An Automated Scheduling and Partitioning Algorithm for Scalable Reconfigurable Computing Systems

Casey Reardon, Alan D. George, Greg Stitt, and Herman Lam
NSF Center for High-Performance Reconfigurable Computing (CHREC)
ECE Department, University of Florida, Gainesville, FL
{reardon,george,stitt,lam}@chrec.org

Abstract—**As reconfigurable computing (RC) platforms are becoming increasingly large-scale and heterogeneous, efficiently scheduling and partitioning applications on these platforms is a growing challenge. While previous approaches support scheduling and HW/SW partitioning for various reconfigurable architectures, none of these approaches have been designed or shown to support large-scale or multi-node RC systems. This paper presents an algorithm that to our knowledge is the first designed to support automated scheduling and HW/SW partitioning on large-scale RC systems. The algorithm presented here uses a two-stage process. The first stage generates an initial schedule using novel list-based scheduling extensions, which feeds an iterative cycle in stage two to search for moves to further optimize the schedule. Initial results show the algorithm can efficiently produce quality schedules for parallel applications on a large-scale RC platform.**

I. Introduction

As platforms use increasing amounts of parallelism and heterogeneity to more efficiently meet the computational demands of modern applications, reconfigurable computing (RC) is becoming recognized as an important and viable paradigm for high-performance and embedded computing. Large-scale RC systems, which may feature a mix of hundreds or thousands of processing devices such as microprocessors and FPGAs, can leverage system-level concepts from conventional high-performance computing and accommodate hardware reconfigurability to efficiently accelerate many operations that would otherwise be performed in software.

The emergence of RC in increasingly parallel high-performance and embedded computing systems raises many challenges. One challenge that designers must confront is the problem of HW/SW partitioning and scheduling complex parallel applications onto a large-scale heterogeneous system. While manually determining and specifying an optimal schedule may be a straightforward process for simple, embarrassingly parallel applications, it can be difficult and time-consuming for complex applications with many parallel tasks on large systems. The challenge is even greater when considering a reconfigurable heterogeneous system, such as an RC platform containing FPGAs and microprocessors, where each type of device is likely to provide a different level of support and performance for a given task. Additionally, users must also decide when to reconfigure the platform's RC devices to support a new set of tasks. To address these challenges, automated HW/SW partitioning and scheduling techniques can be employed to suggest an optimal or near-optimal mapping (i.e., assignment of tasks to specific instances of a resource) of the parallel algorithm onto the execution platform in terms of performance (or other metric(s)). Algorithms based on methods such as Integer Linear Programming (ILP) can even guarantee an optimal schedule, but their computational complexity makes these algorithms undesirable for many situations, especially with large-scale systems.

While previous approaches have addressed various scheduling and HW/SW partitioning problems, they have typically focused on either parallel SW-based systems or HW/SW partitioning for ASICs or small-scale RC architectures. As a result, existing research is lacking in the area of scheduling and HW/SW partitioning for systems that are large-scale, heterogeneous, and dynamically reconfigurable. Therefore, in this paper we present an algorithm designed to support automated scheduling and partitioning on large-scale RC systems. The algorithm presented in this paper, which extends existing RC-based scheduling and partitioning techniques to efficiently support large-scale systems, is divided into two stages. In the first stage, a novel priority and allocation scheme to extend existing list-based scheduling (LBS) techniques is used to generate an initial schedule. In the second stage, a modified Kernighan-Lin heuristic [1] is used that iteratively analyzes a set of moves for each unfixed node in the task graph, implementing the best move after each iteration to move the schedule towards a better solution. The iterative process continues until all tasks have participated in a move or until no more moves produce an improved schedule. Two case studies are presented demonstrating that this algorithm performs well for large-scale RC applications and in a fraction of the time compared to our baseline scheduling algorithm, which employs simulated annealing.

The remainder of this paper is organized as follows. Section II discusses related research in automated scheduling and HW/SW partitioning. Basic problem definitions and assumptions are summarized in Section III. In Section IV, a detailed overview of the algorithm is presented. Case studies testing and demonstrating the effectiveness of the scheduling algorithm are presented in Section V. Finally, conclusions and future work are discussed in Section VI.

II. Related research

Many previous algorithms have been developed for HW/SW partitioning [2], [3] and scheduling on parallel heterogeneous systems [4]. Meanwhile, fewer research projects have focused on automated scheduling and partitioning in RC environments, though several approaches have been proposed and are summarized in the remainder of this section.

