Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2015, Article ID 783237, 12 pages
http://dx.doi.org/10.1155/2015/783237

Research Article

Dynamic Task Distribution Model for On-Chip Reconfigurable High Speed Computing System

Mahendra Vucha[1] and Arvind Rajawat[2]

[1]*Faculty of Engineering, Christ University, Bangalore, Karnataka 560074, India*
[2]*Maulana Azad National Institute of Technology, Bhopal, Madhya Pradesh 462003, India*

Correspondence should be addressed to Mahendra Vucha; mahendra.vucha@christuniversity.in

Received 30 June 2015; Revised 1 September 2015; Accepted 4 November 2015

Academic Editor: Michael Hübner

Modern embedded systems are being modeled as Reconfigurable High Speed Computing System (RHSCS) where Reconfigurable Hardware, that is, Field Programmable Gate Array (FPGA), and softcore processors configured on FPGA act as computing elements. As system complexity increases, efficient task distribution methodologies are essential to obtain high performance. A dynamic task distribution methodology based on Minimum Laxity First (MLF) policy (DTD-MLF) distributes the tasks of an application dynamically onto RHSCS and utilizes available RHSCS resources effectively. The DTD-MLF methodology takes the advantage of runtime design parameters of an application represented as DAG and considers the attributes of tasks in DAG and computing resources to distribute the tasks of an application onto RHSCS. In this paper, we have described the DTD-MLF model and verified its effectiveness by distributing some of real life benchmark applications onto RHSCS configured on Virtex-5 FPGA device. Some benchmark applications are represented as DAG and are distributed to the resources of RHSCS based on DTD-MLF model. The performance of the MLF based dynamic task distribution methodology is compared with static task distribution methodology. The comparison shows that the dynamic task distribution model with MLF criteria outperforms the static task distribution techniques in terms of schedule length and effective utilization of available RHSCS resources.

1. Introduction

Microprocessors are at the core of high performance computing systems and they provide flexibility for wide range of applications at the expense of performance [1]. Application Specific Integrated Circuit (ASIC) supports fixed functionality and superior performance for an application but it restricts the architecture flexibility. Reconfigurable computing (RC) [2] promises greater flexibility without compromise in performance. The RC with Field Programmable Gate Array (FPGA) architecture brings the phenomenon of dynamic reconfiguration of custom digital circuits without physically altering the hardware to provide more flexible and low cost solution for real time applications. However the microprocessor acts as softcore processor that executes software tasks described in High Level Language (HLL) whereas the RC architecture FPGA acts as hardcore processor that reconfigures its hardware for the behaviour of hardware tasks described in Hardware Description Language (HDL). The partial reconfiguration behaviour of FPGA supports parallel tasking by partitioning its hardware into finite number of Reconfigurable Logic Units (RLUs) and each independent RLU reconfigures its hardware for the behaviour of hardware task. The dynamic reconfigurable phenomenon of RC brought higher performance for complex applications by reducing instruction fetch and decoding and executing bottleneck [1–3]. So, High Speed Computing System (HSCS) should have one or more resources of such RC architectures as processing element (PE) to enhance the speed of application execution. There are hybrid systems such as reconfigurable system on chip (RSoC) [4] and MOLEN architecture [3] in the literature that have integrated both microprocessor and FPGA to support software as well as hardware tasks in an application. This paper also targeted a hybrid system described on a single chip FPGA. The homogeneous computing systems having array of similar PEs [5, 6] provide parallel processing to

the distributed applications at the expense of number of resources whereas heterogeneous computing systems having array of dissimilar PEs [7–9] support distributed applications at the expense of dissimilar communication protocols (i.e., buses and bridges) between heterogeneous resources. The reconfigurable systems having FPGA as PE [10–13] bring phenomenon of dynamic reconfiguration to the applications at the lack of softcore processor or at the expense of off-chip softcore processor interface and efficiency. A hybrid computing platform called Reconfigurable High Speed Computing System [3, 4] (RHSCS) having integrated softcore PE (MicroBlaze) and hardcore PE (RLUs) configured on a single chip FPGA minimizes communication cost and also supports both software tasks and hardware tasks execution. The RHSCS can provide optimal intermediate computing platform for execution of software tasks and hardware tasks exist in distributed applications. So, the resources of RHSCS need to be shared among the tasks of an application and it is achieved in this research by designing an efficient dynamic task distribution methodology for RHSCS.

The remainder of the paper is organized as follows. The literature review is presented in Section 2, task distribution problem and strategies are in Section 3, proposed dynamic task distribution methodology is in Section 4, and experimental results and discussions are in Section 5, and the paper is concluded in Section 6.

2. Literature Review

This section brings the literature review of various task distribution methodologies for reconfigurable heterogeneous computing systems that have multiple dissimilar processing elements. A computing platform called MOLEN polymorphic processor described in [3] modeled with both general purpose and custom reconfigurable processing elements. The MOLEN processor is designed with arbitrary number of processing elements to support both hardware and software tasks. An efficient multitask scheduler [14] proposed for runtime reconfigurable system introduced a new parameter called *Time-Improvement* as cost function for compiler assisted scheduling models. The Time-Improvement parameter is the combination of reduction-in-task-execution time and distance-to-next-call. The efficient multitask scheduler [14] is demonstrated for the MOLEN polymorphic processor [3] environment where the control of tasks assigned to General Purpose Processor (GPP) and tasks execution is assigned to reconfigurable processing elements. The scheduler in [14] outperforms its contemporary algorithms and accelerates task execution by 4% to 20%. In [15], an online hybrid scheduling model is demonstrated for CPU-FPGA platform where tasks are represented in three categories such as software tasks (ST) executed only on CPU, hardware tasks (HT) executed only on FPGA, and hybrid tasks (HST) executed on both CPU and FPGA. The hybrid scheduling model [15] is the integration of task allocation, placement and task migration modules, and schedule of the tasks of an application based on their *reserved time*. An online HW/SW partitioning and coscheduling algorithm [16] is proposed for GPP and Reconfigurable Processing Unit (RPU) environment in which *Hardware Earliest Finish Time* (HEFT) and *Software Earliest Finish Time* (SEFT) are estimated for tasks of an application. The difference between HEFT and SEFT used to partition the tasks and then scheduled tasks list has been prepared based on EFT for GPP and RPU as well. The reconfigurable computing coscheduler (ReCoS) [17, 18] integrates the strengths of Hardware Computing (HC) and Reconfigurable Hardware (RH) scheduling policies in order to effectively handle the RC system constraints such as number of FFs, LUTs, multiplexers, CLBs, communication overheads, reconfiguration overheads, throughputs, and power constraints. Hardware supported task scheduling is proposed in [4] for dynamically Reconfigurable SoC that utilizes the resources effectively for execution of applications. The RSoC architecture comprises a general purpose embedded processor along with two L1 data and instruction caches and a few Reconfigurable Logic Units (RLUs) on a single chip. In [4], task systems are represented as Modified Directed Acyclic Graph (MDAG) and the MDAG defined as tuple $G = (V, E^d, E^c, P)$, where V is set of nodes, E^d and E^c are the set of directed data edges and control edges, respectively, and P represents the set of probabilities associated with E^c. The conclusion of the research [4] states that dynamic scheduling (DS) does not degrade as the complexity of the problem increases whereas the performance of Static Scheduling (SS) declines. The DS outperforms SS when both task system complexity and degree of dynamism increase. Compiler assisted runtime scheduler [19] is designed for MOLEN architecture where the runtime application is described as Configuration Call Graph (CCG). The CCG assigns two parameters called *distance to the next call* and *frequency of calls in future* to the tasks of an application and these parameters act as cost function to distribute the tasks. HW/SW codesign techniques have been demonstrated in [20] for dynamically reconfigurable architectures with the aim of deciding execution order of the tasks at runtime based on their EDF. In [20], the authors have demonstrated a HW/SW partitioning algorithm, a codesign methodology with dynamic scheduling for discrete event systems, and a dynamic reconfigurable computing multicontext scheduling algorithm. These codesign techniques [20] minimize application execution time by parallelizing tasks execution and the model is controlled by host processor for both shared memory and local memory based Dynamic Reconfigurable Logic (DRL) architectures. The coscheduling techniques in [20] brought better optimization for shared memory architecture over local memory architectures when the DRL cells are more than three. A HW/SW partitioning algorithm in [21] is presented for task partition as software tasks and hardware tasks based on their waiting time and resources availability. A methodology is proposed in [22] for building real time reconfigurable systems in order to ensure that all constraints of an application are met. In [22], Compulsory-Reuse (CR) and Loading-Back factor are estimated for tasks of an application to support reuse of resources. A deadline partitioning scheduler is proposed for scheduling dynamic hard real time task sets onto fully and partially reconfigurable systems [23] with the objective of reducing tasks rejection ratio. The scheduler in [23] computes

