchers have reported various floating-point arithmetic designs on FPGA devices [16]–[19]. The resource requirement of double-precision (64-bit) floating-point operators of different variants based on the available literature is listed in Table 3. This includes the parameters of the various implementation variants, the exception being adder/subtractor and multiplier. For the adder/subtractor and multiplier operators, *imp_2*, *imp_3* and *imp_4* are assumed to occupy 50%, 25% and 12.5% resource of *imp_1*, respectively. If we assume that the design of both adder/subtractor and multiplier follows Karatsuba approach [20], the

TABLE 3
Resource Requirement (on SRC-6) & Processing Time of Different Variants of Double-precision (64-bit) Floating-point Operators

Operator	Resource Utilization(%)[*]				Processing Time (ms)[†]			
	imp_1	*imp_2*	*imp_3*	*imp_4*	*imp_1*	*imp_2*	*imp_3*	*imp_4*
$+/-$[‡]	5.71 [16]	2.85	1.43	0.71	16.0	32.0	64.0	128.0
$\times$[§]	7.26 [17]	3.63	1.81	0.91	16.0	64.0	256.0	1024.0
$\div$	10.17[¶]	9.03[¶]	5.94[¶]	0.99[¶]	16.0	20.0	493.8	975.6
$\sqrt{\ }$	9.40[¶]	4.99[¶]	3.03[¶]	1.41[¶]	16.0	29.1	507.9	1134.8

$*$. 15% of slices in device are assumed to be reserved for vendor service logic;
Resource usage on SGI RC100 = Resource usage on SRC-6 $\times$ 0.3793;
Resource usage on Cray XD1 = Resource usage on SRC-6 $\times$ 1.4309.
†. The time to perform 1.6 M operations under 100 MHz frequency.
‡. Processing time increases 2 times with half the resource usage.
§. Processing time increases 4 times with half the resource usage.
¶. Based on [18]

performance of the adder/subtractor and the multiplier will drop to half and one fourth, respectively, if the logic resource is reduced to half of the original value.

The processing time for each of the primitive operators in Table 3 is obtained as follows. The *imp_1* versions of all operators are fully pipelined, and process one input every clock cycle. Since the number of data items processed by each node (and therefore each operator within a node) is 1,600,000, the time take to process it at a clock rate of 100 MHz is 16ms. The maximum clock frequency is limited to 100 MHz in the reconfigurable platforms considered, and slower clocks may be derived internally within the FPGA by using their internal digital clock managers. For the task variants other than the fully pipelined versions, the processing time is appropriately scaled up based on available literature and as explained earlier for the adders and multipliers.

Table 4 lists the resource requirement of four implementation variants of each node composed of the primitive operators characterized by Table 3. For obtaining the parameters listed in the table, *imp_i* version of a node is taken to consist of only the *imp_i* variants of its constituent operators. The motivation behind this choice is the fact that if a slow version of one of the constituent operators is used, then there is limited benefit (and logic resource penalty) in using faster variants for the other constituent operators within the node. The processing time for each implementation variant listed in Table 4 is subject to the slowest constituent operator, which takes the longest processing time.

4.2.2 Experimental Setup and Results

The proposed approach uses the RDMS algorithm as part of the genetic algorithm procedure. The "task configuration time" is a fictitious number used in RDMS, and is basically a fraction of the full configuration time based on the percentage of FPGA occupied by the task implementation [3]. The task configuration time for SRC-6 is listed in Table 5. The numbers for the other platforms are not listed here for brevity, but may be deduced based on the node resource occupancy for the platform (Table 4) and the full configuration time (Table 2).

The off-chip data transfer time for each of the tasks is listed in Table 6. As noted earlier in Section 2.2, the on-chip data transfer between tasks within a configuration is not considered by the RDMS algorithm. The values listed in Table 6 for the three platforms are based on the interconnect bandwidth (Table 2), number of bytes transferred for each particle over the task graph edge considered and the number of particles that are transferred over the edge in the graph (1,600,000).