A pair of algorithms has been developed for scheduling on reconfigurable architectures with hard real-time constraints. An evolutionary scheduling algorithm built into the CORDS co-synthesis framework for distributed real-time reconfigurable embedded systems is discussed in [5]. An integrated HW/SW partitioning and scheduling algorithm for co-synthesis of hard real-time RC systems is described in [6], which uses the Latest Deadline First heuristic for software tasks and then moves tasks to an FPGA co-processor (adding FPGAs to the architecture as needed) until a feasible schedule exists that meets all real-time deadlines.

Stitt proposes a pair of efficient sub-heuristics for HW/SW partitioning and multi-version implementation exploration of reconfigurable hardware circuits which are specialized for support of balanced (i.e., execution time is evenly distributed throughout each region of the algorithm) and unbalanced applications [7]. Li et al. describe a HW/SW partitioning approach for embedded reconfigurable architectures developed for the Nimble framework [8]. The algorithm identifies regions of interest in the task graph, and exhaustively searches those regions for a locally optimal partitioning to produce near-optimal solutions for the global system.

Banerjee et al. [9] introduces an iterative-based algorithm to determine scheduling and module placement of tasks in a partial-reconfiguration environment, consisting of a single a CPU and FPGA with support for configuration pre-fetching.

Another iterative algorithm, which employs a strategy similar to the Kernighan-Lin heuristic, is proposed in [10] that combines HW/SW partitioning, hardware design-space exploration (DSE), and scheduling. MAGELLAN extends this work by implementing DSE for loop-unrolling and defines additional types of moves for scheduling control-dataflow task graphs [11]. As with all of the existing approaches discussed in this section, their algorithm assumes a single-node architecture with few devices. But since their algorithm supports scheduling, partitioning and automated DSE of RC applications with loop-unrolling and dynamic reconfiguration (characteristics we believe to be important in complex large-scale RC applications), the algorithm presented in this paper builds upon the work in [10], [11] to support large-scale RC systems by leveraging the iterative DSE exploration process they use while adding moves for combining tasks into reconfigurable hardware cores and introducing a new LBS heuristic scheme in the initial scheduling phase.

III. Problem Definition and Assumptions

Before describing our scheduling algorithm, the goals and basic assumptions applied throughout this research need to be defined. The primary goal is to design an efficient automated scheduling and partitioning algorithm that supports large-scale RC applications and platforms. The algorithm's primary task is to minimize the completion time of the entire application on the given platform, and to use as few architecture resources as needed to achieve the minimal schedule length in order to help limit the system's power consumption. The remainder of this section defines basic architecture and algorithm assumptions and input variables used for this research.

The platform architecture is assumed to consist of one or more uniform, fully connected nodes. Each node contains one or more CPUs and one or more FPGAs, as defined by the user's architecture model. Parameters are assigned to each FPGA to characterize the logic and memory capacities of each device, as well as the reconfiguration time. Communication between devices within the same node assumes one of three rates. The term $R_{CPU-CPU}$ specifies the communication rate between any two CPUs in the same node, $R_{FPGA-FPGA}$ specifies the communication rate between any two FPGAs in the same node, and $R_{CPU-FPGA}$ specifies the communication rate from any CPU to any FPGA (or vice versa) in the same node. Communication between any two nodes is uniform and approximated by the inter-node communication rate R_{inter}. The communication time between tasks within the same device is assumed to be negligible compared to other communication times. Future extensions to the algorithm could be implemented to support additional architectural characteristics, such as separate rates for reads and writes between a CPU and FPGA, or platforms whose nodes are arbitrarily connected. Furthermore, while this paper focuses on platforms that feature homogenous CPUs and FPGAs, the approach described here can naturally be applied to platforms with more than two types of processors, including processor types beyond CPUs and FPGAs.

