A Framework for Dynamic Resource Assignment and Scheduling on Reconfigurable Mixed-Mode On-Chip Multiprocessors

Xiaofang Wang and Sotirios G. Ziavras
Department of Electrical and Computer Engineering
New Jersey Institute of Technology
Newark, NJ 07102, USA
{xw23, ziavras}@njit.edu

Abstract

Encouraged by continuous advances in FPGA technologies, we explore high-performance Multi-Processor-on-a-Programmable-Chip (MPoPC) reconfigurable architectures. This paper proposes a methodology for assigning resources at run time and scheduling large-scale floating-point, data-parallel applications on our mixed-mode HERA MPoPC. HERA stands for HEterogeneous Reconfigurable Architecture. An application is represented by a novel mixed-mode task flow graph which is scheduled to run under a variety of independent or cooperating parallel computing modes: SIMD (Single-Instruction, Multiple-Data), Multiple-SIMD and MIMD (Multiple-Instruction, Multiple-Data). The reconfigurable logic is customized at static time and reconfigured at run time to match application characteristics. An in-house developed parallel power flow analysis code by Newton's method is employed to verify the methodology and evaluate the performance. This application is of utmost importance to any power grid.

1. Introduction

The recent capabilities of state-of-the-art FPGAs in terms of resources and speed provide new opportunities in reconfigurable computing. The peak performance of individual floating-point (FP) operations on FPGAs has outnumbered in the last few years that of modern microprocessors and is growing faster than the latter [1]. Recent relevant work [2-4] further confirms and strengthens this trend. It is viable now for FPGAs to accommodate complex high-performance applications. Encouraged by continuous advances in FPGA technologies, we explore high-performance cluster-on-a-chip (i.e., MPoPC) reconfigurable architectures for large-scale computation-greedy, data-parallel applications. These applications often involve complex matrix-based algorithms where software programmability and standard FP representation are indispensable.

*This work was supported in part by the U.S. Department of Energy under grant DE-FG02-03CH11171.

Scalability and portability are also essential to performance due to the variant size of matrices and the ever changing parameters of various applications. Compiling tools for FPGAs targeting spatial computing from high-level descriptions, such as C/C++, Java and Matlab [e.g., 9-13], are emerging to deal with challenges in multi-million-gate FPGA designs. FPGA-based systems have enjoyed great success in accelerating many computation-intensive applications due to their superior resource utilization [5-6]. However, most often reconfigurable computing systems are *application-specific programmable circuits* (ASPCs) tailored to a particular algorithm. Therefore, they are developer programmable rather than user programmable. If the design size overflows the resources, which is often the case for matrix-based applications, device reprogramming is required; it usually takes tens to one hundred milliseconds and may have an adverse effect on the execution time. This issue becomes more serious with increases in the chip size [7] because the size of configuration data is proportional to the total number of on-chip resources. Also, the time-consuming hardware design and implementation procedure has to be repeated every time a change is made to the source code.

Our solution is based on user programmable MPoPCs that support partial runtime reconfiguration. In addition to alleviating the above difficulties, MPoPCs can offer several advantages for our target applications. Medium- to coarse-grain modular architectures (e.g., XPP [8]) are simpler and more scalable than complex custom designs. Hence, MPoPCs can potentially achieve higher system frequencies than the latter. This feature also reduces the verification time, which is a major part of current deep-submicron designs. The communication latencies in MPoPCs can be very low. They also have lower cost and design risk compared to their ASIC peers. Another benefit is that we can leverage tremendous research results in computer architecture and parallel processing, and also incorporate reconfiguration. More importantly, we can customize the hardware architecture based on application characteristics and, hence, achieve a higher utilization of hardware resources compared to systems with general-purpose microprocessors or ASICs.

This paper proposes a framework for mapping, scheduling and executing applications with large data sets on HERA [22], a mixed-mode MPoPC that we have designed and implemented on Xilinx Virtex II FPGAs. HERA can be reconfigured at run time to support a variety of independent or cooperating parallel computing modes, such as SIMD, Multiple-SIMD and MIMD to match the unique requirements of applications. Not only does our approach exploit mixed-mode parallelism but we also put a major effort on effective resource management at both static and run time. A mixed-mode *Task Flow Graph* is constructed for a given application; tasks are classified according to their appropriate computing mode. A *parameterized hardware component library* (PHCL) is adopted to customize individual components in HERA configurations based on task requirements. During the execution of an application, we reallocate on-the-fly the required number of processors with appropriately reconfigured units to tasks based on the system status. An in-house developed parallel power flow analysis code for Newton's method [19] is employed to verify the methodology and evaluate the performance. Real-time solutions to this problem are of paramount importance to any power grid [18].

The remainder of the paper begins with an introduction of the key architectural features in HERA. Section 3 presents an overview of our framework. Section 4 focuses on task profiling. The architecture synthesis and reconfiguration scheme are presented in Section 5. Section 6 is devoted to the integrated dynamic resource scheduling and task execution process. The application study can be found in Section 7 and the conclusions in Section 8.