While obtaining results using the proposed approach, the chromosome population of the genetic algorithm in Fig. 4 is set to $Q = 100$, with $K = 200$ generations ; crossover and mutation rates are respectively taken to be 0.7 and 0.05, which seem to give good results for the task graphs we have used. Since the number of implementation variants for each task is $J = 4$, two bits are used per gene, or 36 bits per chromosome (since the

TABLE 4
Resource Requirement & Processing Time of Task Variants

Node[*] No.	Constituent Operators[§]	Resource Utilization (%)[†]				Processing Time (ms)[‡]			
		imp_1	*imp_2*	*imp_3*	*imp_4*	*imp_1*	*imp_2*	*imp_3*	*imp_4*
1,2	3A	17.13	8.56	4.28	2.14	16.0	32.0	64.0	128.0
3,4,5,6	1A	5.71	2.85	1.43	0.71	16.0	32.0	64.0	128.0
7	1M,1D	17.42	12.66	7.75	1.90	16.0	64.0	493.8	1024.0
8,9	2A,3M	33.20	16.60	8.30	4.15	16.0	64.0	256.0	1024.0
10	1D	10.17	9.03	5.94	0.99	16.0	20.0	493.8	975.6
11,15	1A,4M,1D	44.91	26.41	14.62	5.33	16.0	64.0	493.8	1024.0
12	1S	9.40	4.99	3.03	1.41	16.0	29.1	507.9	1134.8
13	4M	29.04	14.52	7.26	3.63	16.0	64.0	256.0	1024.0
14	1M	7.26	3.63	1.81	0.91	16.0	64.0	256.0	1024.0
16	3A,4M,1D	56.33	32.12	17.48	6.76	16.0	64.0	493.8	1024.0
17	2A,2M	25.94	12.97	6.48	3.24	16.0	64.0	256.0	1024.0
18	3A,3M	38.91	19.45	9.73	4.86	16.0	64.0	256.0	1024.0

$*$. For task graph node numbering, please refer [3]
†. The listed resource utilization is on SRC-6 Platform;
Resource usage on SGI RC100 = Resource usage on SRC-6 $\times$ 0.3793;
Resource usage on Cray XD1 = Resource usage on SRC-6 $\times$ 1.4309.
‡. The time to perform 1.6 M operations under 100 MHz frequency.
§. A: adder/subtractor, M: multiplier, D: divider, S: square root.

TABLE 5
Configuration Time of Task Variants on SRC-6

Node No.	Configuration Time[*] (ms)				Node No.	Configuration Time (ms)			
	imp_1	*imp_2*	*imp_3*	*imp_4*		*imp_1*	*imp_2*	*imp_3*	*imp_4*
1,2	10.28	5.14	2.57	1.28	12	5.64	2.99	1.82	0.85
3,4,5,6	3.43	1.71	0.86	0.43	13	17.42	8.71	4.36	2.18
7	10.45	7.60	4.65	1.14	14	4.36	2.18	1.09	0.54
8,9	19.92	9.96	4.98	2.49	16	33.80	19.27	10.49	4.06
10	6.10	5.42	3.56	0.59	17	15.56	7.78	3.89	1.95
11,15	26.95	15.84	8.77	3.20	18	23.34	11.67	5.84	2.92

$*$. The task configuration time is a fictitious number as explained in the text, and is essentially a measure of the resource consumption of the task variant. For node numbers, please see [3].

TABLE 6
Off-chip Communication Time on Three Platforms (ms)

S. N.*	D. N.*	Platform		S. N.	D. N.	Platform	
		SGI RC100	SRC/Cray			SGI RC100	SRC/Cray
0	1	18.29	27.43	6	15	6.10	9.14
0	2	18.29	27.43	7	17	6.10	9.14
0	3	6.10	9.14	8	15	6.10	9.14
0	4	6.10	9.14	8	11	6.10	9.14
0	5	6.10	9.14	9	11	6.10	9.14
0	6	6.10	9.14	9	12	6.10	9.14
0	7	12.19	18.29	10	14	6.10	9.14
0	17	6.10	9.14	10	13	6.10	9.14
1	8	18.29	27.43	11	15	6.10	9.14
2	18	18.29	27.43	12	14	6.10	9.14
2	8	18.29	27.43	13	16	6.10	9.14
2	9	18.29	27.43	14	16	6.10	9.14
3	11	6.10	9.14	15	17	6.10	9.14
3	10	6.10	9.14	16	17	6.10	9.14
4	11	6.10	9.14	17	18	6.10	9.14
5	15	6.10	9.14				

