Hardware Supported Task Scheduling on Dynamically Reconfigurable SoC Architectures

Zexin Pan and B. Earl Wells, *Member, IEEE*

***Abstract*—Dynamically reconfigurable system-on-a-chip (RSoC) technology features embedded microprocessors that are dispersed on the same die with significant amounts of programmable logic fabric. In this paper, we present a strategy to solve the recently emerging problem of how to utilize the flexible but still limited RSoC resources in an effective manner for a multi-task application. The major contribution of this paper is the development of a dynamic task scheduling algorithm that can be implemented in fixed or reconfigurable hardware that will perform the online scheduling of task systems onto the RSoC type architecture. The results from extensive simulations demonstrate the benefits of the proposed dynamic scheduling approach as compared with that of other static scheduling techniques taken from the technical literature.**

***Index Terms*—Reconfigurable architectures, scheduling, VLSI.**

I. INTRODUCTION

DYNAMICALLY reconfigurable system-on-a-chip (RSoC) architectures, which integrate in the same die embedded microprocessors, on-chip memory, reconfigurable logic blocks, and multiple Intellectual Property (IP) cores, are now practical and commercially available. Such architectures promise the flexibility of traditional general-purpose processors while also providing the efficiency and high performance of application-specific integrated circuits (ASICs). An example is the Xilinx Virtex-4 family of field-programmable gate arrays (FPGAs) that integrates on the same IC up to two PowerPC 405 processors with up to 200 000 programmable logic cells [1].

To manage the complexity and tap the full potential of these RSoC architectures presents many challenges [2]. One of the most daunting challenges is how to efficiently schedule a given set of tasks that makes up an application onto the RSoC. Task scheduling is a well-known NP-complete problem [3] that has also been addressed many times throughout the technical literature in reconfigurable computing.

Extensive research has been performed in static scheduling (e.g., compiler-time scheduling) as is evidenced by the large number of heuristic algorithms [4]–[7]. However, these approaches face difficulties in dealing with nondeterministic systems whose run-time characteristics (e.g., execution time and execution condition) are not well known *a priori*. Typical static scheduling approaches assume that the application's execution time is the worst-case execution time [7]. This assumption often makes poor use of the available resources. Moreover, most of these approaches can only be applied to task systems that employ data-flow style graphs with no control information. This makes them inapplicable to applications that are computationally complex as well as those that are control-dominated having nondeterministic intertask communication behavior.

Operating systems for reconfigurable architectures [8], [9] are ideal for complex systems due to their applicability to nondeterministic systems. Unfortunately, these techniques often require a large code footprint, which can be prohibitive in memory-constrained embedded systems. Moreover, the required housekeeping functions associated with traditional software-based real-time operating systems tend to be a major source of power consumption in embedded system implementations [10]. Also, the responsiveness of such systems tends to be very sensitive to the attributes and complexity associated with the task system (i.e., number of tasks, task periods, task deadlines, task runtimes, etc.). Such sensitivity, caused by the highly sequential operation of the embedded processor, often results in nondeterministic run-time operation. Placing the scheduler and a portion of the task system in reconfigurable hardware can significantly improve the level of realizable parallelism and reduce overheads, leading to more deterministic execution. In summary, moving the operating system functionality into hardware may help improve an applications performance, power consumption, and level of real-time/deterministic execution.

Currently, hardware-supported dynamic task scheduling for RSoC architectures has not been deeply addressed in the literature. There are some research efforts, though, being undertaken in the area of real-time embedded systems that illustrate the benefits and performance improvement of moving into hardware functionality that traditionally has been assigned to the software-based operating systems [11], [12].

In this paper, we propose an augmented approach to dynamic scheduling of pre-partitioned task systems that are used to represent future high performance multifaceted embedded systems that contain nonuniform/nondeterministic control structures. Such applications include complex systems that contain digital signal processing, cryptography, graphical, or multimedia type components that must perform their operation under strict time constraints. As an initial step to solve theses types of problems, a microarchitecture for multitasking on reconfigurable architectures was proposed in [13]. An improvement of this technique was presented in [14]. However, a major drawback of these methods is that they cannot be applied to applications

Manuscript received May 09, 2006; revised June 07, 2007. First published September 26, 2008; current version published October 22, 2008.

Z. Pan is with Adtran, Inc., Huntsville, AL 35806 USA (e-mail: jackson.pan@adtran.com).

B. E. Wells is with the Department of Electrical and Computer Engineering, University of Alabama in Huntsville, Huntsville, AL 35899 USA (e-mail: wells@ece.uah.edu).

Digital Object Identifier 10.1109/TVLSI.2008.2000974

that have control dependencies (i.e., inter-task dependencies that are not known until run time). In previous work [15], we proposed a multitasking microarchitecture that can handle nondeterministic systems (i.e., systems with both data and control dependencies). This paper expands upon that presentation. It also presents a comprehensive set of simulation experiments that allows for an empirical analysis of the effectiveness of the scheduling methodologies to be made.

The remaining portion of this paper is organized as follows. In Section II, the various terms and the task system that is used in this research are defined. In Section III, the targeted RSoC architecture is described. In Section IV, the multitasking microarchitecture as well as the scheduling algorithm is discussed. In Section V, the simulation results are presented. In Section VI, the concluding remarks are made.

II. Task System

A. Task Definition

In this research, the input application is represented as a task system that is decomposable into a set of irregularly structured communicating tasks. Each task in the task system is assigned a task ID, a task type, and a task priority. The task ID uniquely identifies each task instance. The task type is used to identify tasks in the overall task systems that function identically with one another and are implemented in the same manner. Thus a task type could represent: 1) identical software code segments to be implemented on an embedded processor or 2) identical reconfiguration contexts (i.e., bit-streams) that are to be implemented within reconfigurable logic. The same type of task may be used many times within a task system but each task will receive distinct data and will produce unique results that are a function of its inputs. The task priority determines the urgency of each task. It consists of both a static part and a dynamic part. The static part is calculated at compile time using a static slack-based priority function [16]. The dynamic part is assigned at run time by the scheduler microarchitecture.