2. HERA: a Mixed-Mode MPoPC

HERA targets matrix-based applications with large data sets. The general organization of our HERA machine with *m* x *n* PEs interconnected via a 2-D mesh network is shown in Figure 1. We employ fast, direct NEWS (North, East, West and South) connections for communications between nearest neighbors. Every column has a *Cbus* and all the *Cbuses* are connected to the *Column Bus* for broadcasting SIMD instructions and their data. The computing fabric is controlled by the system *Sequencer* that communicates with the host processor via the PCI bus for data I/O. The *Global Control Unit* (GCU), included in the system *Sequencer*, fetches instructions from the *global program memory* (GPM) for PEs operating in SIMD. A PE of HERA is an in-house designed RISC processor built around a 7-stage pipelined FPU with dual-port *local data memory* (LDM) and *local program memory* (LPM). The sizes of LPM and LDM are also determined during the architecture synthesis stage. A novel feature of HERA's memory hierarchy is that the B port of every PE is shared with the neighbors to the south and east. A PE can directly write to or read from the LDMs of its west and north neighbors via their B ports. This feature can be used to eliminate omnipresent large block data transfers between nearest neighbors in numerous block-based matrix computations (e.g., LU factorization). HERA's modular instruction set contains about 30 instructions classified in six major groups: integer arithmetic, FP arithmetic, memory access, jump and branch, PE NEWS communications and system control. The operating mode of each PE can be configured dynamically by the host processor through its *Operation Mode Register* by using a *Configure* instruction. Combined with the PE ID number and masks, the system can be configured dynamically into a mixed-mode computing system capable of supporting simultaneously SIMD, MIMD and multiple-SIMD.

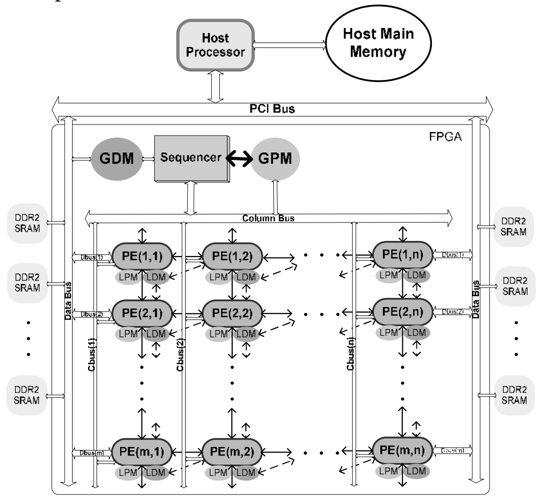

Figure 1. HERA organization.

3. Framework Overview

Assume a matrix-based FP application and a programmable chip that supports run-time reconfiguration. The latter has the following available resources: Ψ (logic resources expressed in logic cells) and ζ (total memory capacity expressed in bits), Our objective is to maximize resource utilization in the reconfigurable logic and potentially minimize the total execution time of the overall application, $T_{total} = \sum_{i=1}^{n} T(i)$, where $T(i)$ is the execution time of task S_i (all tasks in the application are considered). Figure 2 shows the general architecture of our framework for heterogeneous MPoPCs like HERA. There are five major phases in this framework: *task profiling*, *system synthesis*, *task coding* using the target system's instruction set, *system implementation* on FPGAs, and *dynamic, simultaneous resource-efficient task decomposition*, *mapping, scheduling and execution.* The implementation on FPGAs follows the same procedure as any VHDL-based design methodology. Due to limited space, we focus on task profiling, system synthesis, and dynamic resource management and mapping, scheduling and execution.

4. Task Characterization

A typical data-parallel FP application in engineering and science consists of blocks of nested loops which consume most of the overall execution time; these loops are controlled by conditional statements. In our framework, the behavioral description of the application is first analyzed to construct a *Task Flow Graph* (TFG), $G = (S, D)$; it is a *weighted, directed acyclic graph* (wDAG). S and D represent the sets of nodes and edges, respectively. Figure 3 shows a typical TFG. Each node in this graph represents a task $S_i \in S$, where $i \in [1, n]$ is inclusive of all the tasks. There are two types of tasks: SIMD tasks (denoted by circles) and MIMD tasks (denoted by octagons). Associated with each task S_i are its computing mode (SIMD/MIMD), FP computation number and FP operation types, represented by $\varepsilon(S_i)$, $FP(S_i)$ and $\pi(S_i)$, respectively. The memory requirements (in bits) of each task are represented by two parameters, $\sigma_c(S_i)$ and $\sigma_d(S_i)$, for the instruction memory and data memory, respectively. A directed edge between two tasks S_i and S_j represents a data dependence between them and its weight $D_{i,j} \in D$ represents the volume of data in bits that goes from task S_i to S_j. An *entry task* is defined as a node with no incoming edges (e.g., S_1 in Figure 3) and an *exit task* is defined as a node with no outgoing edges (e.g., S_8). The selection of an optimal mode for each task can be a complex procedure and is not the focus of this paper. An SIMD task in our study is typically a data-parallel block (e.g., a nested loop) which can benefit from synchronous execution under SIMD, while an MIMD task is more of the control-flow style.

5. Architecture Synthesis and Reconfiguration

In our framework, dynamic reconfiguration of computing resources for a given application is employed since FP computing cores are very resource expensive. Only the required FP operations are supported each time by PEs. The functionality of PEs can be changed via hardware reconfiguration at run time, as needed by the tasks.