The application is assumed to be specified as a directed acyclic task graph $G(V, E)$, where the vertices V represent tasks and edges E represent data dependencies between tasks. For each task V_i, $T_P(V_i)$ defines the execution time of task V_i on processor type P (i.e., CPU or FPGA). Such values can typically be ascertained from profiling tools, synthesis reports, or predictive modeling techniques, and are commonly assumed inputs to scheduling algorithms. $U_{P,log}(V_i)$ and $U_{P,mem}(V_i)$ define the baseline logic and memory utilization respectively of the task V_i for processor type P. Currently, the utilizations are only used determine if resource constraints are violated; potential performance degradations from memory or logic contention within a device are not considered. A utilization of 1 is assumed for all tasks with regard to software-based CPUs (i.e., only one task may execute at a time on any CPU, each core of a multi-core CPU is treated as a separate CPU). An instance of $T_P(V_i) = null$ signifies that execution of task V_i is not supported on processor type P. Each task also contains a parameter defining the degree of loop unrolling (or parallelization) that can be exploited for concurrent execution on one or more devices. An unrolled task may span multiple devices and multiple nodes, as such tasks often comprise the bulk of an application's execution time. Tasks that do not support any loop unrolling are assigned a degree of 1. When unrolled, the task's execution time (prior to communication considerations) is assumed to decrease linearly, and utilization is assumed to increase linearly, proportional to the degree of unrolling. These assumptions are generally valid for highly parallel applications which are the focus of this research. Finally, for each edge in the graph $E_{i,j}$ that connects task

```
//Preprocessing
for each Task $V_i$ in Task Graph do
    Calculate $DAW$ for $V_i$
    Calculate $CW$ for $V_i$
    Set $V_i$ as $Unfixed$
end for

//Initial LBS Stage
while List $UnscheduledTasks$ is not empty do
    $V_{hp} \leftarrow$ Unscheduled Task with Highest Priority
    Allocate Resources and Map $V_{hp}$ to Platform
    Update List $UnscheduledTasks$
    for each Task $V_j$ in $UnscheduledTasks$ do
        Update $CW$ for $V_j$
    end for
end while

//Iterative DSE Cycle Stage
while List $UnfixedTasks$ is not empty do
    for each Task $V_k$ in List $UnfixedTasks$ do
        Evaluate All Moves For Task $V_k$
    end for
    if Schedule Improvement From $BestMove > 0$ then
        Apply $BestMove$ to Current Schedule
        Set Task $V_{BestMove}$ as $Fixed$
        Update List $UnfixedTasks$
    else
        Exit DSE Cycle
    end if
end while
return Final Schedule
```

Fig. 1. Pseudocode for General Algorithm

V_i to V_j, a cost value is attached which defines the amount of data transferred from V_i to V_j.

IV. Algorithm Overview

This section describes our algorithm for automated scheduling and partitioning of large-scale RC systems. The algorithm presented in this section extends existing techniques while operating under the assumptions outlined in Section III. Following pre-processing, the scheduling and partitioning algorithm is divided into two stages. In the first stage, novel list-based scheduling heuristics are used to quickly generate an initial schedule for the system. In the second stage, an extension of the Kernighan-Lin heuristic is employed that takes the initial schedule and iteratively analyzes a set of moves for each unfixed node in the task graph, implementing the best move after each iteration. Pseudocode for the general algorithm is presented in Fig. 1. The following subsections detail the two stages of the algorithm followed by a basic example which illustrates how the algorithm operates.

A. List-Based Scheduling Stage

The first stage of our algorithm uses a novel list-based scheduling (LBS) heuristic to efficiently generate an initial schedule intended to at least reasonably approximate an optimal schedule. Typically, LBS algorithms maintain a list of tasks available for scheduling, and calculate a priority for each task based on a chosen metric. During each step, the available task with the highest priority is selected and mapped onto the architecture, and the list of tasks available for scheduling is updated as well as any dynamically maintained metrics for each unscheduled task. Our LBS heuristic follows this process, and the remainder of this subsection defines the priority metrics used during the LBS stage and the procedure used to map tasks to the architecture once they have been selected for scheduling.

TABLE I
Summary of Key LBS-stage Algorithm Metrics

Symbol	Name	Description
$DAW_P(V_i)$	Device Affinity Weight	A value from 0 to 1 that represents the task's preference for execution on processor type P
$CW_P(V_i)$	Concurrency Weight	The fraction of computational load from this task w.r.t. all tasks at the same level of the task graph on P
$T'_P(V_i)$	Normalized Processing Time	The task's processing time on P multiplied by the single-instance, per-node utilization of P

Before the LBS process begins, a few metrics are calculated for each task, which are summarized in Table I. These metrics are unique to this algorithm and represent extensions to previous LBS techniques. The first is called the Device Affinity Weight (DAW), which represents how strongly a task favors execution on a particular type of processor over any others. For a task V_i and processor type P, the DAW is calculated as

$$DAW_P(V_i) = 1 - \frac{T'_P(V_i)}{\sum_q (T'_q(V_i))} \quad (1)$$

where $T'_P(V_i)$ represents the normalized processing time for task V_i on processor type P, which takes into account resource utilization by V_i in addition to the total processing time. Assuming N_P is the number of processors of type P on a single node of the platform, and U_{avg} is the average of the memory and logic utilization of the task on processor type P, $T'_P(V_i)$ is calculated as follows.

$$T'_P(V_i) = T_P(Vi) \times U_{avg}(V_i, P)/N_P \quad (2)$$

Currently, the LBS stage assumes only one task can occupy an FPGA at any time but allows a single task to be unrolled on one or more devices. The iterative DSE stage later considers potential groupings of separate tasks into hardware cores.