Each task is characterized by an appropriate set of parameters (e.g., average execution time, worst-case execution time, hardware resource area utilization, etc.). In addition, a task is assumed to conform to the following restrictions:

- a task represents a nonpreemptive unit of computation in an embedded system;
- granularity of the task is considered to be moderate to coarse in order to amortize out the fixed overhead associated with reconfiguration;
- inter-task communication is assumed to occur through an on-chip data buffer that is large enough to store the information being transferred between tasks;
- tasks for a reconfigurable logic cell (LC) may be dynamically swapped in and out using dynamic reconfiguration capability of a LC.

B. Task System Representation

The input application workload to the task scheduler is represented by a task system that is modeled as a modified directed acyclic graph (DAG) as shown in Fig. 1. Such a graph is defined as a tuple $G = (V, E^d, E^c, P)$, where V is a set of nodes

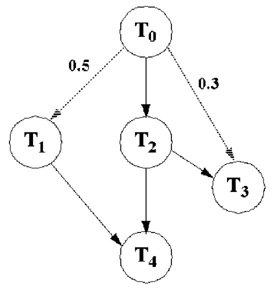


Fig. 1. Modified directed acyclic graph.

(tasks); E^d and E^c are the sets of directed data edges and directed control edges, respectively; $E^d \cap E^c = \emptyset$, $E^d \cup E^c = E$, where E is the set of all edges in G; P represents the set of probabilities associated with E^c.

The data flow dependencies that exists between two tasks are represented as a directed data edge $e_{ij}^d = (T_i, T_j)$, where $e_{ij}^d \in E^d$. These edges are represented in Fig. 1 by solid directed lines. If such an edge exists between nodes, the sink node (the data dependency sink node, e.g., T_2) is said to be data dependent on the source node (the data dependency source node, e.g., T_0). This means that the sink task must wait for the source task to complete execution (and receive the data it produces) before it can begin its execution. If the run-time conditions have determined that a data flow dependent source task is not to execute during the current task graph instantiation phase, then the precedence relationship between the two tasks is considered to be satisfied only after when it was determined that the source task would not execute. As alluded to in the previous paragraph, it is assumed that the system to be processed is first partitioned into a master set of communicating tasks where only a subset of the set of tasks are executed during a particular task graph instantiation. This is accomplished in this model, by employing directed control edges that enter and control each conditional task. A directed control edge between two tasks, T_i and T_j, is defined as $e_{ij}^c = (T_i, T_j)$ (i.e., dashed lines as shown in Fig. 1), where $e_{ij}^c \in E^c$. The inclusion of control edges results in a task graph representation that supports the execution of a nondeterministic number of task nodes and a nondeterministic partial ordering of the task system. But this non-determinism is bounded in the sense that the task execution characteristic must not violate the constraints set forth in the original master task graph representation.

A sink node (e.g., T_1) is said to be control dependent on a source node (e.g., T_0) if the runtime conditions present in the task associated with the source node determines whether or not the sink node can execute. We call such a source node (i.e., a node with directed control edges at its output, e.g., T_0) a control dependency source node, and such a sink node (i.e., a node with a directed control edge at its input, e.g., T_1 and T_3) a control dependency sink node. Unlike the directed data edge, which can connect any two nodes as long as the acyclic property of a graph is retained, two nodes T_i and T_j can be connected by a directed control edge $e_{ij}^c = (T_i, T_j)$ only if T_i is not a control dependency sink node and T_j is not a control dependency source node. This restriction guarantees that a node cannot have a directed control edge at both its input and output and will greatly

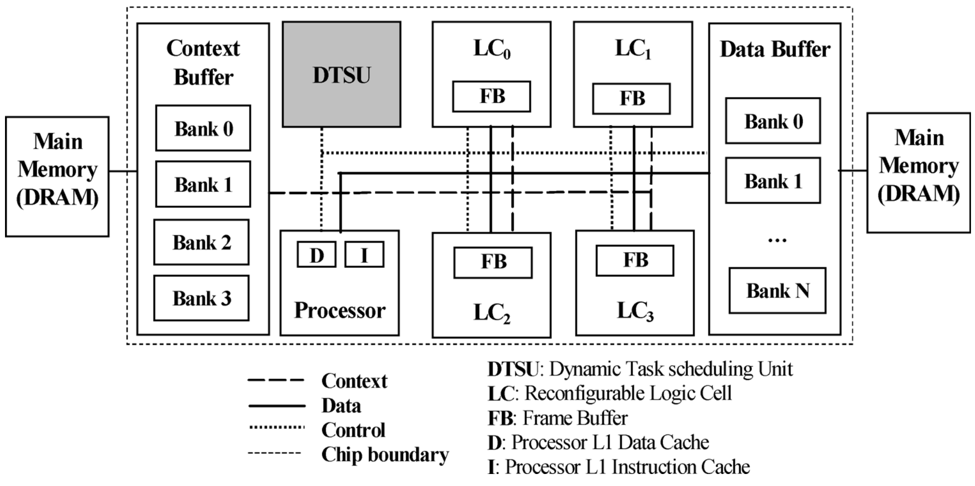


Fig. 2. Dynamically RSoC architecture.

reduce the hardware complexity of task scheduling microarchitecture.

It should be noted that this restriction will most likely increase the complexity of the application partitioning process. This effect, though, is believed to be minimal since it is always possible to encapsulate into a single task nondeterministic portions of the application. If the granularity of these portions is too large to reside in a single partition of the reconfigurable logic then the task can be divided into a chain of subtasks that are data flow dependent on one another but are control flow dependent on the same task that determines the run time condition necessary for execution. The proposed model supports traditional procedural programming methodologies and object oriented techniques that fully specify the meaning and purpose of the tasks.

In our model, the directed control edges are labeled in a probabilistic manner with a scalar value. This scalar value indicates the probability $p_j \in P$ of occurrence of the corresponding sink task in multiple task graph iterations (frames). A control condition represents a data dependent precedence relationship that when invalid prohibits the subsequent task from executing during a given instantiation of the task graph. So the necessary condition for such a sink node to execute is that the control condition is satisfied and the required data has been received. The probability parameter p_j, which is in general assumed to be independent of each other, can be viewed as being the average or expected proportion of time that such a control condition is satisfied.