5.1. Parameterized Hardware Component Library (PHCL)

PHCL plays a major role in our methodology. The performance of the included components, in terms of speed and resource requirements, is manipulated in our approach. All the components are designed and compiled in VHDL. The major parameterized components for our matrix-based applications include:

- Variable precision pipelined FP function cores (including IEEE-754 single- and double-precision implementations, and other non-IEEE custom formats**)**. Figure 4 shows the block diagram of an FP core whose major parameters are listed in Table 1. The cores are parameterized by the mantissa and exponent sizes. Different choices for the mantissa and exponent lead to different data ranges and resource requirements. Since FP cores are the main consumers of logic resources and embedded DSP cores in FPGAs, it is very important to choose the appropriate precision for the function cores.
- HERA architecture templates.
- Memory blocks, including single- and dual-port memories.

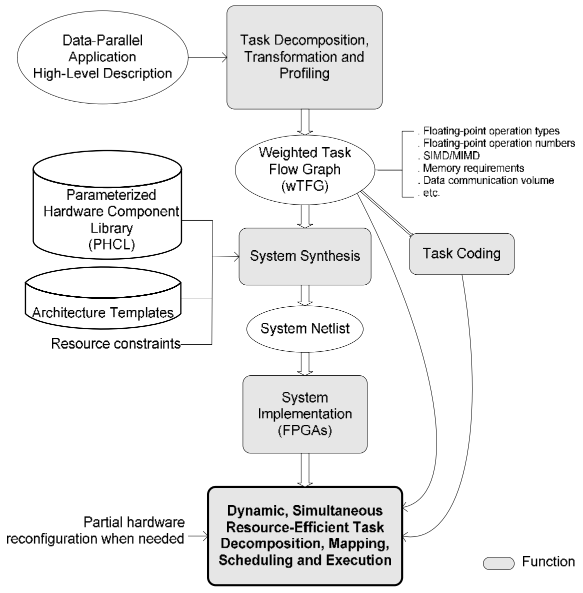
Data-Parallel Application High-Level Description
Task Decomposition, Transformation and Profiling
Weighted Task Flow Graph (wTFG)
. Floating-point operation types
. Floating-point operation numbers
. SIMD/MIMD
. Memory requirements
. Data communication volume
. etc.
Parameterized Hardware Component Library (PHCL)
Architecture Templates
Resource constraints
System Synthesis
Task Coding
System Netlist
System Implementation (FPGAs)
Partial hardware reconfiguration when needed
Dynamic, Simultaneous Resource-Efficient Task Decomposition, Mapping, Scheduling and Execution
Function

Figure 2. Framework overview/flowchart.

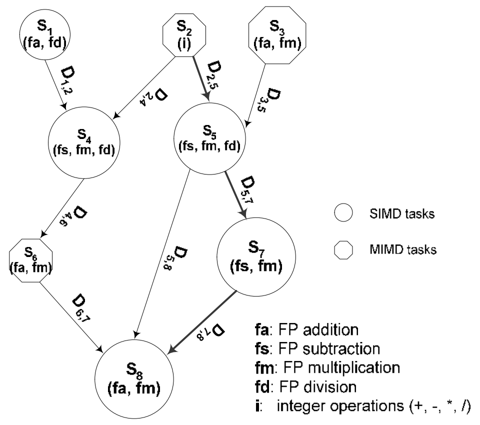

Figure 3. A typical task flow graph.

- Custom function units, such as trigonometric function implementers.
- Various registers.
- Integer function cores parameterized by word size.

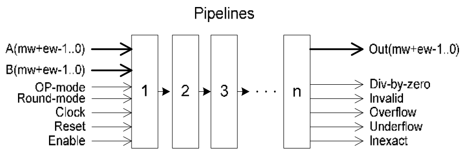

Figure 4. Block diagram of an FP core.

Table 1. PHCL parameters of the FP core in Figure 4

Function	FP division
Mantissa (bits)	24
Exponent (bits)	8
Latency (cycles)	27
Frequency(MHz)	189
Logic resources (slices)	979
Embedded DSP blocks	None
Memory blocks (BRAM)	None
Target device	Virtex II XC2V6000-5

5.2. System Synthesis

The architecture generator takes a TFG and generates the initial architecture based on the computation types and amounts in tasks. A system template in VHDL is selected first from PHCL, with the basic PE interconnection and interface to the sequencer. We assume that the requirements of this template for logic resources and memory blocks are known as $L_{sys}(p)$ in logic cells and $M_{sys}(p)$ in bits, for a given number of PEs p. The PE datapath (functionalities and width), total number of PEs and layout of PEs are customizable. The system generator first analyzes the data range of the application and decides on an appropriate precision for the FP cores. The total number of PEs is largely determined by this choice. Only the required function cores are included in PEs. It is possible that all the operation types are required throughout execution but only one or two operations are needed for a few tasks. For example, FP division is less frequent than multiplication and addition in many data-parallel applications. Based on our implementation results, a single-precision IEEE-754 FP divider is at least more than two or five times larger in space than an FP adder or multiplier, respectively. These numbers are even larger for a double-precision FP divider. For less frequently used operations, we consider the following cases. An FP divider is used as an example.