Each task will have a DAW value for each type of processor in the architecture. The DAW metric, whose value ranges from 0 to 1, will give more weight to tasks that strongly favor one type of processor over others, or can only execute on one type of processor. Tasks that do not require or strongly favor one type of processor will have lower affinities since they more likely can afford to be allocated the resources that are leftover after more critical tasks are scheduled. Tasks that are only able execute on one type of processor will be assigned a DAW of 1 for that device, and 0 for all others. The normalized processing time is designed to decrease the weight for tasks that use only a fraction of a processor, or uses a processor that is more prevalent in the architecture than other types of processors. Therefore tasks that demand a lot of resources and/or scarce resources will have higher affinities. These metrics allow the algorithm to make intelligent scheduling decisions based on the makeup of the platform and regardless of its size. With the DAW and T'_P values, the priority can be calculated for each task which is used to determine the order that tasks are scheduled and mapped. In our algorithm, the LBS priority of a task V_i is the maximum DAW multiplied by the normalized processing time for the task on the type of processor corresponding to the maximum DAW value (P_{max}).

$$Priority(V_i) = DAW_{P_{max}}(V_i) \times T'_{P_{max}}(V_i) \qquad (3)$$

As can be seen from Eq. 3, a higher priority will be given primarily to tasks that are more computationally intensive, while also favoring tasks with a strong processor affinity from their DAW set. This allows the algorithm to preferentially schedule tasks that are more demanding in terms of computation and resource restrictions, allowing the less demanding tasks to use leftover resources in the architecture. This behavior is important since efficient scheduling on large-scale systems will often require the scheduler to effectively make use of all resources in the system.

Once a task has been selected for scheduling based on their LBS priority, processing resources must be be allocated for the task. This involves determining the number, type, and location of processor(s) used to execute the task. While DAW is used to determine *when* a task is scheduled via the task priority, it is not used to determine *where* the task is mapped onto the platform. Instead, to facilitate the allocation process a metric called the Concurrency Weight (CW) is maintained for each task, which represents the fraction of total computation a task contains compared to all unscheduled parallel tasks that reside on the task's primary t-level. The *primary t-level* of a task is defined as the number of hops along the longest path from the task to the root (or top, thus the name *t*-level) of the task graph. A task also *resides* on any t-level that is between its own primary t-level and the highest t-level of any of its children. The determination of t-levels is further illustrated in the example in Section IV-C. Thus, for a given processor type P and task V_i,

$$CW_P(V_i) = \frac{DAW_P(V_i) \times T'_P(V_i)}{\sum_n (DAW_P(V_n) \times T'_P(V_n))} \qquad (4)$$

where n represents the set of unscheduled tasks that reside at the same primary t-level of task V_i. The CW serves as a bound that defines the fraction of available processors of type P that may be allocated to the task. A processor (which could be a CPU or FPGA) is considered available if there is no other task executing on it at the earliest possible start time for the task. CW values are dynamically maintained, thus each time after a task is mapped and scheduled, CW values are recalculated for all unscheduled tasks.

When a task is selected to be scheduled, it will be mapped onto the platform to minimize the finishing time of the entire task bounded by the number of resources it is allowed. The determination of the expected completion time of the task is the sum of the required computation, incoming communication, and hardware reconfiguration times. When two separate tasks are mapped to the same FPGA a reconfiguration of the FPGA is taken into account, which may delay the time at which the latter task may receive data and execute on the FPGA. When calculating potential communication delays, the algorithm takes into account whether data must be transmitted within a node and/or between nodes, and applies the appropriate communication rates defined in Section III.

The number of processors that may be allocated to a task is the product of the number of available processors and the corresponding CW. In general, the algorithm will round down the number of processors that a task may use, with a few exceptions. For example, the algorithm will round up the number of processors allowed for allocation if no other concurrent unscheduled task is likely to use an instance of the processor (i.e., $DAW < 0.2$), or if rounding down would result in zero processors allowed for allocation to the task. The CW metric and subsequent allocation scheme enables the algorithm to fairly and efficiently map complex algorithms to systems with lots of devices during LBS. While the tactics described here may lead to a conservative allocation of resources and thus sub-optimal scheduling decisions, this is considered acceptable in order to prevent tasks from being starved of resources when they become available for execution. Furthermore, minor allocation optimizations are intended to be identified and corrected in the iterative DSE stage where more exhaustive analyses are performed. Once all tasks have been initially scheduled, the algorithm moves on to the iterative DSE stage.