∗. S. N.: Source Node; D. N.: Destination Node. For node numbers, please refer [3]

number of tasks is $N = 18$). Crossover and mutation rates are empirically determined to give the best performance, as is the normal practice [13], [21]. In [13], the effect of different rates for scheduling task graphs on parallel homogeneous processor systems is studied by experimentally varying the rate. They determine the best crossover rate to be 0.7. In [10] also, a crossover rate of 0.7 is used for HW-SW partitioning of task graphs. It turns out that for our case also, the value of 0.7 gives better results than other values. The best mutation rate determined in [13] is 0.3, which is also the value used in [10]. However, for our case the mutation rate needs to be very low, since the mutation mechanism is very disruptive because it could potentially flip every bit in the chromosome. We therefore use a mutation rate of 0.05, which works well for the task graphs we used. The choice of crossover and mutation rates is a trade-off between *exploration* and *exploitation* [22]. Lower rates will help in *exploiting* the good individuals within a population in order to transmit the good traits to the next generation. On the other hand, a higher rate prevents the algorithm from being caught in local minima, by allowing *exploration* of a larger area in the search space.

Fig. 7 show the progress of the estimated hardware execution time T_{hwe}^{min} during the iterations (or generations) of the genetic algorithm. The figures show it for the three different platforms considered, using $K = 100$ generations.

- On SRC-6 platform, the genetic algorithm groups the 18 tasks into 3 configurations and reduces the inter-configuration communication overhead and hardware processing time to the minimum. The implementation versions, for all the 18 tasks in the final mapping result, are listed in Table 7.

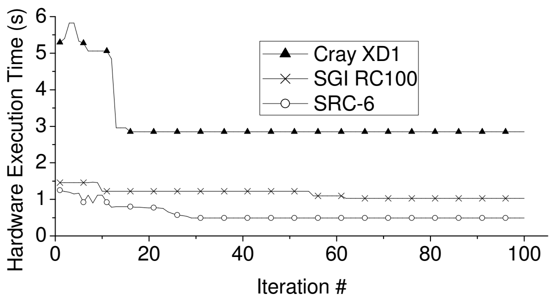


Fig. 7. The Simulation of Hardware Execution Time

TABLE 7
Selection of Hardware Implementation Variant for Each Task using Genetic Algorithm

Platform	Hardware Task No.																	
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*
SGI RC100	3	3	1	1	2	1	2	1	2	2	2	2	2	2	2	2	2	2
SRC-6	3	3	3	3	2	3	1	2	2	1	2	2	1	1	2	2	1	1
Cray XD1	4	4	4	3	4	2	4	3	4	4	4	3	4	2	4	4	4	4
Cray Y*	3	3	1	3	3	1	1	2	2	1	2	2	2	1	2	2	1	1

∗. Assuming the full configuration time is 100 ms. Hardware task numbers correspond to the N-body task graph in [3].

- On both SGI RC100 and Cray XD1 platforms, the optimal mapping is achieved by grouping all hardware tasks into one single FPGA configuration, by using the smaller implementation variants, as listed in Table 8. The use of a single configuration helps due to the large reconfiguration overhead, particularly for Cray XD1; a reduction in the execution time by avoiding one extra configuration offsets the increase in execution time using slower (and smaller) task variants. If we assume there exists a new platform Cray Y, which is same as Cray XD1 in all respects except that the full configuration time is 100 ms, it is observed that the chosen mapping takes 4 configurations and *imp_4* (the smallest variant) is not selected for any task, as shown in Table 7 and Table 8.

Table 8 compares the GA-based proposed approach against the fixed mapping choices. Since there are four variants for each task, there are four fixed mapping choices, as noted in Section 3.2. From the table, it is observed that the proposed approach can give a significant improvement over the naive approach of choosing the same variant for all tasks. Compared with the slowest fixed mapping choice, the proposed algorithm gives an improvement of up to 78.6% in the total execution time.