weight, defined as ratio of execution time to deadline of the task, as cost function to distribute randomly generated periodic task sets. An efficient task scheduler is proposed in [24] for heterogeneous computing systems based on EFT, level of the task, and MLF as cost functions. A case study and design challenges of various task distribution methodologies were presented in [25] for single and multiple processing element computing systems. In summary, the task distribution techniques developed for CPU-FPGA system accelerate the application execution whereas CPU is engaged for controlling tasks execution instead of executing tasks. Since CPU is utilized for controlling tasks execution, the task distribution models described in the literature may generate significant overheads when a task or application demands for software execution. The task distribution methodologies in the literature may also degrade the efficiency of an application execution due to communication overheads between off-chip CPU and FPGA. These issues were addressed in [26] by designing a task distribution model based on MLF distribution policy for a computing platform having softcore and hardcore processing elements on a single chip FPGA. In this paper, the methodology in [26] is described and presented for real life benchmark applications to evaluate the effectiveness of the task distribution methodology.

3. Task Distribution Problem and Strategies

The main objective of task distribution is to map a given application represented as Direct Acyclic Graph (DAG) to the resources of computing platform RHSCS to minimize total execution of the application while utilizing the resources effectively. This section defines strategies like task graph representation, targeted computing architecture, and overview of task distribution model and finally demonstrates the dynamic and static task distribution with an example.

3.1. Application as Task Graph. Applications can be represented as a Directed Acyclic Graph (DAG) $G = (V, E)$, where V represents set of N tasks $V = \{v_1, v_2, v_3, \ldots, v_N\}$ and E represents set of edges $E = \{e_{12}, e_{13}, \ldots, e_{21}, e_{23}, \ldots, e_{ij}, \ldots\}$ between the tasks. Each edge $e_{ij} \in E$ represents the precedence constraint such that task v_i should complete its execution before v_j. In a DAG, a task without any predecessor is an entry task and task without successor is an exit task. The tasks in DAG are weighted with the attributes like a_i task arrival time, d_i task deadline, w_i task area in terms of number of bit slices required, rc_i task reconfiguration time, he_i task execution time on RLU, and se_i task execution time on softcore processor, where $i = 1, 2, 3, \ldots, N$ and N is equal to number of tasks in DAG. The tasks in DAG are executed on the reconfigurable computing platform RHSCS modelled on single chip Virtex-5 FPGA device having 69120 bit slices, 148 BRAM, and 64 DSP cells for custom logic reconfiguration. Each bit slice in the targeted Virtex-5 FPGA consists of four function generators, four storage elements, arithmetic logic gates, large multiplexers, and fast carry look-ahead chain.

3.2. Targeted RHSCS Architecture. The RHSCS consists of a processor MicroBlaze (available as softcore IP in Xilinx Embedded Development Kit) configured in part of FPGA as softcore PE and multiple RLUs configured in remaining part of FPGA as hardcore PEs. The hardcore PEs in RHSCS act as reconfigurable computing area and support dynamic reconfiguration for hardware tasks. The softcore PE and hardcore PE in RHSCS are used to execute software tasks and hardware tasks of an application, respectively. The RHSCS is also equipped with shared memory and cache memory to store task executable files and data. The cache memory supports softcore PE to store instructions as well as data whereas the shared memory stores the task executables and input/output data for both softcore PE and hardcore PE. The resources in targeted architecture are interconnected through high speed communication protocols that support data interchange between memory and PEs. The memory and communication protocols are also configured on the chip where PEs exist. In RHSCS, the RLU size is maintained constant and tasks are assigned to the RLUs based on area required for their execution. In this research, the resource reconfiguration latency is assumed as constant and is not accounted for in performance calculations.

3.3. Task Distribution Flow. The RHSCS offers cost effective solution for computationally intensive applications through hardware reuse. So, there is a need for mapping potentially parallel tasks in an application to the resources of RHSCS. An overview of different steps in distribution of tasks of an application to the platform RHSCS is demonstrated in Figure 1.

Initially, an application is represented as Directed Acyclic Graph (DAG) and the tasks of DAG are sent to prioritization module and then to HW/SW resource mapping module. The prioritization module assigns priorities to the tasks of DAG based on their attributes in such a way that ensures schedulability. The HW/SW resource mapping module partitions the tasks into three types called software tasks (ST), hardware tasks (HT), and hybrid tasks (HST) based on their attributes and preemption nature, as stated below.

Rule 1. The set of tasks which can be preempted and could not find required area RLU on RHSCS can be treated as software task set (ST): $\mathrm{ST} = \{\mathrm{st}_1, \mathrm{st}_2, \ldots, \mathrm{st}_m\}$, $\mathrm{st}_i \in \mathrm{ST}$ $(1 \leq i \leq m)$, having the parameters a_i, d_i, and se_i, and they could run only on softcore PE (i.e., microprocessor configured on FPGA) of RHSCS.