B. Iterative Design-Space Exploration Stage

As previously discussed, it is likely that sub-optimal choices will be made during the LBS process, as any single LBS heuristic will struggle to optimally account for every scheduling decision that must be made in large-scale RC environments for complex task graphs. Therefore, this algorithm uses a modified Kernighan-Lin heuristic to perform DSE and find modifications to the initial schedule that will improve the application's expected runtime and potentially correct for sub-optimal decisions made during LBS.

At the beginning of this stage of the algorithm, all tasks are marked as *unfixed*, followed by the iterative cycle. During each iteration, the algorithm analyzes all potential moves for each *unfixed* task. Moves for each task include changing the number, type, and/or location of processors allocated for the task. These moves are similar to many of those discussed and supported in [11]. Only valid moves are considered for each task; thus, e.g., the algorithm will not change the number of devices allocated to tasks that do not support loop unrolling. In addition, our algorithm extends previous work by considering a new move for creating hardware cores. In this move, combinations of hardware tasks are grouped into cores, such that reconfiguration and communication is unnecessary between tasks in the same core on the same device. For

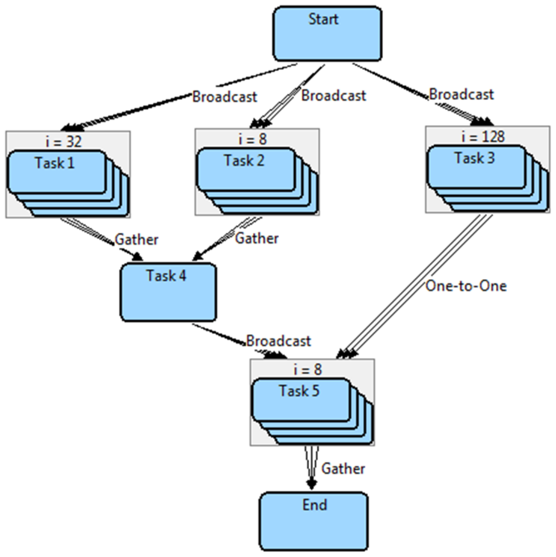

Fig. 2. Task Graph for Walkthrough Example

TABLE II
Task Parameters for Walkthrough Example

	$T(V_i)$		$U(V_i)$	
	FPGA	CPU	Logic	Memory
Task 1	null	10s	NA	NA
Task 2	16s	200s	0.50	0.50
Task 3	128s	128s	0.25	0.25
Task 4	4s	3s	1.0	1.0
Task 5	32s	32s	0.5	0.5

TABLE III
Calculated Task Metrics for Walkthrough Example

	$DAW(V_i)$		Pri	$CW(V_i)$		**# Allocated**	
	FPGA	CPU		FPGA	CPU	FPGA	CPU
Task 1	0	1	5.0	NA	0.31	0	2
Task 2	0.98	0.02	3.9	0.22	0.12	1	0
Task 3	0.89	0.11	30.7	0.78	0.57	3	0
Task 4	0.60	0.40	2.4	0.07	0.08	0	1
Task 5	0.80	0.20	3.2	1	1	4	0

each move, the expected runtime of the whole application is recalculated using the scheduler described in Section IV-A. At the end of each iteration, the move that provides the largest improvement to the overall schedule is selected, and the task associated with the move is changed from *unfixed* to *fixed*. For the case of a hardware core move, the associated tasks are *fixed* into the core, but the core itself is initialized as *unfixed* and may be re-allocated (or grouped to form another core) in successive iterations. The iterative process repeats until all tasks and cores are *fixed* or until there is an iteration where no moves improve the schedule (this differs from the traditional Kernighan-Lin heuristic, which always applies the best move at the end of each iteration until the algorithm completes, regardless of whether the move improves the overall state of the system). Moves that worsen the system's schedule are not accepted in our heuristic in order to minimize the algorithm's processing time.

C. Illustrative Example

To demonstrate the algorithm to the reader, this section presents a walkthrough for a simple example. For simplicity, this example assumes a single-node architecture that features two (2) CPUs and four (4) FPGAs (case studies with a multi-node platform are presented in Section V). All devices communicate with each other over a 1 GB/s bus. Fig. 2 shows the task graph that will be used in this example, which was built using the RC Modeling Language (RCML) environment [12]. The stacked-box icon for Tasks 1, 2, 3, and 5 represents a task that supports loop unrolling. The number directly above the stacked boxes represents the degree of loop unrolling that the task supports (e.g., Task 1 may be unrolled and parallelized by a factor of 32). Parameters for the *total* serial runtime and baseline FPGA hardware utilization are attached to each task and summarized in Table II.