A. Large tasks appear in the critical path.

A critical path in a TFG is a group of dependent tasks forming a linear array that comprises an entry task and an exit task, and has the largest number of FP operations and communication volume among all paths. It potentially determines the execution time of the entire application. Several algorithms exist in the literature to find such a path [14]. For example, among all paths, $S_1 \rightarrow S_3 \rightarrow S_5$ forms the critical path in the TFG of Figure 5.a; also an FP divider is required by S_3 which is a large task based on its number of FP operations. In this case, an FP divider is initiated for all PEs at the very beginning.

B. Small tasks appear in the critical path.

S_5 in Figure 5.b is such an example. Because S_3 is the task that potentially contributes the most to the execution time, the priority is to maximize the number of PEs for S_3 with the inclusion of an FP adder and a multiplier as well. None of the PEs contains an FP divider until the execution of S_5. Some PEs will be reconfigured to add an FP divider when the time comes for S_5.

C. Tasks are not in the critical path.

This case is treated similarly to *Case B.* For example, an FP divider is added to some PEs at the time needed to accommodate task S_4 shown in Figure 5.c.

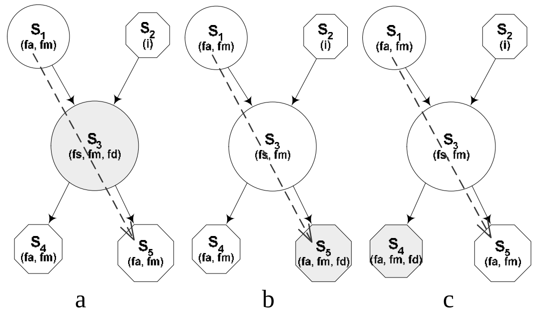

Figure 5. Example of PE function selection.

These decisions are based on our adaptive parallelization technique and the device reconfiguration latencies. We could make a more accurate decision by only adding an FP divider to some PEs just when needed. However, the device reconfiguration time of modern FPGAs is still significant and there is no sign that it will be reduced significantly in the near future. It may have an adverse effect if we either reprogram fully the chip or a large number of PEs required by large tasks. Thus, we are mainly interested in the partial runtime reconfiguration (PRTR) of FPGAs when no computation can be scheduled on some PEs while the remaining PEs are still busy with assigned tasks.

Let p be the total number of PEs that can be implemented in the system and PE$_i$ requires $L(i)$ logic resources. Also, the capacity of the instruction and data memories of each PE$_i$ is $M_c(i)$ and $M_d(i)$, respectively. The following equations represent the objective functions in the recursive minimization procedure of our proposed system synthesis.

$$L_{sys} + \sum_{k=1}^{p} L(k) < \Psi$$

$$M_{sys} + \sum_{k=1}^{p} \{M_c(k) + M_d(k)\} < \zeta$$

6. Dynamic Resource Assignment and Task Scheduling

This section discusses our *Critical-Path driven maximum Resource Utilization dynamic Scheduling* (CP-maxRUS) heuristic. Based on various application scenarios, system architectures and performance objectives, extensive scheduling research targeting multiprocessors has been done for fixed parallel architectures [14-15]. Scheduling is an NP-complete problem and a good heuristic for suboptimal performance should be the goal. There are more key features that distinguish our strategy from existing approaches. First, we target a real reconfigurable system under various resource constraints. Second, we explore adaptive parallelization of tasks; i.e., we determine the appropriate number of PEs and the binding of tasks to PEs at run time instead of making static decisions. Third, our adaptive parallelization is PE-oriented rather than task-oriented.

6.1. Loop Partitioning

Loop-based partitioning is the basis of our adaptive parallelization. We restrict our effort to assigning each time the complete or part of an iteration to a PE. Hence, flow dependence is allowed inside an assigned iteration.

A. FOR loops without cross-iteration dependence

Assume that the total number of PEs available to the loop is α and the total number of iterations is *L*. The loop space is split into α groups of size $\lfloor L/\alpha \rfloor$ or $\lceil L/\alpha \rceil$. Each PE gets such a group and the corresponding data set. These loops map naturally to the SIMD mode and no communication is required. *FOR* loops without both flow dependence and cross-iteration dependence are treated the same way.

B. FOR loops with cross-iteration dependence [16]

We assume that the distance between successive data dependent iterations is *w*. Let the total iteration space *L* in the loop be a multiple of *w* and the loop can be divided into *w* partitions. The i^{th} partition contains the iterations $l_0 + i + k*w$, where $k \in [0, L/w-1]$ and l_0 is the starting point of the loop index, for $i \in [0, w-1]$. Each partition is then further divided into α groups of size $\lfloor L/(\alpha * w) \rfloor$ or $\lceil L/(\alpha * w) \rceil + 1$ of continuous iterations. Each PE gets such a group and the corresponding data set. By distributing data this way, data communication is restricted between two neighboring groups.

C. Heterogeneous Loops

Each iteration in heterogeneous loops has different FP operations. Let f_i be the number of FP operations in the i^{th} iteration and $f_i = a*i + b$. Such a loop can be transformed into a homogeneous loop by combining the i^{th} and $(n-i-1)^{th}$ iterations into a new loop with a constant number of operations [17]. Then the above partitioning techniques can be applied to these loops.