The data and control dependencies define the execution condition for each task in a modified DAG. In our model, we assume that a sink node can be data dependent on multiple source nodes but is only control dependent on at most one source node. Furthermore, a data dependency from some source node is satisfied when either the source node has finished execution or when it has been determined that the source node will not execute based on its runtime determined control condition. For example, T_4's data dependence on T_1 is satisfied when either 1) T_1 finishes execution or 2) T_1 will not execute based on the run-time results obtained after T_0 completes its execution. This assumption is intended to avoid the situation where a task (e.g., T_4) is blocked forever because it waits for data from a source node (e.g., T_1) that will not execute.

The following definitions are used to describe the necessary conditions for the tasks to execute and to reconfigure.

Definition 1: A task is *ready-for-execution* when its control dependency (if present) and all of its data dependencies (if present) have been satisfied.

Definition 2: A task, which is not a control dependency sink node, is *ready-for-reconfiguration* when all its data dependency source tasks have started their execution.

It is important to note that use of the modified DAG does not limit its application to aperiodic task systems. This is because in many periodic task systems the modified DAG can be considered to represent an unrolled task system, where there is one major frame that is executed periodically.

C. Terminology

A processing element (PE) can be defined as an entity that is used for task execution. In our targeted RSoC architecture, it is either a CPU or an LC. In this model, we assume that only one task executes on a given LC at any given point in time (i.e., single-context devices). When a task finishes execution on a LC, a new task can be swapped in. If the task types of the two different tasks match, no LC context configuration occurs (i.e., this concept is named, reconfiguration context reuse). Otherwise, the bit-stream representing the new task needs to be downloaded into LC prior to new task being run. In order to minimize the reconfiguration overhead, we use a configuration prefetching technique that is explained in Section III.

III. Targeted RSoC Architectures

Fig. 2 gives an overview of the proposed RSoC architecture. It is comprised of a general-purpose embedded processor with L1 Data and Instruction caches, a number of LCs, a dynamic task scheduling unit (DTSU), and two on-chip L2 multi-bank memory subsystemsdata buffer and context bufferall implemented as a single chip. The embedded processor and LCs are the PEs of this chip. The dynamic task scheduling unit executes the dynamic task scheduling algorithm. This module directly

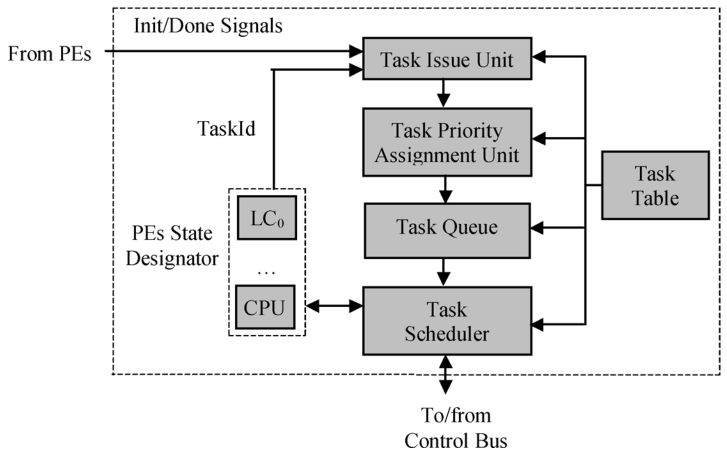


Fig. 3. DTSU.

controls the data prefetch units in the Data Buffer and context prefetch units in the LCs using the control (i.e., register) bus. These prefetch units implement the hardware-based data and configuration prefetch mechanisms that are used in our target architecture. This scheduler responds to interrupt signals from the LCs and processor to indicate the status of a given PE (these signals are not shown in Fig. 2).

The data buffer is a shared memory buffer that handles the data that is transferred between different PEs; the context buffer stores the reconfiguration contexts (i.e., bit-streams) used to configure the LC blocks. For the sake of simplicity, the crossbar style point-to-point link from each PE to Data Buffer is depicted as a simple Data bus in Fig. 2. In a similar manner, the point-to-point link from each LC to Context Buffer is depicted as a Context bus. In this scheme, each LC block can be independently reconfigured. Therefore, the proposed target architecture can support *multiple simultaneous reconfigurations*.

IV. Multitasking Microarchitecture and Scheduling Algorithm

Fig. 3 shows the internal microarchitecture of the dynamic task scheduler (DTSU). The PE's designator is a data structure that includes the pertinent assignment information of the tasks that are currently assigned to each PE (e.g., task ID, task type). The task issue unit (TIU) is responsible for dispatching tasks after the necessary conditions (i.e., ready-for-execution and ready-for-reconfiguration) have been met. The task priority assignment unit (TPAU) assigns a dynamic task priority to each dispatched task. The Run-time Task Scheduler performs the dynamic nonpreemptive scheduling algorithm, which assigns each task to a different PE, schedules the context/data fetch/prefetch and activates the task execution.

A. *Task Table*

The task table is a lookup table indexed by the task ID. Each entry of this table has six components: the ID, type and priority fields are used to hold the task ID, the task type, and the task priority, respectively. The Status 1 field holds the task execution status. The Status 2 field indicates whether a task is already loaded into a PE. The Par field indicates whether a task is executed on a LC or on the processor. This task placement information is determined by the particular task HW/SW partitioning strategy that is employed. The development of automated HW/SW partitioning algorithms is currently an active area of research [17]–[20] but discussion of these methodologies is beyond the scope of this paper.

B. *Task Issue Unit*

Fig. 4 illustrates the basic structure of task issue unit. Both the successor table and the predecessor table are $N \times N$ (N is the number of tasks) matrices indexed by the task ID. Each entry in the successor table is a N-bit vector that contains a task's data and control dependency sink information, while the predecessor table stores data dependency source information for each task. The test matrix is an $N \times M$ matrix ($M = 1+$ the maximum number of data dependency sources for each task in the task graph) indexed by the task ID. Each entry in this table is a M-bit vector: the lower ($M - 1$) bits represent the number of data dependency sources for a task; the most significant bit indicates whether the task is control dependent on a task or not. There are three N-bit task status registers: 1) the Done register is used to indicate which task finishes execution; 2) the Init register indicates which task has started execution; and 3) the Cont register indicates which task is enabled by its control dependency source.