The algorithm begins with preprocessing which first calculates the DAW values for each task, which are shown in the second and third columns of Table III. Since the parameters for Task 1 signify that it can only execute on a CPU, $DAW_{CPU}(V_1)$ is given a value of 1 and $DAW_{FPGA}(V_1)$ a value of 0. Meanwhile, Task 4 has an FPGA execution time of 4s and a CPU execution time of 3s (this could represent a control-intensive task with little parallelism that would see no speedup when mapped to hardware). Based on Eq. 1, $DAW_{FPGA}(V_4)$ is calculated as $1 - \frac{1}{1.5+1} = 0.6$. Similarly, $DAW_{CPU}(V_4)$ is calculated as $1 - \frac{1.5}{1.5+1} = 0.4$. The fourth column of Table III, labeled Pri, lists the resulting priority values for each task, which are calculated using Eq. 3.

As can be seen from Fig. 2, Tasks 1-3 have a primary t-level of one since they are directly below the root, while Task 4 has a t-level of two. Task 3, which has a *primary* t-level of one, also *resides* on the second t-level of the graph since its only child (Task 5) has a t-level of three. Therefore, the CW values for Tasks 1-3 will all take each other into account when determining their own CW. Meanwhile, the CW for Task 4 will factor the load from Task 3 (since Task 3 resides on t-level two), but Task 3 will not factor the load from Task 4 in its own CW (since Task 4 does not reside on t-level one). The initial CW values for each task are shown in the final two columns of Table III.

When the LBS phase begins, Tasks 1-3 are initially considered since they may all execute after the application has started (represented by the Start task). Task 3 is scheduled first, having the highest priority among the three first-level tasks. The CW values limit Task 3 to use up to three of the FPGAs in the architecture and one CPU. Given these constraints, using three FPGAs is the fastest way for Task 3 to execute on the platform (giving Task 3 a completion time of 11 seconds), thus it is allocated three FPGAs and scheduled accordingly. This causes the CW values for Tasks 1 and 2 to be readjusted without consideration of Task 3. This means that $CW_{FPGA}(V_2)$ becomes 1 since no other unscheduled tasks may execute concurrently with it on an FPGA(s). The CW_{CPU} weights for Tasks 1 and 2 are readjusted to 0.72 and 0.28 respectively. Task 1 is scheduled next and is allocated two CPUs (since $DAW_{CPU}(V_2) < 0.2$, we round up the number of CPUs that may be allocated to Task 1), followed by Task 2 being allocated the last available FPGA. At this point, Task 4 is available for scheduling and is allocated one CPU (even though $DAW_{FPGA}(V_4) > DAW_{CPU}(V_4)$, execution on a CPU enables an earlier finish time, thus Task 4 is allocated and scheduled on a CPU). Finally, Task 5 is allocated and mapped onto all four FPGAs. While Task 5 could also use both CPUs in addition or in place of the FPGAs, doing so does not improve the completion time of the entire task. All of the FPGAs must be reconfigured before Task 5 may execute since previous tasks were allocated to them, but in this example the FPGA reconfiguration delay is hidden while Task 5 waits for Task 4 to complete.

At this point of the algorithm, the initial schedule from the LBS stage has been generated, and the iterative DSE process begins. Since no moves can improve the schedule generated by the list-based scheduler, no moves would be implemented during the first iteration and the algorithm would complete.

V. Case studies

To illustrate the effectiveness of our scheduling algorithm, two case studies are presented. Both case studies involve scheduling RC applications on a large-scale RC platform called Novo-G [13]. The Novo-G platform consists of 24 nodes. Each node in our model contains one Xeon E5520 CPU and four Altera Stratix-III E260 FPGAs on a Gidel ProcStar-III board (note: the real system will soon contain two quad-FPGA boards per node). A PCIe 8x connection connects the CPU and FPGA board, while the FPGA board supports 25.6 Gb/s communication directly between FPGAs. The nodes are all connected to a 20 Gb/s DDR InfiniBand switch. For each case study, the expected execution time of the application (hereafter referred to as the schedule length) generated by the initial LBS stage (LBS Only) and the full algorithm (LBS+DSE) are compared to the schedule generated by a baseline simulated annealing (SA) scheduling algorithm developed for this environment. Furthermore, the processing time for each algorithm is also reported to analyze the processing requirements of each algorithm. Each of the scheduling algorithms are executed and timed on a desktop computer with a 1.86 GHz Intel Core2 6300 CPU.