Rule 2. The set of tasks which cannot be preempted and could find required area RLU on RHSCS can be treated as hardware task set (HT): $\mathrm{HT} = \{\mathrm{ht}_1, \mathrm{ht}_2, \ldots, \mathrm{ht}_n\}$, $\mathrm{ht}_i \in \mathrm{HT}$ $(1 \leq i \leq n)$, having parameters a_i, d_i, w_i, c_i, and he_i, and they could run only on hardcore PE (i.e., RLU configured on FPGA) of RHSCS.

Rule 3. The set of tasks which can be preempted and could find required area RLU on RHSCS can be treated as hybrid task set (HST): $\mathrm{HST} = \{\mathrm{hst}_1, \mathrm{hst}_2, \ldots, \mathrm{hst}_p\}$, $\mathrm{hst}_i \in \mathrm{HST}$ $(1 \leq i \leq p)$, having parameters a_i, d_i, w_i, c_i, se_i, and he_i, and

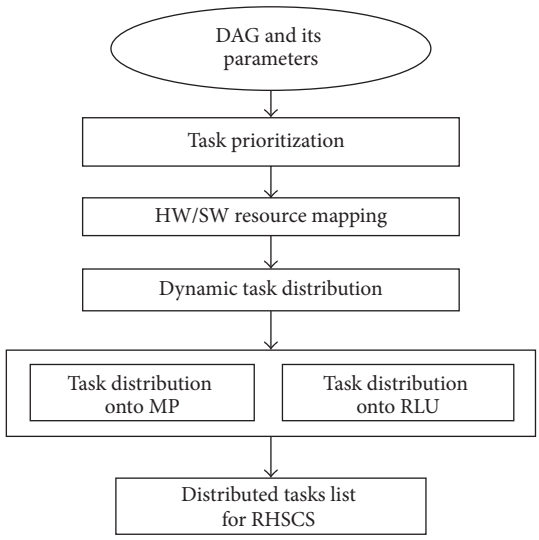


Figure 1: Overview of task distribution flow.

they could run either on softcore PE or on hardcore PE of RHSCS. The hybrid tasks in HST set can be treated as software tasks or hardware tasks based on resources availability at the instant of task distribution for execution.

The partitioned tasks are further sent to task distribution stage. In the task distribution stage, distribution tasks list is prepared for the resources of RHSCS based on task distribution policy and resources availability. The task distribution can be done statically or dynamically, stated as follows.

Static Task Distribution. The static task distribution considers all task attributes needed for task distribution, such as the structure of the application, execution time of individual tasks, and communication cost between the tasks, in advance, and makes task distribution decisions statically once at the start of task distribution and cannot be changed throughout.

Dynamic Task Distribution. The dynamic task distribution also considers task attributes needed in advance but it makes task distribution decisions dynamically at runtime based on the resources availability and task distribution policy. The aim of dynamic task distribution is not only enhancing the execution time but also optimizing resources utilization while minimizing communication overheads.

The static task distribution and dynamic task distribution methodologies are demonstrated with an example in the next subsection.

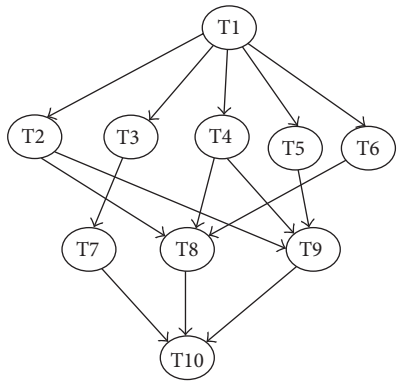
3.4. Motivational Example. A hypothetical sample task graph [8, 25] is shown in Figure 2 and targeted onto the RHSCS having one softcore PE (microprocessor) and three hardcore PEs (i.e., RLUs) as computing resources.

Generally, execution time of task graph depends on computing resources on which the tasks are executed. The various configurations of computing platform RHSCS for execution of the hypothetical sample task graph (HTG), shown in Figure 2, are demonstrated in Figure 3 with their respective execution timings in nanosecond.

The HTG execution on single core microprocessor configured in FPGA is shown in Figure 3(a) and its ideal execution time is 127 ns. Similarly, execution of the HTG on a RLU configured in FPGA is shown in Figure 3(b) and its execution time is 101 ns. So, execution time of the application can be minimized when RLU alone acts as computing resource. As the FPGA supports partial reconfiguration, FPGA is clustered into multiple RLUs to support parallel task execution and it further minimizes execution time of real time applications. Static distribution of parallel tasks in HTG to a reconfigurable computing system having three RLUs gives execution time of 65 ns as shown in Figure 3(c). Similarly, parallel tasks in HTG are distributed dynamically to the reconfigurable computing system having three RLUs as shown in Figure 3(d) and its execution time is 63 ns. In real time, tasks called critical tasks may demand higher size RLUs which are not made available on FPGA so that the critical tasks remain forever in waiting for resources and this leads to infinite execution time; that is, task graph does not get executed completely to meet its deadline. Such critical tasks can be represented as software tasks to be executed on microprocessor. In this work, tasks are preferred to be executed on RLUs but the tasks which do not find required size RLU on FPGA are treated as critical tasks, that is, software tasks are executed on microprocessor. For example, if we assume that the size of RLUs made available on FPGA is 200 bit slices, the tasks T1, T9, and T10 in HTG become critical tasks and the HTG does not get executed completely on the RLUs made available in FPGA. The scenario of HTG distribution to the platform having three RLUs (where each RLU size is equal to 200 bit slices) configured on FPGA is shown in Figure 3(e). In Figure 3(e), the tasks T9 and T10 do not find required RLU area and the tasks wait forever for execution that leads to infinite execution time. The infinite execution time indicates that the HTG is not completely executed (i.e., tasks T9 and T10 are not executed) due to lack of resources and it can be addressed effectively by introducing a microprocessor in combination with RLUs on a single chip FPGA. The static task distribution for such on-chip RHSCS platform, having microprocessor and three RLUs of each with size 200 bit slices on a single chip FPGA, is shown in Figure 3(f) and its execution time is 74 ns. Since T9 and T10 are executed on softcore PE, the execution time of the HTG in Figure 3(f) is more than execution time in Figure 3(d) but it ensured schedulability for the DAG. Similarly, the dynamic task distribution of the task graph to the platform RHSCS is shown in Figure 3(g) and its execution time is 71 ns. From Figures 3(c), 3(d), 3(f), and 3(g), it is clear that the dynamic task distribution (DTD) enhances execution speed of an application compared to static task distribution (STD). So, in this paper a DTD methodology is presented and demonstrated for real life benchmark applications.

4. Dynamic Task Distribution Methodology

The task distribution methodology dynamically decides task execution on the resources of RHSCS. The proposed DTD methodology decides optimal task execution sequence and speedup application execution.



Task (node)	Area (number of bit slices on FPGA)	Execution time (ns)	
		Softcore processor	FPGA
T1	200	14	12
T2	180	13	10
T3	120	11	9
T4	180	13	10
T5	150	12	9
T6	170	13	11
T7	90	7	5
T8	70	5	3
T9	250	18	15
T10	300	21	17

Figure 2: Hypothetical sample task graph [8, 25] and its attributes.