When a task (excluding control dependency source tasks in order to avoid misprefetching context of its control dependency sink task) starts its execution, the Task Issue Unit receives the Init signal from the corresponding PE while also receiving this task's task Id from the PE's state designator module. At that moment, the Task Issue Unit checks whether some of the tasks have become ready for reconfiguration. A major objective of this hardware is to prefetch the task bit-stream ahead of time so that the LC reconfiguration overhead can be at least partially hidden. The dispatched task enters the task priority assignment unit with the precalculated static task priority. The task priority assignment unit then assigns a zero dynamic task priority (explained in Section IV-C) to the dispatched task before placing it in the task queue.

When a task finishes its execution, the task issue unit receives the Done signal from the associated PE while also receiving this task's task ID from the PE's state designator. If this task enables/

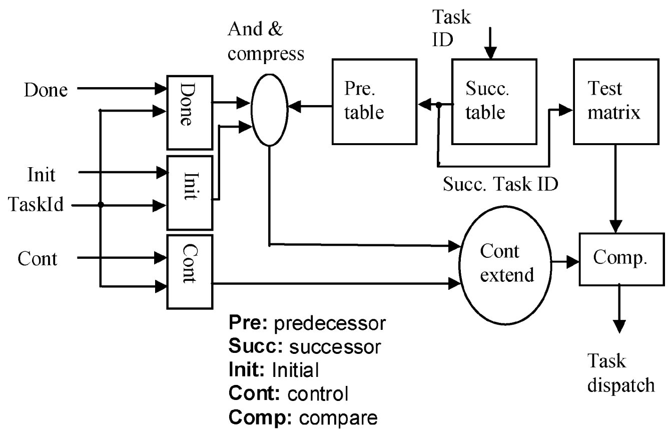


Fig. 4. Task issue unit.

disables some control dependency sink tasks, the task issue unit also receives their task IDs and execution conditions (true/false) from the PE's state designator. The task issue unit then checks the execution conditions of this finished task's data and control dependency sink tasks, and dispatches ready-for-execution tasks to the task priority assignment unit with the precalculated static task priority. The task priority assignment unit then assigns to it a dynamic task priority.

C. Task Priority Assignment Unit

The task priority assignment unit adds a fixed dynamic priority (i.e., the fixed dynamic task priority is chosen to be greater than the highest static task priority) to each ready-for-execution task's static priority part. The purpose is to guarantee that the each task that is ready-for-execution always has higher priority than any tasks that are ready-for-reconfiguration.

D. Task Queue

The proposed task queue is a modified shift register priority queue [21] used to store the issued tasks in a sorted order. The queue communicates with the task issue unit and the task scheduler and has four operations: delete (D), enqueue (E), dequeue (Q), and search (S). The delete and search operations are included to facilitate the operations of the task scheduler. The enqueue and dequeue operations conform to the standard queue methods.

E. Task Scheduler and Scheduling Algorithm

The PE state machine is shown in Fig. 5. We will focus on the LC state machine [see Fig. 5(a)]. The possible LC states and their explanations are as follows.

- *Idle:* LC is not occupied by any active task.
- *Reconfiguration:* LC is populating its context memory using the bit-stream of a new task.
- *Wait:* LC has finished reconfiguration, but it cannot move to Switch state since the task on it has not met execution condition.
- *Switch:* Similar to processor context switch. The task has its bit-stream loaded into the context memory of the host LC and satisfied execution condition.

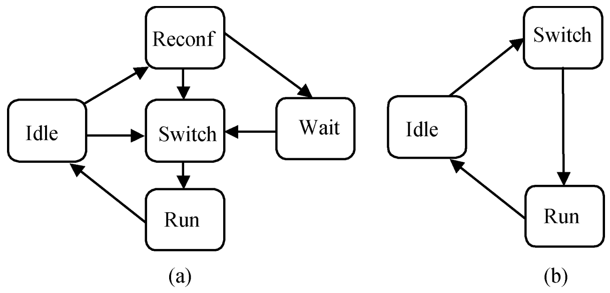


Fig. 5. LC and (a) processor (b) states machines.

- *Run:* LC is executing a task. The input data streams into the LC from the data buffer while the output data streams out of the LC to the data buffer.

The manner in which an LC switches from one state to another can be described as follows: upon leaving the Reconfiguration state, the LC proceeds to 1) the Switch state where it reads data that is being passed from its predecessor task(s). Then it switches to Run state, or 2) the Wait state until the data is ready. When an LC finishes executing, it proceeds to the Idle state. Based on the scheduling algorithm, the task scheduler decides whether to download a new bit-stream to an Idle LC or begin the execution of a new task (i.e., the LC turns to Switch state), which has the same task type as the previous one. The finite-state machine of the processor [see Fig. 5(b)] follows the conventional software process context switch mechanism.

The task scheduler unit is responsible for executing the dynamic task-scheduling algorithm. It schedules tasks from the task queue based on dynamic task scheduling algorithm, PEs' state, and the PE status (PE state, task ID, and task type) information from the PE state designator module. The objective of the scheduling algorithm is to minimize the application's execution time by employing a local optimization technique. More specifically, the scheduler attempts to reduce the LC reconfiguration overhead by reusing previous context or overlapping the currently scheduled LC's reconfiguration with task execution on other LCs.

The scheduler and scheduling algorithm are invoked whenever 1) an LC (processor) becomes idle or 2) the task queue enqueues a new task. If there is an idle LC, then the candidate task selection is accomplished by sending a search (S) operation to the task queue with 1) a dummy entry having desired task type (i.e., the type of just finished task on the idle LC) and 2) a search depth (i.e., the number of tasks being searched), which is a kind of look-ahead strategy into the task queue. This strategy represents the configuration reuse concept, which is used by the scheduling algorithm to reduce the run-time reconfiguration overhead. On the other hand, if the search returns nothing, the task entry whose bit-stream is not present in the LCs will be given higher priority. This idea is intuitive since we'd like to have as many task contexts as possible present in the PEs so that in subsequent scheduling of tasks there is better chance to reuse some of them. Processor scheduling is skipped here because: 1) it is not the focus of this paper and 2) this topic is adequately covered elsewhere [22].