The first case study is a model of an application which performs target detection and classification on a hyperspectral image (HSI) [14]. A hyperspectral image is a collection of 2-D images, all of the same scene but each containing a small, unique portion of the overall spectrum picked up by the sensors. The task graph for the HSI application is illustrated in Fig. 3. As previously discussed, the stacked icons for the ACSM Calc and TC tasks signifies that loop unrolling can be employed to the degree specified by the number just above the boxes. All tasks in HSI may execute on CPUs, while the ACSM Calc and TC tasks may also be mapped to FPGAs.

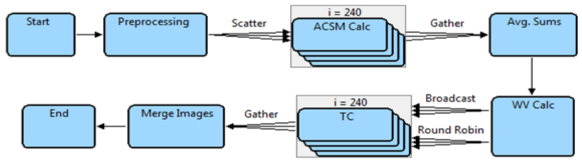

Fig. 3. HSI Task Graph

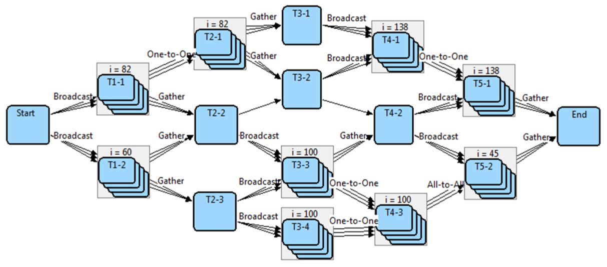

Fig. 4. MVA Task Graph

The second case study is a mean-value analysis (MVA) structured task graph. MVA is a common technique used in analysis of queueing networks, and its task graph structure is often used as a benchmark for scheduling algorithms. To additionally stress the scheduling algorithms, a typical MVA task graph is augmented with sizable loop unrolling amounts assigned to several of the tasks. Randomized FPGA and CPU execution times are assigned to each task as well. The task graph for the MVA case study is shown in Fig. 4.

Table IV summarizes the results from each of the scheduling case studies. The schedule length columns represent the expected completion time of the application based on the generated schedule. For the HSI case study, the results show that all of the scheduling algorithms produced the same schedule length of 15.1s. Since the HSI task graph consisted of a single path of tasks, it was relatively easy for both the LBS algorithm and the simulated annealing algorithm to generate what turned out to be an optimal schedule. Since the LBS stage of our algorithm produced an optimal schedule, no moves were implemented during the iterative DSE stage and the schedule remained unchanged at the end of the algorithm. While it is not remarkable that each algorithm produced an optimal schedule for this task graph, the processing time spent by our algorithm is considerably less than the processing time of the simulated annealing algorithm. Since HSI is likely to be representative of the structure of many RC applications, the results of this case study are encouraging. Additionally, the algorithm could provide reliability and significant time savings to users who would otherwise need to manually perform scheduling and analysis as part of the design process.

For the MVA case study, we see significant differences in the schedule lengths produced by the two stages of our algorithm and the SA algorithm. The SA algorithm generates a schedule length of 34.2s, while the schedule length generated by our algorithm is 35.7s. The SA algorithm is able to produce a slightly better schedule in this case study since it evaluates a much larger number of overall potential mappings, incurring

TABLE IV
SCHEDULING ALGORITHM CASE STUDY RESULTS

Scheduling Algorithm	HSI		MVA	
	Schedule Length	Processing Time	Schedule Length	Processing Time
LBS Only	15.1s	0.3s	41.0s	0.8s
LBS+DSE	15.1s	2.2s	35.7s	110.0s
SA	15.1s	7.5s	34.2s	426.9s

a higher processing time as well. Meanwhile, our algorithm focuses on a smaller subset of potential schedules, using the initial schedule from the LBS stage as a starting point. But due to the greedy nature of the iterative DSE process, a move can be implemented and fixed into the final schedule that prevents the algorithm from ever arriving at the best solution generated by the SA algorithm, as was the case with MVA.

Unlike the HSI case study, six moves are implemented during the DSE stage of our algorithm for the MVA case study. The additional moves and iterative cycles during the DSE stage are the primary reasons for the larger discrepancy between the LBS Only and LBS+DSE processing times in the MVA case study versus the HSI case study. Since the processing time of the algorithm is largely proportional to the number of iterative cycles performed during the DSE stage, it is important that a reasonably good schedule is generated during the initial LBS stage in order to reduce the number of moves that the DSE stage needs to make.