4.1. Dynamic Task Distribution Model. In order to achieve high efficiency in hardware utilization and speed up the application execution, the DTD model is described in three levels as shown in Figure 4. Level 1 provides interface to load the tasks of an application represented as DAG and the tasks are arranged as per their level in DAG. The level annotated tasks are stored into DAG Queue according to their level increasing order. In level 2, the tasks in DAG Queue are mapped to hardware and software resources of RHSCS and then partitioned into software tasks (ST), hardware tasks (HT), and hybrid tasks (HST) depending on their design parameters as stated in Section 3.3. The partitioned ST, HT, and HST are stored in Hardware Task Queue (HT Queue), Software Task Queue (ST Queue), and Hybrid Task Queue (referred to as HST Queue), respectively. In level 3, the tasks in HT Queue, ST Queue, and HST Queue are prioritized dynamically based on their predefined parameter Minimum Laxity First (MLF). The MLF based prioritized tasks are then sent to either CPU Implementation Queue or RLU Implementation Queue based on tasks execution nature and computing resources availability. The resource mapped tasks in CPU Implementation Queue and RLU Implementation Queue are then distributed to the resources of RHSCS for execution. The DTD based on MLF distribution policy is shown in Figure 4 having seven modules to describe its behaviour.

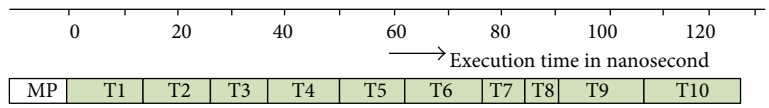
The Application Decode Module (ADM) loads and stores the tasks of DAGs into DAG Queue. The Task Annotation Module arranges the tasks in DAG Queue based on their level in DAG. The HW/SW Task Partitioning Module maps the tasks in DAG Queue to the resources of computing platform RHSCS and stores them into ST Queue, HT Queue, and HST Queue. Dynamic Task Prioritization Module assigns priorities dynamically based on MLF distribution policy to the tasks in ST Queue, HT Queue, and HST Queue. The Task Load Module loads the task executable files for execution onto softcore PE of RHSCS. Similarly, the Task Configuration Module configures the task bit-stream files for execution onto hardcore PEs, that is, RLUs of RHSCS. The pseudo codes for reading the tasks of DAG and tasks level annotation in level 1, HW/SW resource mapping in level 2, and dynamic task distribution in level 3 are discussed in the coming subsections.

4.2. Task Level Annotation. In level 1, the Application Decode Module loads the applications described as DAG and computes the adjacency matrix for the DAG that describes dependency of the tasks in a DAG. The adjacency matrix also holds the level [11] of individual tasks in DAG and Task Level Annotation Module finds the level of individual tasks and arranges them in the level increasing order. In any DAG, source task gets first level and sink task gets last level in order to maintain dependency between tasks while executing. The pseudo code to read DAG and to annotate the levels of the tasks of DAG is described in Algorithm 1.

Time complexity of the task level annotation algorithm depends on number of DAGs and maximum number of tasks in a DAG. The time complexity for task level annotation would be $O(M \times N^2)$ when there are M number of DAGs and maximum of N tasks in a DAG. The level annotated tasks of DAG are sorted according to their level increasing order and then moved to HW/SW resource mapping stage using Task_Resource_Mapping function that maps the task to the resources of RHSCS.

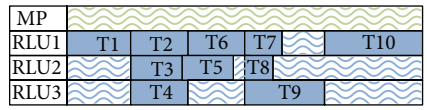
4.3. HW/SW Resource Mapping. In this stage, the level annotated tasks in DAG are mapped to the resources of RHSCS and partitioned [15] into software tasks (ST) and hardware tasks (HT). The hybrid tasks (HST) category is not considered in this paper because all tasks of DAGs are assumed as nonpreemption tasks. The Software Task Queue (ST Queue) and Hardware Task Queue (HT Queue) in task distribution model are reserved to store software tasks and hardware tasks, respectively. Initially these queues would be empty and then the partitioned ST and HT are stored into the respective queue as their level increasing order. The pseudo code of the function Task_Resource_Mapping for mapping tasks of DAG to the resources of RHSCS is described in Algorithm 2.

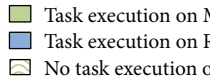
Time complexity of the task resource mapping algorithm depends on maximum number of tasks in a DAG. The time complexity for resource mapping would be $O(N^2)$ when there



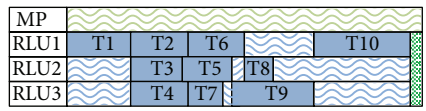
(a) HTG tasks distribution onto microprocessor (MP)

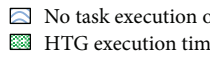
(b) HTG tasks distribution onto single RLU configured in FPGA



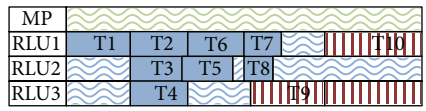


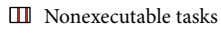
(c) Static task distribution of HTG tasks onto three partial RLUs in FPGA



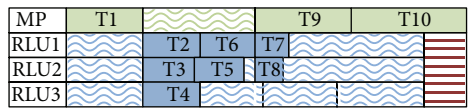


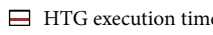
(d) Dynamic task distribution of HTG tasks onto three partial RLUs in FPGA



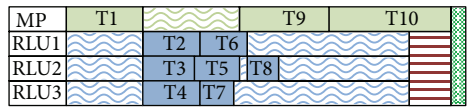


(e) HTG critical tasks distribution onto three partial RLUs in FPGA





(f) Static task distribution of HTG tasks onto RHSCS



(g) Dynamic task distribution of HTG tasks onto RHSCS

Figure 3: RHSCS configurations for static and dynamic task distribution.

are N tasks in a DAG. The resource mapped tasks in HT Queue and ST Queue are further moved to task distribution stage using Task_Distribution function that configures the tasks onto the resources of RHSCS for their execution.