Other loops that can be transformed into *FOR* loops (e.g., *WHILE* loops) are treated similarly.

6.2. PE Searching

The order in which we search for candidate PEs is an important step affecting the overall mapping performance. Based on HERA's organization and interconnection network, we propose our *PULSE* column-biased search process that uses the pattern ↓↑↓↑↓ since the distance between two adjacent stops in this search pattern is always one. Another reason is one port of the data memory of each PE is shared with its immediate neighbors to the west and south.

Assume that the numbers of PEs assigned concurrently to tasks S_i and S_j are p_i and p_j, respectively, and $x = min\{p_i, p_j\}$. The objective function to be minimized in this step is the communication time among tasks $T_{com} = \sum_{i,j=1}^{n} \max\{[D_{ij}/(\lambda * x) + T_{ini}] * H(i,j) + T_{cflit}\}$, where D_{ij} is the amount of data communicated between these two groups of PEs, λ is the transfer speed in bits/second between two immediate neighbors, T_{ini} is the overhead to initialize the transfer, *H(i, j)* is the number of hops between two communicating PEs and T_{cflict} is the routing delay caused by data collisions. In order to reduce the collision and communication costs, data locality is taken into account when mapping tasks to PEs.

6.3. Dynamic CP-maxRUS Strategy

At this point, the target system is modeled as an undirected graph *GT* = (*P*, *L*), where a vertex $P_i \in P$ represents PE_i and an undirected link L_{ij} represents a bidirectional communication channel between PE_i and PE_j, for $i, j \in [1, p]$. Each PE_i is associated with a parameter $\upsilon(PE_i)$ that records its functionality and a parameter $cm(PE_i)$ that represents its current computing mode. The weight $w(L_{ij})$ on each L_{ij} denotes its minimum communication cost. The minimum communication cost is calculated based on the minimum hops between PE_i and PE_j. Also, communication jobs always have higher priority than computation jobs (i.e., PEs are always forced to forward data to their neighbors even when they are busy). A priority is assigned to each task in TFG; it changes dynamically as scheduling proceeds. This is because the dynamic assignment of PEs, the dynamic partitioning and migration of tasks to multiple PEs, and the communication pattern and cost in our policy result in changes in the critical path. If two tasks are assigned to the same PE, then the communication is removed. In the *CP-maxRUS* procedure, a task is said to be *READY* when all its inputs are available. A *QUALIFIED* PE for a task is defined as a PE that supports all the operation types $\pi(S_i)$ in the task S_i. We define the *average execution time* (AET) for each task S_i on the p_i processor as $AET(S_i, p_i) = (O_i / p_i) + T_c(S_i, p_i)$, where O_i is the remaining FP operation numbers of S_i, $T_c(S_i, p_i)$ is the communication overhead caused by distributing task S_i to p_i PEs compared to scheduling on one PE. The *CP-maxRUS* procedure is shown in Figure 6.

```
1.  WHILE $Pt \neq \emptyset$ and $S \neq \emptyset$ DO  // Pt is the set of available PEs
2.  {
3.      Find out $CP_1, CP_2, CP_3$, …, until all tasks are included;
4.      Calculate the priority of each task; sort S in the decreasing
        order of priority;
5.      Pick the top task $S_i \in S$;
6.      IF $\varepsilon(S_i)$ = MIMD and READY THEN
7.        { $\forall PE_i \in Pt, i=1,\dots, pt$,
8.          Search for the PEs that hold the inputs of $S_i$;
9.          IF ( $p_i \neq 0$) THEN      // $p_i$ : qualified PEs in Pt for $S_i$
10.           {Find q PEs in these $p_i$ PEs with the minimum $AET(S_i, q)$ and assign these q PEs to $S_i$;
11.            $\forall PE_i, i=1,\dots, q$, set $cm(PE_i)=MIMD$;}
12.         ELSE
13.         {Reconfigure one or more PEs into a qualified PE for $S_i$;
14.          Assign this PE to $S_i$;
15.          Set $cm(PE)=MIMD$; }
16.          Delete $S_i$ from S; }
17.     ELSEIF $\varepsilon(S_i)$ = MIMD and not READY THEN
18.       {  Pick the next task in S;
19.          GOTO 6;}
20.     ELSEIF $\varepsilon(S_i)$ = SIMD and READY THEN
21.       {  Assign all qualified PEs $\in Pt$ to $S_i$;
22.          $\forall$ qualified $PE_i \in Pt$, Set $cm(PE_i)=SIMD$;
23.          Delete $S_i$ from S; }
24.     ELSEIF $\varepsilon(S_i)$ = SIMD and not READY THEN
25.       {  $\forall S_k, k \in [1, m]$, that hold the inputs of $S_i$,
          Add the qualified PEs $\in Pt$ to these m tasks and
           repartition the tasks so as to
              $AET'(S_1, p_1)=AET'(S_2, p_2)=\dots=AET'(S_k, p_k)$,
           where $AET'(S_k, p_k) = (O'_k / p_k) + T_c(p_k)$, and $O'_k$ is the
           remaining number of operations for task $S_k$ and $k \in [1, m]$;
           Set $cm(PE_k) = \varepsilon(S_k)$, if $PE_k$ is assigned to process $S_k$ ;}
26.        WHILE (not all the m tasks are finished)        //Some
                other PEs may become idle during waiting
27.           {Reconfigure unqualified PEs for $S_i$ into qualified
                PEs; // Partial hardware reconfiguration
28.            GOTO 25;  }
29.       }
30.  }
```

Figure 6. Dynamic scheduling algorithm.