TABLE I
SIMULATION CLASSIFICATION FOR NONDETERMINISTIC TASK SYSTEMS

Simulation Case	**Degree of Nondeterminism**	**Task Execution Time**	**Intertask Dependencies**
1	Zero	Deterministic	Fixed
2	Low	Probabilistic	Fixed
3	Medium	Deterministic	Control/Data Dependencies
4	High	Probabilistic	Control/Data Dependencies

Increasing degree of nondeterminism ↓

V. PERFORMANCE EVALUATION OF THE DYNAMIC TASK SCHEDULING ALGORITHM

In this section, we present simulation results that are produced by running the proposed dynamic task scheduling algorithm on a set of synthetic task systems that are generated by TGFF [23]. Previous work [13], [14] that applies dynamic task scheduling to dynamically reconfigurable hybrid architectures does not target nondeterministic task systems. As a result, there is no known body of work by other researchers with which the proposed dynamic task scheduling algorithm's performance can be directly compared. The simulations discussed below attempt to determine how well the dynamic scheduling solutions compare to those produced by static scheduling algorithms when the task system generation and architectural parameters are varied. To remove the influence of the particular offline HW/SW partitioning approach that may be employed it is assumed that all the tasks in the system have been designated to execute on the LC resources.

A. Simulation Classification

The TGFF only generates static data flow task graph. In order to mimic nondeterministic task systems, a run-time parameter, π, called the degree of dynamism of the task system, has been introduced in the simulation. This parameter indicates the proportion of tasks (i.e., control dependence sinks) that will not appear in a particular instantiation (i.e., a major frame) of a task graph due to run-time execution condition. Higher values of π will result in task graphs that have progressively more variability in precedence relationship characteristics. Based on π and nondeterministic task execution time, we have divided the set of simulations into four cases as shown in Table I.

B. Assumptions in Static Scheduling Techniques

With the simulations being classified into four main cases that are based on the degree of non-determinism in task systems, we are able to compare the quality of the schedules that are produced by our dynamic scheduling algorithm (DS) with that which is produced by the multiheuristic parallel-based scheduling algorithm (MHPSA) (SS) [20]. The MHPSA simultaneously employs genetic algorithm, particle swarm, and simulated annealing techniques to arrive at a near optimal solution in polynomial time. It should be noted that the run time and computational complexity of the SS is many orders of magnitude greater than the DS and is not conducive to implementation in reconfigurable hardware. In order to achieve fair comparison, it is assumed that the SS technique can only use estimated execution information when making scheduling decision. Specifically, a global execution order and assignment of each task is first determined using average task execution times and a deterministic task system (i.e., assuming only static data flow existing), then a schedule is made for each frame using this global information with the exact task execution time and execution condition.

C. Task System Configuration

Table II illustrates eight sets of task systems that were used to represent two types of applications: task systems with low parallelism (i.e., Type I applications) and task systems with high parallelism (i.e., Type II applications). For each set, five task system instances were generated. So the Critical Path and Degree of Parallelism columns report average numbers over five instances. In all these sets, each task graph has an average of 25 tasks with the number of task types being set at 5 or 20, respectively, and the number of task successors being set at 2 or 5, respectively. The Type I application was synthetically generated by the TGFF utility, by setting the number of task graphs parameter equal to one. The Type II application was synthetically generated by setting this parameter to three, as indicated in the table. For LCs, the task average execution time is assumed to be 3000 normalized Units of Time (UT), with a variability of 2000.

D. Design of Simulation

The simulation factors that are considered include:

- *Simulation Case:* 4 (see Table I).
- *Number of Task Systems:* 40 task systems in each simulation case (5 instances/per set $\times$ 8 sets).
- *Architectural Parameters:* the LC reconfiguration time (3 levels), # of LCs (3 levels), and scheduler search depth (15 levels).
- π: 2 levels 0.1 and 0.2 for simulation case 3/4. 1 level (i.e., 0) for simulation case 1/2, where 0 represents all tasks will be executed in a task graph.
- *# of Frames:* for simulation case 3/4, the results from dynamic task scheduling are averaged over 40 frames, while for simulation case 1/2, only 1 frame is required.

E. Approximate Lower-Bound Solutions for Scheduling

Definition 3: Given the critical path (CP) [16] of a task graph G, the number of task types (TT) of G, the LC reconfiguration time (RT) and the number of LCs (NL) in a reconfigurable architecture RP, the approximate lower-bound value (ψ) for scheduling G onto RP is

$$\Psi = \frac{(CP + TT * RT)}{NL}. \tag{1}$$

TABLE II
INPUT TASK SYSTEM CONFIGURATION

Type of Task System	Task System Set	# of Task Graphs	# of Task Types	# of Task Successors	Critical Path (UT)	Degree of Parallelism
Task system with low parallelism	TS1	1	5	2	32425.2	2.2
	TS2	1	5	5	18631.8	3.8
	TS3	1	20	2	31375.6	2.3
	TS4	1	20	5	22089.6	3.5
Task system with high parallelism	TS5	3	5	2	34714.2	6.0
	TS6	3	5	5	21537.2	10.0
	TS7	3	20	2	36138.4	5.9
	TS8	3	20	5	24522.6	9.3

TABLE III
COMPARISON OF DS AND SS FOR TYPE I TASK SYSTEMS (SIMULATION CASE 1)

# of LCs	RT	TS1		TS2		TS3		TS4		Average over Row		DS/SS
		DS	SS	DS	SS	DS	SS	DS	SS	DS	SS	
3	1/4x	1.08	1.04	1.22	1.18	1.05	1.01	1.19	1.16	1.14	1.10	1.04
	1x	1.37	1.12	1.41	1.25	1.18	1.04	1.30	1.17	1.32	1.14	1.15
	4x	2.16	1.41	2.11	1.36	1.63	1.14	1.58	1.17	1.87	1.27	1.47
4	1/4x	1.02	1.00	1.29	1.21	0.98	0.95	1.19	1.11	1.12	1.07	1.05
	1x	1.30	1.01	1.58	1.28	1.09	0.93	1.28	1.13	1.31	1.09	1.21
	4x	2.19	1.19	2.25	1.37	1.78	1.05	1.70	1.14	1.98	1.19	1.67
5	1/4x	1.01	1.00	1.19	1.13	0.97	0.96	1.07	1.00	1.06	1.02	1.04
	1x	1.26	1.00	1.55	1.23	1.05	0.89	1.20	1.03	1.27	1.04	1.22
	4x	2.27	1.06	2.42	1.29	1.95	1.00	1.74	1.10	2.10	1.11	1.88
Average over column		1.52	1.09	1.67	1.26	1.30	0.99	1.36	1.11	1.46	1.11	1.31