4.4. Task Distribution. Task distribution is demonstrated in two phases as combination of dynamic task prioritization and resource management. The partitioned tasks in Algorithm 2 are further prioritized based on task distribution policy called Minimum Laxity First (MLF) and distributed them onto available resources of RHSCS. There are RLU Implementation Queue and CPU Implementation Queue, in task distribution model, reserved to store the tasks which could be executed on hardcore PE (RLUs) and

```
/* Input: DAGs of application and number of DAG
   Output: Level annotated tasks of DAGs */
(1)  Read number of DAGs
(2)  for i = 1 to number of DAG do
(3)     Read number of tasks in DAG_i
(4)     for j = 1 to number of tasks in DAG_i do
(5)          Read the number of tasks which depend on task T_j
(6)     end for
(7)     Compute Level of the tasks in DAG_i
(8)     for j = 1 to number of tasks in DAG_i do
(9)          while (Level of task T_j > 0) do
(10)              Assign Level to the task T_j
(11)         end while
(12)    end for
(13)    Sort the tasks in DAG_i according to the Level assigned
(14)    Task_Resource_Mapping (Level annotated tasks of DAG_i, number of tasks in the DAG_i)
          /* Function of Task_Resource_Mapping is described in Algorithm 2 */
(15) end for
```

Algorithm 1: Pseudo code for task level annotation.

```
/* Input: Level annotated tasks of a DAG and number of tasks in the DAG
   Output: ST and HT Partitioned tasks of DAG */
(1)  Read the Level annotated tasks of DAG and number of tasks in DAG from algorithm
(2)  Initialize HT Queue and ST Queue
(3)  while (number of Level annotated tasks in DAG > 0) do
(4)     for j = 1 to number of Level annotated tasks in DAG do
(5)          if area of task T_j < size of available RLU then
(6)               assign T_j to HT Queue
(7)          else
(8)               assign T_j to ST Queue
(9)          end if
(10)    end for
(11)  end while
(12)  Task_Distribution (partitioned tasks of DAG, number of tasks in the DAG)
        /* Function Task_Distribution is described in Algorithm 3 */
```

Algorithm 2: Pseudo code for Task_Resource_Mapping function.

softcore PE (microprocessor) of RHSCS. The pseudo code for Task_Distribution function that distributes the tasks of DAG onto the resources of RHSCS is described below as Algorithm 3.

The Task_Distribution function in Algorithm 3 accepts the partitioned tasks of a DAG as input and computes Minimum Laxity First (MLF) parameter for the tasks which are in HT Queue and ST Queue. The expression for MLF is $t_{\mathrm{MLF}} = d_j - e_j - a_j$ for task T_j and it represents time flexibility of the task for execution. The MLF acts as task distribution policy to prioritize the partitioned parallel tasks before distributing them onto RHSCS. The RLU Implementation Queue holds the tasks which could be executed on hardcore PE (RLUs) whereas the CPU Implementation Queue holds the tasks which could be executed on softcore PE (MP). The tasks in HT Queue are sent to RLU Implementation Queue and the tasks in ST Queue are sent to CPU Implementation Queue. Finally, the tasks in RLU Implementation Queue and CPU Implementation Queue are distributed and executed on hardcore PE and softcore PE, respectively. Time complexity of the task distribution presented in Algorithm 3 depends on maximum number of tasks in a DAG and also on number of PEs in RHSCS. So, the time complexity of task distribution would be $O(N^2 \times (P + Q))$ when there are N tasks in a DAG, P hardware PEs, and Q software PEs in RHSCS. In real time, number of tasks in an application is always very much greater than number of PEs in RHSCS. So, the time complexity of task distribution is $O(N^2)$, where $N \gg (P + Q)$.

Time complexity of the proposed DTD methodology depends on time complexity of task level annotation, HW/SW resources mapping, and task distribution algorithms. The time complexity of DTD model is $O(M \times N^2)$ for M number of DAGs with maximum of N tasks in each DAG.

```
/* Input: partitioned tasks of a DAG and number of tasks in the DAG
   Output: Resources assignment and dynamic task execution order for the tasks in a DAG */
(1)  Read the partitioned tasks of a DAG and number of tasks in the DAG from the Algorithm 2
(2)  Initialize RLU Impementation Queue and CPU Impementation Queue
(3)  while (number of tasks in DAG > 0) do
(4)    for j = 0 to number of tasks in DAG do
(5)        Compute the cost function MLF for the task T_j in their respective queues
(6)    end for
(7)    Assign Priority to the partitioned tasks in queues according to their MLF
(8)    Sort tasks of DAG according to their assigned priority increasing order
(9)    for j = 0 to number of tasks in DAG do
(10)       if (T_j ∈ HT Queue) then
(11)           assign T_j to RLU Implementation Queue
(12)    else
(13)           assign T_j to CPU Implementation Queue
(14)       end if
(15)   end for
(16)   while ((RLU Implementation Queue! = empty) &&
                    (CPU Implementation Queue! = empty)) do
(17)       for each RLU in RHSCS do
(18)           if (RLU available == True) then
(19)               Assign next task from RLU Implementation Queue to available RLU
(20)           end if
(21)       end for
(22)       for each MP in RHSCS do
(23)           if (MP available == True) then
(24)               Assign next task from CPU Implementation Queue to available MP
(25)           end if
(26)       end for
(27)   end while
(28) end while
```

Algorithm 3: Pseudo code for Task_Distribution function.

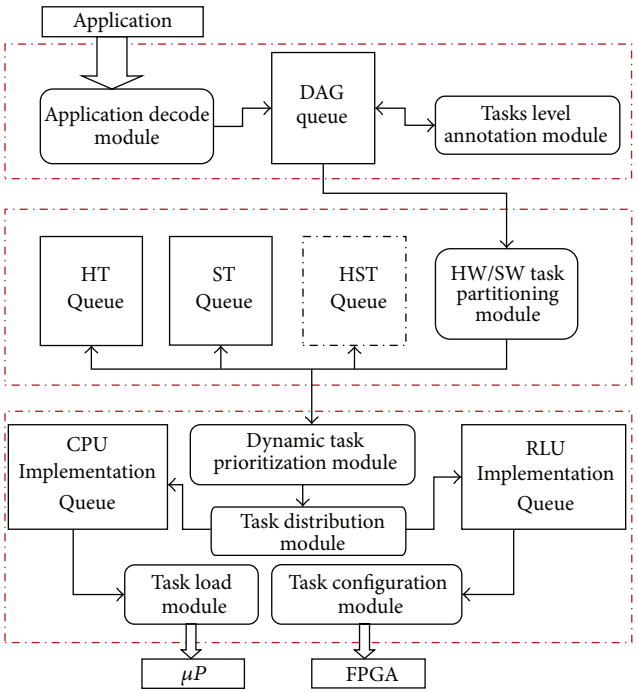


Figure 4: Dynamic task distribution model.

5. Result and Discussion

This section presents implementation scheme, experimental results obtained, performance evaluation of the DTD-MLF methodology, and RHSCS resources utilization.

5.1. Implementation Scheme. Modelling of RHSCS environment and methods followed for application execution on RHSCS is discussed in this subsection.

5.1.1. Modelling of RHSCS Architecture. In this research, RHSCS platform is realized on Virtex-5 FPGA (Virtex-5 XC5VLX110T), as shown in Figure 5, using Xilinx EDK where a MicroBlaze softcore PE is configured in part of the reconfigurable area of FPGA and the rest of reconfigurable area is used for configuration of multiple RLUs, memory, and communication protocols. In the realized RHSCS, the MicroBlaze is a 32-bit RISC architecture equipped with instruction and data cache memory of size 4 KB each, for storing instructions as well as data while executing tasks.