The genetic algorithm has been executed on a Linux box with Intel Xeon 2.8 GHz and 8 GB main memory. The time to finish the first 100 iterations of genetic algorithm is approximately 40 seconds for all cases and mainly contributed by the *Dynamic Programming* used in RDMS algorithm.

TABLE 8
Comparison between Fixed Implementation and Genetic Algorithm on Three Platforms

Platform	Number of Configurations					Hardware Execution Time (s)				
	imp_1	*imp_2*	*imp_3*	*imp_4*	**GA**	*imp_1*	*imp_2*	*imp_3*	*imp_4*	**GA**
SGI	2	1	1	1	**1**	2.086	1.030	1.474	2.101	**1.030**
SRC-6	5	3	2	1	**3**	0.709	0.573	1.250	1.195	**0.489**
Cray	7	4	2	1	**1**	13.300	7.790	4.833	2.959	**2.848**
Cray Y	7	4	2	1	**4**	1.178	0.912	1.385	1.235	**0.809**

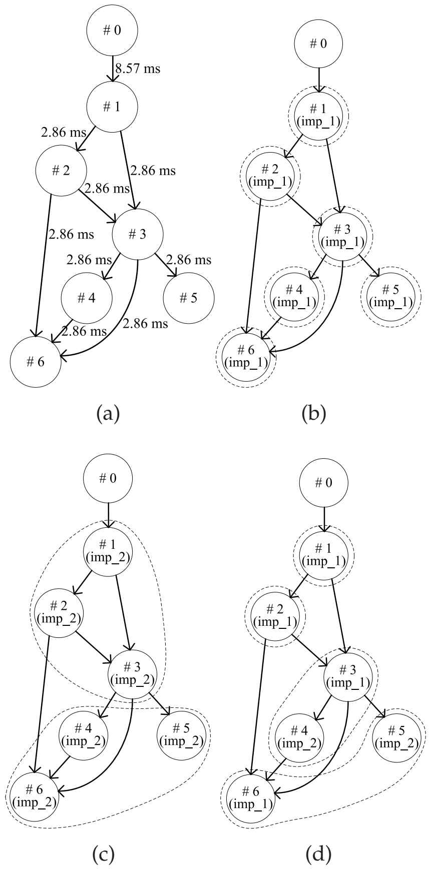


Fig. 8. The Implementation of a Synthetic Graph on SRC-6 (a) The Original Graph. (b) Implementation with Fast Variants. (c) Implementation with Slow Variants. (d) Implementation with GA Mapping (Node #0: Imaginary Root Node).

4.3 Synthetic Graph Experiments on SRC-6

Some experiments are carried out to emulate a synthetic task graph on the SRC-6 platform. The graph that we have used is shown in Fig. 8. The constituent nodes are taken to be comprised of primitive double-precision floating point operators. Two variants of each of the operators are considered available, with characteristics listed in Table 9. The processing time for the faster variant is 5 ms for all operators, for 500,000 data items processed at 100 MHz. The processing time for the slower variant is appropriately scaled, similar to the *N*-body example. Since we do not have the actual variant implementations available, we emulate the actual implementation. Rather than checking the functionality, the basic idea is to verify the proposed approach based on actual data transfers and reconfiguration of the FPGA.

TABLE 9
Two Implementation Variants of Operators on SRC-6

Operator	Implementation Variant 1		Implementation Variant 2	
	Proc. Time	Resource	Proc. Time	Resource
$+/-^*$	5 ms	5.71% [16]	10 ms	2.85%
$\times^\dagger$	5 ms	7.26% [17]	20 ms	3.63%
$\div$	5 ms	10.17% [18]	155 ms	5.94% [18]
$\sqrt{\ }$	5 ms	9.40% [18]	160 ms	3.03% [18]

$*$. Processing time increases 2 times with half the resource usage.
†. Processing time increases 4 times with half the resource usage.