7. Application Study

We illustrate our framework with computation-intensive data-parallel power flow analysis. It requires evaluation of the following equations:

$$P_i = \sum_{k=1}^{N} | y_{ik} V_i V_k | \cos(\theta_{ik} + \delta_k - \delta_i) \qquad (1)$$

$$Q_i = \sum_{k=1}^{N} | y_{ik} V_i V_k | \sin(\delta_i - \delta_k - \theta_{ik}) \qquad (2)$$

where P_i, Q_i and V_i, are the active power, reactive power and complex voltage at bus i, respectively, with
$V_i = |V_i| \angle \delta_i$, $V_k = |V_k| \angle \delta_k$, $y_{ik} = |y_{ik}| \angle \theta_{ik} = g_{ik} + jb_{ik}$, for $i, k \in [1, N]$; y_{ik} is an element of the bus admittance matrix (Y_{bus} matrix) [18]. Newton's method [19] is a robust solution especially for large power networks consisting of thousands of nodes (buses). In this method, the following linear equations are solved iteratively until the mismatches $\triangle\delta$ and $\triangle V$ are smaller than a pre-specified tolerance:

$$\begin{bmatrix} J^{11} & J^{12} \\ J^{21} & J^{22} \end{bmatrix} \begin{bmatrix} \triangle\delta \\ \dfrac{\triangle V}{|V|} \end{bmatrix} = \begin{bmatrix} \triangle P \\ \triangle Q \end{bmatrix} \qquad (3)$$

The Jacobian matrix $J = \{J^{11}, J^{12}, J^{21}, J^{22}\}$ is reevaluated in each iteration. Due to the limited space in this paper, the equations for calculating the blocks in the Jacobian matrix are omitted here and can be found in [18]. Eqs. (3) are normally solved by LU factorization, which factors the Jacobian matrix into the lower- and upper-triangular matrices L and U. Due to the cubic computation complexity of this process and difficulties in parallelization, the real-time solution by Newton's method has always been a great challenge to computer systems. Since the Jacobian is a very sparse matrix, an innovative partition strategy can rearrange it into the Doubly-Bordered Block Diagonal (DBBD) form shown in Figure 7; the entire application can then be solved efficiently in parallel. The relevant tasks are shown in Table 2 at the end of the paper.

IEEE-754 single-precision units were chosen based on the two benchmark matrices in Table 3; Table 4 lists their performance after place-and-route for the XC2V6000-5 FPGA on the Annapolis WILDSTAR II-PCI board that we used in this experiment. The board contains twelve 2MB DDR2 SRAM chips and two FPGA devices. We set the system frequency at 80MHz. From the task table, we can see that tasks $S_5^i(k_i)$, $S_6^i(k_i)$ and $S_7^i(k_i)$ consume most of the execution time, especially for large matrices. The available number of PEs to these tasks has a large impact on the entire performance. We compared the performance of three cases during execution: no device reconfiguration (NR), full runtime reconfiguration (FRTR) and PRTR. Figure 8 shows the numbers of PEs during execution for the 300-bus system. PRTR provides the largest number of PEs for all tasks, and NR has a fixed and the least number of PEs for all tasks. In this application, FRTR also performs well in terms of available PEs. However, we must consider the cost of different schemes. Although we have a low number of PEs in NR, there is no reconfiguration overhead. The overhead is significant for FRTR, especially when dealing with a large number of reconfigurations during execution. We have four reconfigurations in Figure 8 for FRTR. Also, FRTR prevents the overlapping of tasks and some resources are wasted.

In PRTR, reconfiguration overlaps computation as much as possible and tasks can be overlapped as well, so the overhead is minimized. The execution times for the two benchmark matrices are presented in Table 5. Due to the lack of efficient support for PRTR on our board, the results with PRTR are based on VHDL simulation, where actual implementation data on the board are used for the execution time of tasks in order to compare with the actual measurements of the FRTR and NR schemes. The reconfiguration time of the entire device we are using is about 50ms at 50MHz [20]. The PRTR time is estimated by the percentage of reconfigured resources. In future work, a CAD tool based on JBits APIs can be used to implement PRTR [21]. *Data I/O* refers to the data transfer between the on-board SRAM chips and the on-chip memory of PEs. For the 300-bus case, the device reconfiguration overhead for FRTR is more than the benefit brought about by more PEs; also, FRTR uses more time than NR. With an increase in the matrix size, hence the amount of computation in large tasks also increases and the benefit of resource reconfiguration is important as a result of increased PE numbers, as shown in the 1648-bus case. PRTR performs the best for both benchmarks.