The RLU configures its custom hardware for hardware tasks and also it supports hardware tasks interface with off-chip peripherals. The on-chip BRAM of size 64 KB acts as shared memory for MicroBlaze and RLUs to store executable

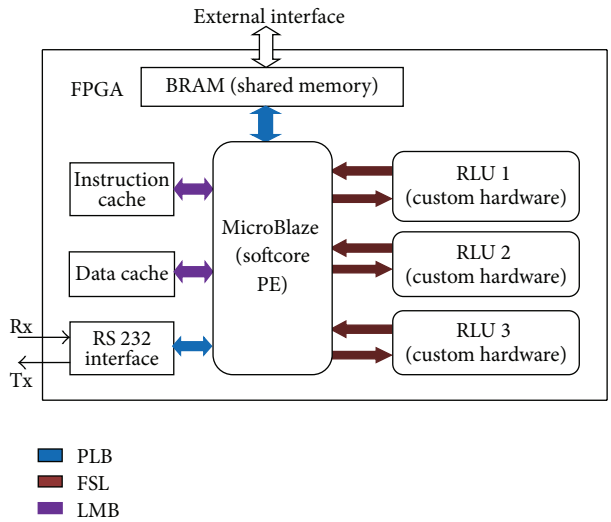


Figure 5: On-chip Reconfigurable High Speed Computing System.

files, input, and output data. BRAM memory controller is configured along with BRAM to load task executables, input, and output data from external environment and also it controls data interchange between BRAM and MicroBlaze. The data interchange between BRAM and custom hardware can be done through MicroBlaze with the help of communication protocols. These functional blocks MicroBlaze, RLUs, BRAM, instruction, and data cache memory are interconnected through communication protocols like Processor Local Bus (PLB), Local Memory Bus (LMB), and Fast Simplex Link (FSL). The PLB provides interface between MicroBlaze and BRAM through BRAM controllers that load instructions, input data, and store back output data after computation. The LMB supports interfacing of cache memory with MicroBlaze to minimize memory access overheads. The FSL is used to interface custom hardware configured in RLU with MicroBlaze and it has 32-bit FIFO implemented on BRAM to support data streaming between MicroBlaze and custom hardware. Since the Virtex-5 FPGA (Virtex-5 XC5VLX110T) device contains total 69120 bit slices, 148 BRAM, and 64 DSP cells for custom logic reconfiguration, the on-chip RHSCS configured on Virtex-5 FPGA utilized 3825 bit slices, 4 BRAM cells, and 3 DSP cells for various functional blocks and communication protocols. The configured MicroBlaze runs at 125 MZ speed.

5.1.2. Evaluated Applications and HW/SW Development Flow. The behavior of DTD methodology has been demonstrated in Figure 3 with the help of hypothetical sample task graph. In order to evaluate the effectiveness of the DTD-MLF, a few benchmark applications are represented as DAG and the tasks of DAG are distributed onto RHSCS for their execution. The Xilinx standard embedded SW development flow supports task execution on softcore PE whereas the standard FPGA HW development flow supports task execution on RLU.

Embedded SW Development Flow. The behaviour of the tasks in DAG is described in C++ to load and execute them on softcore PE of RHSCS. The tasks C++ code is cross compiled to the softcore PE and executable files are generated. The tasks executable files are executed on MicroBlaze and software design attribute; that is, software execution time (*se*) is acquired. The task executable files and software task attributes are stored in memory for future execution.

Embedded HW Development Flow. The behaviour of the tasks in DAG is described in HDL to configure and execute them on RLU of RHSCS. The HDL code is synthesized to targeted device Xilinx Virtex-5 (XC5VLX110T) to generate gate level netlist and that produces configuration file required for task execution on RLU of RHSCS. The task configuration files are configured on reconfigurable area of FPGA and executed to acquire hardware design attributes, that is, area (*w*) and execution time (*he*). The EDK does not support task switching while executing because hardware tasks are nonpreemptible in nature. The task configuration files and obtained hardware task attributes are stored in memory for their future configuration.

5.2. Performance Metrics. In the literature many researchers have developed methods to enhance execution speed, schedulable bound, and resource utilization. This paper is aimed at improving upon the schedule length, that is, execution speed of an application and effective utilization of RHSCS resources.

5.2.1. Schedule Length. In a DAG, task without any predecessor is an entry task and task without successor is an exit task. Time taken to execute the tasks from entry task to exit tasks in a DAG is called schedule length of the DAG. The schedule length of a DAG depends on computing resources on which the tasks run. The schedule length has to be minimized to achieve optimum execution time for an application.

5.2.2. Resource Utilization. The resource utilization of computing platform is estimated based on the tasks allocated to individual resources of computing platform and time spent in execution of the tasks. An expression to calculate resources utilization is as follows:

$$\text{Resource utlization} = \frac{\sum_{t=0}^{\text{ET}} n \times t}{N \times \text{ET}}, \tag{1}$$

where n is the number of parallel resources utilized in a time slot t, N is the total number of resources in computing platform, and ET is the total execution time of an application.

5.3. Performance Evaluation of Dynamic Task Distribution Model. The dynamic task distribution model based on MLF criteria (DTD-MLF) distributes the tasks of an application to the resources of computing platform RHSCS dynamically based on the cost function MLF of the tasks in DAG. Initially, the DTD-MLF methodology is applied to a HTG [8] shown in Figure 2 and then to the application JPEG shown in Figure 6. The functional behaviour of JPEG application is represented as task graph shown in Figure 6, where T1 is grey conversion, T2 matrix transpose, T3 DCT wrapper 1, T4 DCT wrapper 2, T5 quantization, T6 encoder, and T7 memory read/write.

Table 1: Schedule length and resource utilization of HTG and JPEG based on STD-MLF and DTD-MLF distribution polices.

Task graph	Number of tasks	Schedule length (ns)		% of resource utilization	
		STD-MLF	DTD-MLF	STD-MLF	DTD-MLF
HTG	10	72.0	63.0	35.10	40.00
JPEG	7	40.8	39.1	27.60	28.90
HTG + JPEG	17	96.0	73.0	38.60	50.10

Table 2: Benchmark applications and their tasks distribution to RHSCS.

Task graph	Number of tasks	Schedule length (ns)		% of resource utilization	
		STD-MLF	DTD-MLF	STD-MLF	DTD-MLF
DCT	43	96.25	80.53	58.86	70.93
Diffeq.	15	40.75	28.15	45.50	65.00
Ellip.	38	93.43	80.47	49.27	57.23
FIR	15	52.85	34.37	37.28	57.32
IIR	16	45.4	31.54	45.27	65.17
Lattice	23	59.59	51.61	48.33	55.80
Nc.	61	129.02	115.16	64.63	72.40
Voltera	29	72.36	61.26	54.20	64.02
Wavelet	43	88.04	78.04	63.32	71.43
Wdf7	53	103.92	95.57	63.53	69.09

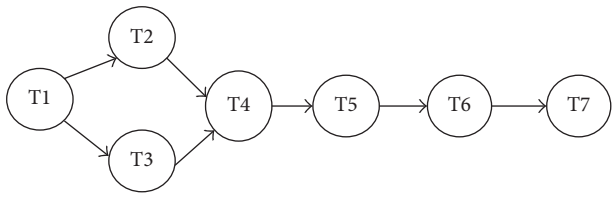


Figure 6: JPEG task graph.