VI. CONCLUSIONS

As the trend towards heterogeneous parallel platforms continues in high-performance and embedded computing, the task of scheduling and partitioning applications on large-scale RC platforms becomes increasingly challenging. Automated techniques for scheduling, HW/SW partitioning, and DSE on scalable RC systems can significantly improve performance and productivity on these systems. While previous research projects have proposed algorithms for automated HW/SW partitioning and scheduling for reconfigurable architectures, none of these approaches support scheduling and partitioning on large-scale or multi-node RC platforms.

In this paper, the first known algorithm designed to perform HW/SW partitioning, scheduling, and DSE for large-scale RC systems is presented. The algorithm uses a two-stage process that performs initial scheduling and partitioning in the first stage, followed by DSE in the second stage. A novel priority and allocation scheme to extend existing list-based scheduling techniques is used in the first stage to obtain an initial schedule for the system. In the second stage, an extension of the iterative Kernighan-Lin heuristic is used that analyzes moves to the unfixed tasks in the graph, implementing the move at the end of each iteration that offers the largest improvement to the total execution time of the application until all tasks are fixed or no moves remain that improve the schedule. After an illustrative example, a pair of case studies were presented showcasing the performance of the scheduling algorithm compared to a baseline simulated annealing (SA) scheduling algorithm. Case study results showed that our algorithm performed nearly as well as the SA baseline, while requiring only a fraction of the time to execute. Future work will look at extensions to support a more generic set of architectures and additional moves during the iterative DSE process, such as pipeline configurations. Finally, comparative studies can be performed to analyze how the approach presented in this paper compares to other existing techniques when adapted to support scalable RC systems.

ACKNOWLEDGMENTS

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422. The authors gratefully acknowledge vendor equipment and/or tools provided by Altera that helped make this work possible.

REFERENCES

[1] B. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical J.*, Feb. 1970.

[2] P. Eles, Z. Peng, K. Kuchchinski, and A. Doboli, "System-level hardware/software partitinoing based on simulated annealing and tabu search," *Journal on Design Automation for Embedded Systems*, vol. 2, no. 1, pp. 5–32, 1997.

[3] J. Henkel, "A low-power hardware/software partitioning approach for core-based embedded systems," in *Proc. of the Design Automation Conference (DAC)*, 1999, pp. 122–127.

[4] D. Menasce, D. Saha, S. C. da Silva Porto, V. Almeida, and S. Tripathi, "Static and dynamic processor scheduling disciplines in heterogeneous parallel architectures," *Journal of Parallel and Distributed Computing*, vol. 28, no. 1, pp. 1–18, July 1995.

[5] R. P. Dick and N. K. Jha, "Cords: Hardware-software co-synthesis of reconfigurable real-time distributed embedded systems," in *Proc. of International Conference on Computer Aided Design (ICCAD)*, November 8-12 1998, pp. 62–67.

[6] F. M. Ali and A. S. Das, "Hardware-software co-synthesis of hard real-time systems with reconfigurable fpgas," *Computers & Electrical Engineering*, vol. 30, no. 7, pp. 471–489, 2004.

[7] G. Stitt, "Hardware/software partitioning with multi-version implementation exploration," in *GLSVLSI '08: Proc. of the ACM Great Lakes symposium on VLSI.* ACM, May 4-6 2008, pp. 143–146.

[8] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, "Hardware-software co-design of embedded reconfigurable architectures," in *Proc. of Design Automation Conference (DAC).* ACM, June 5-9 2000, pp. 507–512.

[9] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Physically-aware hw-sw partitioning for reconfigurable architectures with partial dynamic reconfiguration," in *Proc. of Design Automation Conference (DAC).* ACM, June 13-17 2005, pp. 335–340.

[10] K. S. Chatha and R. Vemuri, "An iterative algorithm for hardware-software partitioning, hardware design space exploration and scheduling," *Design Automation for Embedded Systems*, vol. 5, no. 3-4, pp. 281–293, August 2000.

[11] ——, "Magellan: Multiway hardware-software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs," in *CODES '01: Proc. of the 9th International Symposium on Hardware/Software Codesign*, 2001, pp. 42–47.

[12] C. Reardon, B. Holland, A. George, G. Stitt, and H. Lam, "RCML: An environment for estimation modeling of reconfigurable computing systems," *ACM Transactions on Embedded Computing Systems (TECS)*, to appear.

[13] "http://www.chrec.org/facilities.html," 2009.

[14] C.-I. Chang, H. Ren, and S.-S. Chiang, "Real-time processing algorithm for target detection and classification in hyperspectral imagery," *IEEE Trans. on Geoscience and Remote Sensing*, vol. 39, no. 4, pp. 760–768, April 2004.