TABLE IV
COMPARISON OF DS AND SS FOR TYPE II TASK SYSTEMS (SIMULATION CASE 1)

# of LCs	RT	TS5		TS6		TS7		TS8		Average over Row		DS/SS
		DS	SS	DS	SS	DS	SS	DS	SS	DS	SS	
3	1/4x	1.09	1.12	1.05	1.08	1.13	1.13	1.08	1.08	1.09	1.10	0.99
	1x	1.30	1.27	1.16	1.14	1.38	1.32	1.27	1.22	1.28	1.24	1.03
	4x	1.90	1.47	1.53	1.28	1.85	1.50	1.67	1.29	1.74	1.38	1.25
4	1/4x	1.13	1.15	1.07	1.10	1.15	1.14	1.11	1.10	1.11	1.12	0.99
	1x	1.38	1.28	1.19	1.13	1.42	1.35	1.29	1.21	1.32	1.24	1.06
	4x	1.86	1.43	1.52	1.20	1.97	1.51	1.66	1.29	1.75	1.36	1.29
5	1/4x	1.17	1.17	1.10	1.12	1.20	1.17	1.12	1.12	1.15	1.15	1.00
	1x	1.46	1.28	1.26	1.18	1.46	1.38	1.29	1.24	1.37	1.27	1.08
	4x	1.71	1.28	1.73	1.23	1.99	1.51	1.70	1.34	1.78	1.34	1.33
Average over column		1.44	1.27	1.29	1.16	1.50	1.33	1.35	1.21	1.40	1.24	1.12

All simulation results are normalized to Ψ before recorded.

F. Results and Discussion

In this section, we empirically compare the proposed dynamic tasks scheduling algorithm with the reference static scheduling technique for each simulation case.

1) Simulation Case 1: Simulation Case 1 corresponds to task systems that contain only static data flow task graphs with deterministic task execution times. In other words, it is assumed that SS uses complete, exact execution information of task system.

Tables III and IV show DS versus SS for Type I task systems (TS1TS4) and for Type II task systems (TS5TS8), respectively. The Average over Row column reports the normalized schedules over four task systems with # of LCs and RT being fixed. The Average over Column row reports the normalized schedules over all possible hardware configurations with the task system being fixed. DS/SS reports the ratio of normalized DS to normalized SS for each possible hardware configuration.

As can be seen from Table III, for Type I task systems SS performs almost as well as the approximate lower-bound value Ψ (average 1.11 times as Ψ which can be read from column 12 of last row) while DS is slower than SS in each hardware configuration (see Average over Row column); for Type II task systems, Table IV shows that SS performs better than the DS in all hardware configurations except when RT is small (i.e., $RT = 1/4\times$), in which the DS outperforms SS by a neglectable margin ($DS/SS = 0.99$) for the case where the number of LCs is equal to 3 or 4. Overall, the DS underperforms SS by 22% in simulation case 1 (i.e., $(1.46 + 1.40 - 1.11 - 1.24)/(1.11 + 1.24)$). This conclusion is not surprising though since: 1) the SS uses

TABLE V
COMPARISON OF DS AND SS FOR TYPE I TASK SYSTEMS (SIMULATION CASE 3)

# of LCs	RT	π	TS1		TS2		TS3		TS4		Average over Row		DS/SS
			DS	SS	DS	SS	DS	SS	DS	SS	DS	SS	
3	1/4x	0.1	1.11	1.16	1.18	1.26	1.08	1.11	1.17	1.22	1.13	1.19	0.95
		0.2	1.19	1.28	1.19	1.32	1.11	1.20	1.16	1.25	1.17	1.26	0.92
	1x	0.1	1.38	1.27	1.34	1.34	1.17	1.15	1.25	1.24	1.29	1.25	1.03
		0.2	1.47	1.41	1.37	1.43	1.20	1.25	1.23	1.30	1.32	1.35	0.98
	4x	0.1	2.02	1.52	1.83	1.41	1.52	1.21	1.44	1.24	1.70	1.35	1.26
		0.2	2.01	1.64	1.74	1.47	1.48	1.31	1.38	1.30	1.65	1.43	1.16
4	1/4x	0.1	1.03	1.09	1.23	1.27	1.00	1.03	1.17	1.21	1.11	1.15	0.96
		0.2	1.12	1.17	1.25	1.33	1.04	1.11	1.23	1.29	1.16	1.22	0.95
	1x	0.1	1.33	1.13	1.48	1.39	1.11	1.03	1.28	1.23	1.30	1.20	1.09
		0.2	1.42	1.25	1.52	1.48	1.17	1.12	1.32	1.33	1.36	1.29	1.05
	4x	0.1	2.03	1.31	1.98	1.41	1.68	1.13	1.55	1.21	1.81	1.27	1.43
		0.2	2.05	1.42	1.94	1.47	1.65	1.23	1.54	1.29	1.79	1.35	1.33
5	1/4x	0.1	1.00	1.06	1.15	1.19	0.97	1.01	1.07	1.10	1.05	1.09	0.96
		0.2	1.09	1.12	1.22	1.27	1.02	1.07	1.17	1.21	1.12	1.17	0.96
	1x	0.1	1.29	1.09	1.48	1.31	1.10	0.97	1.20	1.14	1.27	1.13	1.12
		0.2	1.41	1.19	1.57	1.42	1.17	1.05	1.30	1.28	1.36	1.24	1.10
	4x	0.1	2.11	1.16	2.16	1.35	1.82	1.08	1.61	1.19	1.93	1.20	1.61
		0.2	2.19	1.26	2.17	1.41	1.82	1.17	1.63	1.29	1.95	1.28	1.52
Average for π=0.1 (over column)			1.48	1.20	1.54	1.32	1.27	1.08	1.30	1.20	1.40	1.20	1.16
Average for π=0.2 (over column)			1.55	1.31	1.55	1.40	1.30	1.17	1.33	1.28	1.43	1.29	1.11