The tasks in HTG are distributed to the resources of RHSCS based on DTD-MLF model as well as static task distribution [4, 9] model with MLF as cost function (STD-MLF). Since RLU maximum area is 200 bit slices, the tasks T9 and T10 in HTG are treated as software tasks and mapped onto MicroBlaze for their execution. DTD-MLF minimizes the schedule length by 12.5% when compared to STD-MLF and also resource utilization is enhanced in DTD-MLF over STD-MLF. The tasks in JPEG task graph are distributed to the resources of RHSCS and then tasks of independent task graph HTG and JPEG together are distributed onto RHSCS. The schedule length and resource utilization of the task graphs are shown in Table 1.

Figures 7(a) and 7(b) show the performance improvement in application execution and enhancement in resource utilization obtained by DTD-MLF compared to STD-MLF. From Figure 7(a), our approach DTD-MLF minimizes the schedule length 12.5% for HTG and 4.2% for JPEG task graph and it is 23.9% when both HTG and JPEG together are targeted for execution, wherein, in Figure 7(b), the RHSCS resources utilization is enhanced by 13.9% for HTG, 4.7% for JPEG, and 29.8% when HTG and JPEG are executed together.

The DTD-MLF and STD-MLF methodologies are further applied to few real life benchmark applications summarized in first column of Table 2.

As stated in Section 5.1.2, the benchmark applications are represented as DAG and then the standard embedded SW development flow and FPGA HW development flow are used to acquire hardware software task attributes on RHSCS. The tasks of the benchmark applications, represented as DAG, are distributed onto RHSCS statically as well as dynamically based on MLF criteria. The number of tasks, schedule length, and resource utilization of the benchmark applications are presented in Table 2. Table 2 demonstrates that the DTD-MLF model minimized schedule length for the benchmark applications over STD-MLF and also the benchmark applications utilized the resources of RHSCS effectively in DTD-MLF compared to STD-MLF. The effectiveness of the DTD-MLF methodology over STD-MLF methodology [4, 9] in terms of schedule length and resource utilization of RHSCS for the selected benchmark applications is shown in Figures 8 and 9, respectively.

From the results, the presented DTD-MLF methodology boosted the application execution over STD-MLF by 16.33% for DCT, 30.92% for Diffeq., 13.97% for Ellip., 34.96% for FIR, 30.52% for IIR, 13.39% for Lattice, 10.74% for Nc., 15.34% for Voltera, 11.36% for Wavelet, and 8.04% for Wdf7. The DTD-MLF enhanced the RHSCS resource utilization over STD-MLF model by 20.51% for DCT, 42.86% for Diffeq., 16.16% for Ellip., 53.76% for FIR, 43.96% for IIR, 15.46% for Lattice, 12.02% for Nc., 18.11% for Voltera, 12.08% for Wavelet, and 8.75% for Wdf7.

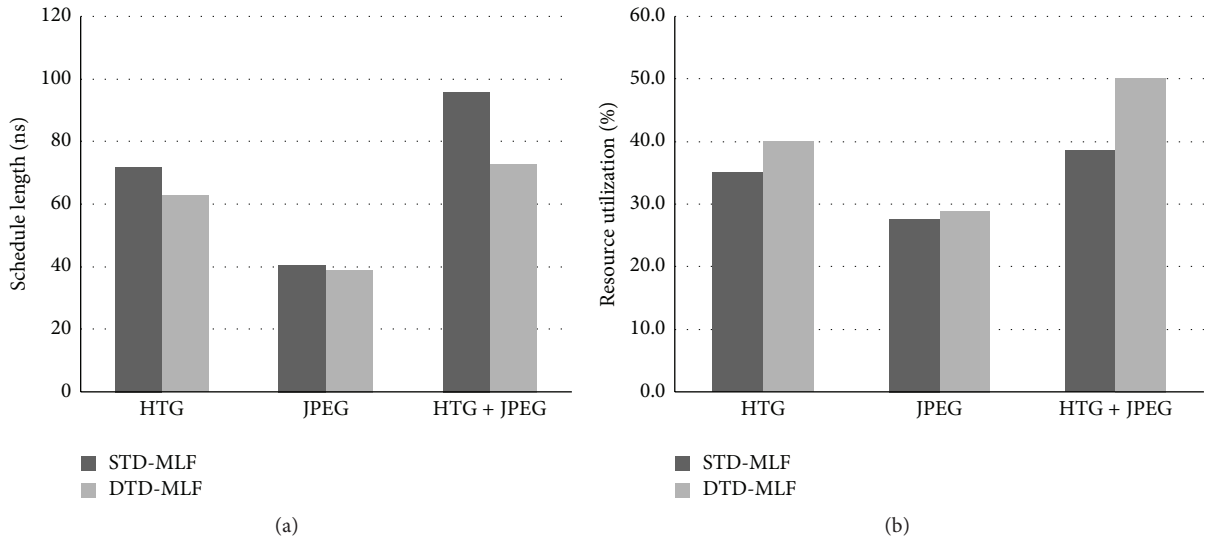


Figure 7: Performance improvement of HTG, JPEG task graphs on RHSCS (a) schedule length and (b) resource utilization.

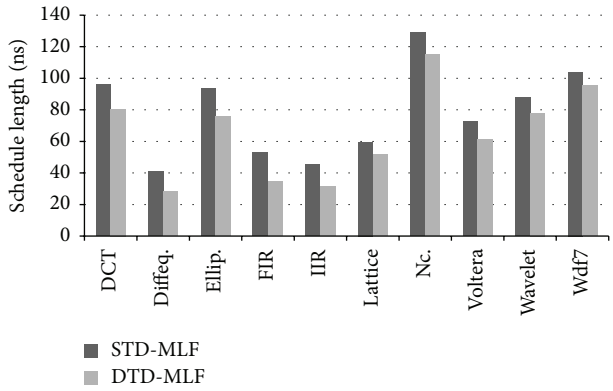


Figure 8: Schedule length of the benchmark applications on RHSCS in both STD-MLF and DTD-MLF scenario.

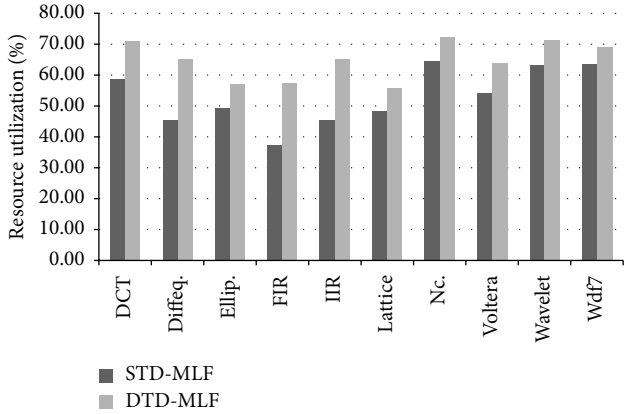


Figure 9: RHSCS resource utilization of benchmark application in both STD-MLF and DTD-MLF scenario.