TABLE 10
Two Implementation Variants of Tasks on SRC-6

Node No.	Operator Combination*	Implementation 1		Implementation 2	
		Proc. Time	Resource	Proc. Time	Resource
1,6	4A, 3M, 0D, 2S	5 ms	63.42%	160 ms	28.36%
2	2A, 3M, 0D, 3S	5 ms	61.40%	160 ms	25.67%
3	4A, 5M, 1D, 0S	5 ms	69.30%	155 ms	35.50%
4	4A, 3M, 0D, 0S	5 ms	44.62%	20 ms	22.31%
5	8A, 3M, 0D, 0S	5 ms	67.45%	20 ms	33.73%

$*$. A: adder/subtractor, M: multiplier, D: divider, S: square root.

The input data size is 500,000 double-precision words of 8 bytes each. Since the bandwidth of interconnect on SRC-6 is 1.4 GB/s, the estimated data transfer time is 2.86 ms. Therefore all the edges in Fig. 8 have values that are multiples of 2.86 ms, depending on the number of inputs required for each node. Note that the edge values depicted in the figure correspond to data transfer between host memory and local memory, for inter-configuration data transfer. Data transfer between tasks residing within the same configuration is effectively hidden (Section 2.2). RDMS needs to use an edge value in Fig. 8 only when that edge connects two tasks in different FPGA configurations.

Each task node in the graph shown in Fig. 8 is composed of primitive operators as listed in the second column of Table 10. Again, two variants are considered for each of the nodes, based on two variants of the primitive operators. The fast implementation, *imp_1*, is comprised of only fast primitives and *imp_2* consists of only slow primitive operators. The resource of each task is simply the summation of the resource of its component operators. The operators inside each node operate in a pipelined fashion; therefore, the theoretical throughput of each task is same as that of the slowest operator in the node. The processing time and resource requirement of the two implementation variants of the 6 task nodes are listed in Table 10.

Three different strategies are used for emulating the task graph of Fig. 8(a) on SRC-6. First, we use a fixed mapping choice of only fast variants for all tasks. Second, we use the fixed mapping choice of slow variants. We

TABLE 11
Implementation Result for Synthetic Graph On SRC-6

Number of Configurations			Hardware Execution Time (s)					
			Estimated			Measured		
imp_1	*imp_2*	**GA**	*imp_1*	*imp_2*	**GA**	*imp_1*	*imp_2*	**GA**
6	2	**4**	0.436	0.463	**0.330**	0.471	0.483	**0.363**

compare these two approaches with the proposed GA-based approach, with the results summarized in Fig. 8 and Table 11. As the results show, the proposed approaches gives the best execution time, and it achieves it by using a larger number of FPGA configurations compared with the use of the slow variants (Fig 8(c)). The measured hardware execution time for the three experiments is also listed in Table 11, which is close to the estimated value, thus validating our approach.

5 Conclusion

In this paper, a new approach for mapping task graphs to reconfigurable hardware is explored, based on the availability of multiple implementation variants of each hardware task in the application. The mapping of task instances to the appropriate variant depends not only on the FPGA capacity and reconfiguration time, but also on the interaction between tasks in the task graph. The methodology proposed in this paper makes use of a genetic algorithm for obtaining the best mappings. The proposed approach can be used with any scheduling algorithm under the hood, and we have chosen RDMS algorithm because it takes care of data dependencies. Results using simulations for random tasks graphs as well as a real application, in addition to synthetic graph experiments on a real reconfigurable platform, demonstrated the effectiveness of the proposed approach.

Acknowledgment

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. IIP-0706352. The authors are grateful to Dr. Esam El-Araby and Dr. Harald Simmler for the helpful discussion.

References

[1] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell, "The promise of high-performance reconfigurable computing," *IEEE Computer*, vol. 41, no. 2, pp. 78–85, Feb. 2008.

[2] P. Saha, E. El-Araby, M. Huang, M. Taher, S. Lopez-Buedo, T. El-Ghazawi, C. Shu, K. Gaj, A. Michalski, and D. Buell, "Portable library development for reconfigurable computing systems: A case study," *Parallel Computing*, vol. 34, no. 4+5, pp. 245–260, May 2008.