TABLE VI
COMPARISON OF DS AND SS FOR TYPE I TASK SYSTEMS (SIMULATION CASE 3)

# of LCs	RT	π	TS5		TS6		TS7		TS8		Average over Row		DS/SS
			DS	SS	DS	SS	DS	SS	DS	SS	DS	SS	
3	1/4x	0.1	1.07	1.21	1.04	1.19	1.10	1.18	1.08	1.15	1.07	1.18	0.91
		0.2	1.07	1.25	1.04	1.23	1.09	1.22	1.06	1.19	1.06	1.22	0.87
	1x	0.1	1.26	1.46	1.15	1.41	1.31	1.46	1.24	1.35	1.24	1.42	0.87
		0.2	1.24	1.56	1.15	1.51	1.28	1.52	1.21	1.44	1.22	1.51	0.81
	4x	0.1	1.74	1.78	1.44	1.55	1.71	1.62	1.55	1.51	1.61	1.61	1.00
		0.2	1.69	1.95	1.42	1.66	1.62	1.71	1.47	1.58	1.55	1.72	0.90
4	1/4x	0.1	1.12	1.25	1.07	1.20	1.13	1.21	1.09	1.19	1.10	1.21	0.91
		0.2	1.12	1.29	1.07	1.24	1.12	1.25	1.08	1.23	1.10	1.25	0.88
	1x	0.1	1.34	1.52	1.19	1.46	1.34	1.46	1.25	1.40	1.28	1.46	0.88
		0.2	1.32	1.61	1.19	1.54	1.31	1.53	1.23	1.47	1.26	1.54	0.82
	4x	0.1	1.77	1.73	1.47	1.49	1.75	1.67	1.58	1.58	1.64	1.62	1.02
		0.2	1.73	1.87	1.48	1.64	1.66	1.75	1.50	1.64	1.59	1.73	0.92
5	1/4x	0.1	1.17	1.31	1.11	1.23	1.16	1.25	1.11	1.21	1.14	1.25	0.91
		0.2	1.17	1.34	1.11	1.27	1.15	1.30	1.11	1.25	1.13	1.29	0.88
	1x	0.1	1.44	1.54	1.26	1.47	1.38	1.52	1.27	1.43	1.34	1.49	0.90
		0.2	1.44	1.62	1.27	1.55	1.36	1.59	1.24	1.50	1.33	1.57	0.85
	4x	0.1	1.59	1.51	1.60	1.41	1.82	1.70	1.60	1.60	1.65	1.56	1.06
		0.2	1.56	1.62	1.56	1.57	1.73	1.78	1.52	1.69	1.59	1.67	0.96
Average for π=0.1 (over column)			1.39	1.48	1.26	1.38	1.41	1.45	1.31	1.38	1.34	1.42	0.94
Average for π=0.2 (over column)			1.37	1.57	1.25	1.47	1.37	1.52	1.27	1.44	1.32	1.50	0.88
Average over column			1.38	1.52	1.26	1.42	1.39	1.49	1.29	1.41	1.33	1.46	0.91

computationally expensive heuristics and 2) for case 1 simulation, the SS uses complete and exact execution information to schedule tasks.

Tables III and IV show that the performance of SS generally declines as the problem size increases: i.e., from 1.11 (column 12 of last row in Table III) for Type I task systems (which have 75 tasks) to 1.24 (column 12 of last row in Table IV) for Type II task systems (which have 25 tasks). This gives us the following observation.

2) Observation 1: As the problem size increases, the performance of SS tends to decrease.

On the other hand, it seems that the performance of DS does not follow this trend. This can be observed from the ratio of 1.46 (column 11 of last row in Table III) for Type I task systems to 1.40 (column 11 of last row in Table IV) for Type II task systems. So the second observation is as follows.

3) Observation 2: As the problem size increases, the performance of the DS tends to remain constant.

Combining Observations 1 and 2, we can make the following conclusion.

4) Conclusion 1: The performance differences between the DS and SS decreases when the problem size increases.

Calculations indicate that overall, for Type I task systems, the DS is 35% (i.e., $(1.46 - 1.11)/1.11)$) slower than the SS, while for Type II task systems the DS is only 13% (i.e., $(1.40 - 1.24)/1.24)$) slower than the SS.

We shall now examine the aforementioned observations and conclusion in the other three simulation cases.

5) Simulation Case 2: The results from Simulation Case 2 are very close to those in Case 1. This is because from Table I it can be seen that the task system in Case 2 is almost the same as the task system in Case 1 for the SS. In addition, it appears that Observations 1 and 2 and Conclusion 1 are still valid.

6) Simulation Case 3: Simulation Case 3 utilizes task systems that contain task graphs with both data and control dependencies, but the task execution time is deterministic. In other words, not all tasks are executed on a frame basis due to the run-time execution condition but their execution time keeps constant.

Tables V and VI show DS versus SS for Type I task systems and for Type II task systems, respectively. It can be observed that Observation 7.1 and Observation 7.2 still hold. Even better, the performance of DS tends to improve as problem size increases. For example, the DS normalized average schedules are 1.40 ($\pi = 0.1$) and 1.43 ($\pi = 0.2$) for Type I task systems, respectively, but these numbers improve to 1.34 and 1.32 for Type II task systems, respectively.

Finally, an important conclusion is that DS outperforms SS for Type II task systems, and this improvement increases as π increases, i.e., from a ratio of 0.94 to 0.88. Therefore, we can make another conclusion.

7) Conclusion 2: The DS tends to outperform SS when both problem size and the degree of dynamism increase.

8) Simulation Case 4: Just as the results from Case 2 are very close to those from Case 1, the results from Case 4 are also very close to those from Case 3. In other words, the same properties hold for both cases.

9) Conclusion: In summary, we can conclude the following.

- The performance of the DS does not degrade as the complexity of problem increases, while the performance of reference SS does decline as the complexity of problem increases.
- The DS tends to outperform the SS when both task system complexity and degree of dynamism increases.
- The DS has very low algorithmic complexity, therefore, runs very fast, while the SS is very time intensive.