6. Conclusion

In this paper, we have presented DTD-MLF methodology for an on-chip heterogeneous reconfigurable computing platform RHSCS and estimated its effectiveness in execution of selected benchmark applications. The RHSCS has been realized on Virtex-5 FPGA device for applications execution. The RHSCS contains MicroBlaze as softcore PE and multiple RLUs configured on FPGA as hardcore PE. A few benchmark applications have been represented as DAG and design attributes of the tasks in DAG were obtained offline by executing them on the resources of RHSCS. The obtained design attributes of the tasks in DAG have been utilized to find cost function called Minimum Laxity First (MLF) which acts as the criteria for task distribution. The benchmark applications represented as DAG were distributed onto the resources of RHSCS based on DTD-MLF and STD-MLF methodologies. As compared to STD-MLF, the DTD-MLF model boosted the execution speed of benchmark applications up to 34.96%. The DTD-MLF methodology also enhanced the RHSCS resources utilization up to 53.75% for the chosen benchmark applications.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] R. Hartensstein, "Microprocessor is no more general purpose: why future reconfigurable platforms will win," in *Proceedings of the International Conference on Innovative Systems in silicon (ISIS '97)*, pp. 1–10, Austin, Tex, USA, October 1997.

[2] J. Lyke, "Reconfigurable systems: a generalization of reconfigurable computational strategies for space systems," in *Proceedings of the IEEE Aerospace Conference Proceedings*, vol. 4, pp. 4-1935–4-1950, 2002.

[3] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. L. M. Bertels, G. Kuzmanov, and E. M. Panainte, "The MOLEN polymorphic processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, 2004.

[4] Z. Pan and B. E. Wells, "Hardware supported task scheduling on dynamically reconfigurable SoC architectures," *IEEE Transactions on VLSI Systems*, vol. 16, no. 11, pp. 1465–1474, 2008.

[5] S. Darbha and D. P. Agrawal, "Optimal scheduling algorithm for distributed-memory machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 1, pp. 87–95, 1998.

[6] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506–521, 1996.

[7] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA '99)*, pp. 328–335, IEEE, Hong Kong, December 1999.

[8] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[9] M. I. Daoud and N. Kharma, "A high performance algorithm for static task scheduling in heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 68, no. 4, pp. 399–409, 2008.

[10] D. Wang, S. Li, and Y. Dou, "Loop Kernel pipelining mapping onto coarse-grained reconfigurable architecture for data-intensive applications," *Journal of Software*, vol. 4, no. 1, pp. 81–89, 2009.

[11] S. R. Kota, C. Shekhar, A. Kokkula, D. Toshniwal, M. V. Kartikeyan, and R. C. Joshi, "Parameterized module scheduling algorithm for reconfigurable computing systems," in *Proceedings of the 15th International Conference on Advanced Computing and Communication (ADCOM '07)*, pp. 473–478, IEEE, Guwahati, India, December 2007.

[12] A. Ahmadinia, C. Bodda, and J. Teich, "A dynamic scheduling and placement algorithm for reconfigurable hardware," in *Organic and Pervasive Computing—ARCS 2004*, vol. 2981 of *Lecture Notes in Computer Science*, pp. 125–139, 2004.

[13] X.-G. Zhou, Y. Wang, X.-Z. Huang, and C.-L. Peng, "Online scheduling of real-time tasks for reconfigurable computing system," in *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT '06)*, pp. 57–63, Bangkok, Thailand, December 2006.

[14] M. Fazlali, M. Sabeghi, A. Zakerolhosseini, and K. Bertels, "Efficient task scheduling for runtime reconfigurable systems," *Journal of Systems Architecture*, vol. 56, no. 11, pp. 623–632, 2010.

[15] L. Liang, X.-G. Zhou, Y. Wang, and C.-L. Peng, "Online hybrid task scheduling in reconfigurable systems," in *Proceedings of the 11th International Conference on Computer Supported Cooperative Work in Design (CSCWD '07)*, pp. 1072–1077, Melbourne, Australia, April 2007.

[16] M. M. Bassiri and H. S. Shahhoseini, "Online HW/SW partitioning and co-scheduling in reconfigurable computing systems," in *Proceedings of the 2nd IEEE International Conference on Computer Science and Information Technology (ICCSIT '09)*, pp. 557–562, IEEE, Beijing, China, August 2009.

[17] P. Saha and T. El-Ghazawi, "Software/hardware co-scheduling for reconfigurable computing systems," in *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '07)*, pp. 299–300, Napa, Calif, USA, April 2007.

[18] P. Saha and T. El-Ghazawi, "Extending embedded computing scheduling algorithms for reconfigurable computing systems," in *Proceedings of the 3rd Southern Conference on Programmable Logic (SPL '07)*, pp. 87–92, Mar del Plata, Argentina, February 2007.

[19] M. Sabeghi, V.-M. Sima, and K. Bertels, "Compiler assisted runtime task scheduling on a reconfigurable computer," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '09)*, pp. 44–50, IEEE, Prague, Czech Republic, September 2009.

[20] J. Noguera and R. M. Badia, "HW/SW co-design techniques for dynamically reconfigurable architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 4, pp. 399–415, 2002.

[21] M. M. Bassiri and H. S. Shahhoseini, "A HW/SW partitioning algorithm for multitask reconfigurable embedded systems," in *Proceedings of the 20th International Conference on Microelectronics (ICM '08)*, pp. 143–146, Sharjah, United Arab Emirates, December 2008.

[22] J. A. Clemente, J. Resano, and D. Mozos, "An approach to manage reconfigurations and reduce area cost in hard real-time reconfigurable systems," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 4, pp. 90.1–90.24, 2014.

[23] S. Saha, A. Sarkar, and A. Chakrabarti, "Scheduling dynamic hard real-time task sets on fully and partially reconfigurable platforms," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 23–26, 2015.

[24] M. Vucha and A. Rajawat, "An effective dynamic scheduler for reconfigurable high speed computing system," in *Proceedings of the IEEE International Advance Computing Conference (IACC '14)*, pp. 766–773, February 2014.

[25] M. Vucha and A. Rajawat, "Task scheduling methodologies for high speed computing systems," *International Journal of Embedded Systems and Applications*, vol. 4, no. 3, 2014.

[26] M. Vucha and A. Rajawat, "A novel methodology for task distribution in heterogeneous reconfigurable computing system," *International Journal of Embedded Systems and Applications*, vol. 5, no. 1, pp. 19–39, 2015.