[3] M. Huang, V. K. Narayana, H. Simmler, O. Serres, and T. El-Ghazawi, "Reconfiguration and communication-aware task scheduling for high-performance reconfigurable computing," *ACM Trans. Reconf. Technol. Syst.*, vol. 3, no. 4, pp. 20:1–20:25, Nov. 2010.

[4] M. Huang, V. K. Narayana, and T. El-Ghazawi, "Efficient mapping of hardware tasks on reconfigurable computers using libraries of architecture variants," in *Proc. the Seventeenth Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'09)*, Apr. 2009, pp. 247–250.

[5] W. Fu and K. Compton, "An execution environment for reconfigurable computing," in *Proc. 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2005 (FCCM 2005)*, Apr. 2005, pp. 149–158.

[6] T. Lee, J. Henkel, and W. Wolf, "Dynamic runtime re-scheduling allowing multiple implementations of a task for platform-based designs," in *Proc. 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE'02)*, Mar. 2002, pp. 296–301.

[7] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Exploiting application data-parallelism on dynamically reconfigurable architectures: Placement and architectural considerations," *IEEE Trans. VLSI Syst.*, vol. 17, no. 2, pp. 234–247, Feb. 2009.

[8] J. Wang and S. M. Loo, "Case study of finite resource optimization in fpga using genetic algorithm," in *GEC '09: Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2009, pp. 989–992.

[9] Y. Qu, J.-P. Soininen, and J. Nurmi, "A genetic algorithm for scheduling tasks onto dynamically reconfigurable hardware," in *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, 2007, pp. 161–164.

[10] B. Mei, P. Schaumont, and S. Vernalde, "A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems," in *Proc. 11th ProRISC workshop on Circuits, Systems and Signal Processing*, Nov. 2000.

[11] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic Programming : An Introduction On the Automatic Evolution of Computer Programs and Its Applications*. San Francisco, CA: Morgan Kaufmann, 1998.

[12] A. Y. Zomaya, R. C. Lee, and S. Olariu, *An introduction to genetic-based scheduling in parallel processor systems*. John Wiley and Sons, New York, USA, 2001, ch. 5, Solutions to Parallel and Distributed Computing Problems, pp. 111–133.

[13] A. Y. Zomaya, C. Ward, and B. Macey, "Genetic scheduling for parallel processor systems: Comparative studies and performance issues," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 8, pp. 795–812, 1999.

[14] B. Mei, P. Schaumont, and S. Vernalde, "A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems," in *ProRisc Workshop on Circuits, Systems and Signal Processing*, Nov 2000.

[15] A. Orriols-Puig, K. Sastry, P. L. Lanzi, D. E. Goldberg, and E. Bernadó-Mansilla, "Modeling selection pressure in xcs for proportionate and tournament selection," in *Proc. the 9th annual conference on Genetic and evolutionary computation (GECCO '07)*, July 2007, pp. 1846–1853.

[16] G. Govindu, R. Scrofano, and V. K. Prasanna, "A library of parameterizable floating-point cores for FPGAs and their application to scientific computing," in *Proc. The International Conference on Engineering Reconfigurable Systems and Algorithms (ERSA'05)*, June 2005, pp. 137–145.

[17] L. Zhuo and V. K. Prasanna, "Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 4, pp. 433–448, Apr. 2007.

[18] A. J. Thakkar and A. Ejnioui, "Design and implementation of double precision floating point division and square root on FPGAs," in *Proc. IEEE Aerospace 2006*, Mar. 2006.

[19] K. S. Hemmert and K. D. Underwood, "Open source high performance floating-point modules," in *Proc. the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, Apr. 2006, pp. 349–350.

[20] A. A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," *Doklady Akad. Nauk SSSR*, vol. 145, pp. 293–294, 1962.

[21] O. Sinnen, L. A. Sousa, and F. E. Sandnes, "Toward a realistic task scheduling model," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 3, pp. 263–275, 2006.

[22] A. Y. Zomaya and Y.-H. Teh, "Observations on using genetic algorithms for dynamic load-balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 9, pp. 899–911, 2001.