VI. Conclusion

In this paper, we have presented a dynamic task scheduling algorithm that is capable of scheduling nondeterministic systems of tasks onto RSoC hardware environments. The incorporation of control dependencies allows for the modeling and simulation of nondeterministic task structures, which is the primary motivation for employing dynamic scheduling techniques in reconfigurable hardware. A major contribution of this work has been the modeling and simulation of an expanded microarchitecture that can support such non-determinism.

To study the effectiveness of the proposed dynamic scheduler, traditional software-based simulation techniques have been utilized and a simulation framework has been developed. A large number of simulations were performed. The results demonstrate many interesting and viable features of the proposed dynamic scheduling approach while indicating considerable promise for its desired application domain.

References

[1] Xilinx Inc., San Jose, CA, Vertex-4 family overview, 2007. [Online]. Available: http://www.xilinx.com /bvdocs/publications/ds112. pdf

[2] D. Andrews, D. Niehaus, and P. Ashenden, "Programming models for hybrid CPU/FPGA chips," *IEEE Computer*, vol. 37, no. 1, pp. 118–120, Jan. 2004.

[3] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY: Freeman, 1979.

[4] D. Chang and M. Marek-Sadowska, "Partitioning sequential circuits on dynamically reconfigurable FPGAs," *IEEE Trans. Comput.*, vol. 48, no. 6, pp. 565–578, Jun. 1999.

[5] R. P. Dick and N. K. Jha, "CORDS: Hardware-software codesign of reconfigurable real-time distributed embedded systems," in *Proc. Int. Conf. Comput.-Aided Des.*, 1998, pp. 62–68.

[6] R. Maestre, F. J. Kurdahi, M. Fernandez, R. Hermida, N. Bagherzadeh, and H. Singh, "A framework for reconfigurable computing: Task scheduling and context managementA summary," *IEEE Circuits Syst. Mag.*, vol. 2, no. 4, pp. 48–51, 2001.

[7] L. Shang and N. K. Jha, "Hardware-software codesign of low power real-time distributed embedded systems with dynamically reconfigurable FPGAs," in *Proc. 15th IEEE Int. Conf. VLSI Des.*, 2002, pp. 345–352.

[8] G. Wigley and D. Kearney, "The first real operating system for reconfigurable computers," in *Proc. 6th Australasian Comput. Syst. Arch. Conf.*, 2001, pp. 130–137.

[9] H. Walder and M. Platzner, "Non-preemptive multitasking on FPGAs: Task placement and footprint transform," in *Proc. Eng. Reconfig. Syst. Arch.*, 2002, pp. 24–30.

[10] R. P. Dick, G. Lakshminarayana, A. Raghunathan, and N. K. Jha, "Power analysis of embedded operating systems," in *Proc. DAC*, 2000, pp. 312–315.

[11] P. Kuacharoen, M. A. Shalan, and V. J. Mooney, "A configurable hardware scheduler for real-time systems," in *Proc. Eng. Reconfig. Syst. Arch.*, 2003, pp. 96–101.

[12] S. Isaacson and D. Wilde, "The task-resource matrix: Control for a distributed reconfigurable multi-processor hardware RTOS," in *Proc. Eng. Reconfig. Syst. Arch.*, 2004, pp. 130–136.

[13] J. Noguera and R. M. Badia, "Multitasking on reconfigurable architectures: Microarchitecture support and dynamic scheduling," *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 2, pp. 385–406, 2004.

[14] Z. Pan, J. Noguera, and B. E. Wells, "Improved microarchitecture support for dynamic task scheduling on reconfigurable architectures," in *Proc. Eng. Reconfig. Syst. Algorithms*, 2005, pp. 182–188.

[15] Z. Pan, B. E. Wells, and J. Noguera, "Microarchitecture support for the dynamic scheduling of task systems with data control dependencies on reconfigurable architectures," in *Proc. Int. Symp. Perform. Eval. Comput. Telecommun. Syst.*, 2005, pp. 499–508.

[16] B. P. Dave, G. Lakshminarayana, and N. K. Jha, "COSYN: Hardware-software co-synthesis of embedded systems," in *Proc. 34th Annu. Conf. Des. Autom.*, 1997, pp. 703–708.

[17] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," *IEEE Des. Test Comput.*, vol. 10, no. 4, pp. 64–75, 1993.

[18] R. K. Gupta, "Hardware-software cosynthesis of digital systems," Ph.D. dissertation, Dept. Elect. Eng., Stanford Univ., Stanford, CA, 1994.

[19] J. Noguera and R. M. Badia, "HW/SW codesign techniques for dynamically reconfigurable architectures," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 10, no. 4, pp. 399–415, Aug. 2002.

[20] Z. Pan, "Hardware supported task scheduling on dynamically reconfigurable SOC architectures," Ph.D. dissertation, Dept. Elect. Comput. Eng., Univ. Alabama in Huntsville, Huntsville, 2006.

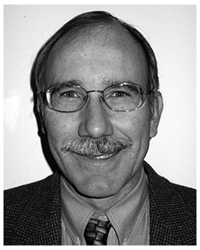
[21] S. Moon, J. Rexford, and K. G. Shin, "Scalable hardware priority queue architectures for high-speed packet switches," *IEEE Trans. Comput.*, vol. 49, pp. 1215–1227, 2000.

[22] N. Nutt, *Operating Systems, A Modern Perspective*. Boston, MA: Addison-Wesley, 1997, pp. 129–150.

[23] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: Task graph for free," in *Proc. Int.Workshop Hardw./Softw. Codes.*, 1998, pp. 97–101.

Zexin Pan received the B.E. degree in automatic control from Xi'an Jiaotong University, Xi'an, Shanxi, P. R. China, in 1990, and the M.S. and Ph.D. degrees in computer engineering from the University of Alabama in Huntsville, Huntsville, in 2002 and 2006, respectively.

His research interests include reconfigurable computing, HW/SW codesign, and system-on-chip design techniques.



B. Earl Wells (M'79) received the B.S., M.S., and Ph.D. degrees in electrical engineering from the University of Alabama, Tuscaloosa, in 1983, 1988, and 1992, respectively.

He is currently a Professor with the Department of Electrical and Computer Engineering, University of Alabama in Huntsville, Huntsville. His research interests include reconfigurable hardware systems and parallel/distributed processing architectures and applications.