$$\left\{\begin{matrix} J_{11} & 0 & 0 & 0 & J_{1n} \\ 0 & J_{22} & 0 & 0 & J_{2n} \\ 0 & 0 & J_{33} & 0 & J_{3n} \\ 0 & 0 & 0 & \dots & \dots \\ J_{n1} & J_{n2} & J_{n3} & \dots & J_{nn} \end{matrix}\right\} = \left\{\begin{matrix} L_{11} & 0 & 0 & 0 & 0 \\ 0 & L_{22} & 0 & 0 & 0 \\ 0 & 0 & L_{33} & 0 & 0 \\ 0 & 0 & 0 & \dots & \dots \\ L_{n1} & L_{n2} & L_{n3} & \dots & L_{nn} \end{matrix}\right\} x \left\{\begin{matrix} U_{11} & 0 & 0 & 0 & U_{1n} \\ 0 & U_{22} & 0 & 0 & U_{2n} \\ 0 & 0 & U_{33} & 0 & U_{3n} \\ 0 & 0 & 0 & \dots & \dots \\ 0 & 0 & 0 & \dots & U_{nn} \end{matrix}\right\}$$

where

$$J_{kk} = L_{kk}U_{kk}$$

$$U_{kn} = L_{kk}^{-1}J_{kn}$$

$$L_{nk} = J_{nk}U_{kk}^{-1},\ \ for\ k \in [1, n-1]$$

$$L_{nn}U_{nn} = J_{nn} - \sum_{k=1}^{n-1} L_{nk}U_{kn}$$

Figure 7. LU factorization of the Jacobian matrix.

Table 4. Performance of single-precision PHCL modules

Function unit	Area (Slices)	Freq. (MHz)	Latency (Cycles)	Embedded DSP blocks
Add/Sub	404	160.1	3	0
Multiplier	95	182.3	3	4
Division	875	186.6	27	0
Square root	721	210	27	0

Table 5. Execution times for the benchmark matrices (sec)

Case		NR	FRTR	PRTR
300-bus	1	0.911	0.802	0.738
	2	0.017	0.025	0.022
	3	0	0.20	0.156
	4	0.928	1.027	0.916
1648-bus	1	13.70	11.72	10.99
	2	0.66	1.03	0.80
	3	0	0.20	0.17
	4	14.36	12.95	11.96

1: Computation; 2: Data I/O; 3: Reconfiguration; 4: Total

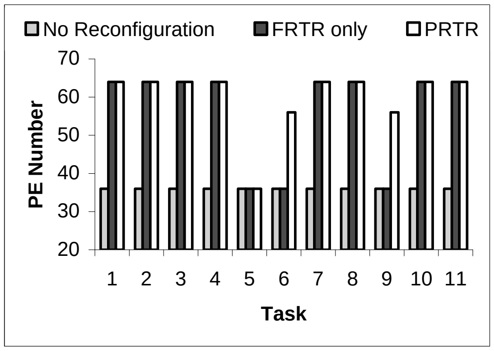

Figure 8. The parallelism profile (i.e., numbers of PEs) during the execution for the 300-bus system.

8. Conclusions

Continuous advances in FPGA technology offer new opportunities in high-performance reconfigurable computing targeting large-scale floating-point applications. MPoPCs offer general-purpose instructions that can reduce hardware reprogramming drastically during the execution of large tasks. They are platforms capable of entering the mainstream of parallel computing at very low cost and small design risk. We have presented our framework for mapping and scheduling data-parallel applications as related to our HERA mixed-mode reconfigurable MPoPC. By customizing and dynamically reconfiguring the MPoPC according to the characteristics of the subtasks in a given application, HERA achieves high utilization of the reconfigurable logic and yields good performance. Our power flow analysis benchmarks testify to this extent.

9. References

[1] K. Underwood, “FPGAs vs. CPUs: Trends in Peak Floating-Point Performance,” *12th ACM/SIGDA FPGA*, 2004, pp.171-180.

[2] L. Zhuo and V. K. Prasanna, “Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on FPGAs,” *18th International Parallel and Distributed Processing Symposium (IPDPS)*, 2004, pp. 92-101.

[3] J. Liang, R. Tessier and O. Mencer, “Floating Point Unit Generation and Evaluation for FPGAs,” *IEEE FCCM*, April 2003, pp. 185–194.

[4] C. H. Ho, M. P. Leong, P. H. W. Leong, J. Becker and M. Glesner, “Rapid Prototyping of FPGA Based Floating Point DSP Systems,” *IEEE International Workshop on Rapid System Prototyping*, July 2002, pp. 19-24.

[5] K. Compton and S. Hauck, “Reconfigurable Computing: A Survey of Systems and Software,” *ACM Comput. Surveys*, Vol. 34, No.2, 2002.

[6] A. DeHon, “The Density Advantage of Configurable Computing,” *IEEE Computer*, April 2000, Vol. 33, No. 4, pp. 41-49.

[7] J.H. Pan, T. Mitra and W.F. Wong, "Configuration Bitstream Compression for Dynamically Reconfigurable FPGAs," *International Conference on Computer Aided Design 2004 (ICCAD).* Nov. 2004, pp. 766-773.

[8] J. Becker and M. Vorbach, "Architecture, Memory and Interface Technology Integration of an Industrial/ Academic Configurable System-on-Chip (CSoC)," *IEEE Symposium on VLSI*, 2003, pp. 107-112.

[9] J. Ou and V. K. Prasanna, "PyGen: A MATLAB/Simulink Based Tool for Synthesizing Parameterized and Energy Efficient Designs Using FPGAs," *IEEE FCCM*, 2004.

[10] B. Hutchings and B. Nelson, "Developing and Debugging FPGA Applications in Hardware with JHDL," *33rd Asilomar Conference on Signals, Systems, and Computers,* Oct. 1999, Vol. 1, pp. 554-558.

[11] D. I. Lehn, R. D. Hudson and P. M. Athanas, "Framework for Architecture-Independent Run-Time Reconfigurable Applications," *SPIE Proceedings*, 2000.

[12] Y. Yi, R. Woods and J. V. McCanny, "Hierarchical Synthesis of Complex DSP Functions on FPGAs," *37th Asilomar Conference on Signals, Systems and Computers*, Nov. 2003, Vol. 2, pp. 1421-1425.

[13] W. Luk, N. Shirazi and P. Y. K. Cheung, "Compilation Tools for Run-time Reconfigurable Designs," *IEEE FCCM*, April 1997, pp. 56–65.

[14] M. Pinedo, *Scheduling: Theory, Algorithms, and Systems,* 2nd Edition, Prentice Hall, 2002.

[15] C. L. McCreary, A. A. Khan, J. J. Thompson and M. E. McArdle, "A Comparison of Heuristics for Scheduling DAGs on Multiprocessors," *8th International Parallel Processing Symposium*, 1994, pp. 446-451.

[16] M. Cierniak, W. Li and M. J. Zaki, "Loop Scheduling for Heterogeneity," *4th IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, 1995, pp. 78-85.

[17] Ding-Kai Chen, J. Torrellas and Pen-Chung Yew, "An Efficient Algorithm for the Run-Time Parallelization of DOACROSS Loops," *Supercomputing*, Nov. 1994, pp. 518–527.

[18] J. J. Grainger and W. D. Stevenson Jr, *Power System Analysis*, McGraw Hill, 1994.

[19] W. F. Tinney and C. E. Hart, "Power Flow Solution by Newton's Method," *IEEE Trans. on Power Apparatus and Systems* Vol. PAS-86, No. 3, 1967, pp. 1146-1152.

[20] Xilinx Virtex-II Platform FPGA User Guide, http://direct.xilinx.com/bvdocs/userguides/ug002.pdf.

[21] D. Mesquita, F. Moraes, J. Palma, L. Moller and N. Calazans, "Remote and Partial Reconfiguration of FPGAs Tools and Trends," *10th Reconfigurable Architecture Workshop*, April 2003.

[22] X. Wang and S. G. Ziavras, "HERA: A Reconfigurable and Mixed-Mode Parallel Computing Engine on Platform FPGAs," *16th International Conference on Parallel and Distributed Computing and Systems (PDCS)*, Nov. 2004, pp. 374-379.

Table 2. Task information in the DBBD power flow algorithm

Tasks	Description	Mode	FP Op.
$S_1(1), \ldots, S_1(n)$	Evaluate Eqs (1) and (2) and calculate $\triangle P$ and $\triangle Q$	SIMD	+, -, * cos, sin
$S_2(1), \ldots, S_2(n)$	Check individual convergence	SIMD	None
S_3	Check global convergence	MIMD	None
$S_4(1), \ldots, S_4(n)$	Construct 3-block groups $\{J_{ii}, J_{in}$ and $J_{ni}\}$	SIMD	+, -, * cos, sin
$S_5^i(1), \ldots, S_5^i(k_i)$, $i = 1, \ldots, m$	LU factorization of 3-block groups of the approximate same sizes	SIMD	+, -, *, /
$S_6(1), \ldots, S_6(n)$	LU factorization of 3-block groups of diverse sizes	MIMD	+, -, *, /
$S_7^i(1), \ldots, S_7^i(k_i)$, $i = 1, \ldots, m$	Multiplication of factored border blocks in S_5	SIMD	+, *
$S_8(1), \ldots, S_8(n)$	Multiplication of factored border blocks in S_6	MIMD	+, *
S_9	Factor the last block J_{nn}	SIMD	+, -, *, /
$S_{10}(1), \ldots, S_{10}(n)$	Forward reduction	SIMD	+, -, *
$S_{11}(1), \ldots, S_{11}(n)$	Backward substitutions	MIMD	+, -, *

n: the total number of 3-block groups in the matrix *J*
m: the total number of task pools that have the approximate same computing cost; each group *i* contains k_i 3-block groups.
q: the total number of 3-block groups with diverse computing cost

$$n = q + \sum_{i=k_1}^{k_m} m$$

Table 3. Optimal partitioning of the Y_{bus} matrices for the benchmark systems

Benchmark System	IEEE 300-bus	1648-bus
Dimensionality of admittance matrix (Y_{bus})	300	1648
Dimensionality of Jacobian matrix (J)	530	2982
Maximum nodes in a block	16	120
Number of independent diagonal blocks	21	18
Minimum dimensionality of independent diagonal blocks	6	33
Maximum dimensionality of independent diagonal blocks	16	120
Dimensionality of the last block	42	134
Size distribution of independent diagonal blocks in Y*	5(9)*, 6(16), 15, 3(14), 2(12), 3(10), 6	120, 109, 99, 3(90) , 5(85), 79, 5(75), 33

*5(9) stands for 5 blocks of approximate size 9 x 9.