Exploiting Application Data-Parallelism on Dynamically Reconfigurable Architectures: Placement and Architectural Considerations

Sudarshan Banerjee, *Member, IEEE*, Elaheh Bozorgzadeh, *Member, IEEE*, and Nikil Dutt, *Fellow, IEEE*

***Abstract*—Partial dynamic reconfiguration, often called run-time reconfiguration (RTR), is a key feature in modern reconfigurable platforms. In this paper, we present parallelism granularity selection (PARLGRAN), an application mapping approach that maximizes performance of application task chains on architectures with such capability. PARLGRAN essentially selects a suitable granularity of data-parallelism for individual *data parallel* tasks while considering key issues such as significant reconfiguration overhead and placement constraints. It integrates granularity selection very effectively in a joint scheduling and placement formulation, necessary due to constraints imposed by partial RTR. As a key step to validating PARLGRAN, we additionally present an exact strategy (integer linear programming formulation). We demonstrate that PARLGRAN generates high-quality schedules with: 1) a set of small test cases where we compare our results with the exact strategy; 2) a very large set of synthetic experiments with over a thousand data-points where we compare it with a simpler strategy that tries to statically maximize data-parallelism, i.e., only considers resource availability; and 3) a detailed application case study of JPEG encoding. The application case-study confirms that blindly maximizing data-parallelism can result in schedules even worse than that generated by a simple (but RTR-aware) approach oblivious to data-parallelism. Last, but very important, we demonstrate that our approach is well-suited for true on-demand computing with detailed execution time estimates on a typical embedded processor. Heuristic execution time is comparable to task execution time, i.e., it is feasible to integrate PARLGRAN in a run-time scheduler for dynamically reconfigurable architectures.**

***Index Terms*—Dynamic reconfiguration, program parallelization.**

I. Introduction

RECONFIGURABLE architectures are popular for applications with intensive computation such as image processing, cryptography, etc., since a limited amount of logic can be customized to set up deep pipelines, and/or exploit more coarse-grain parallelism. Partial dynamic reconfiguration, or run-time reconfiguration (RTR), allows additional customization during application execution enabling true on-demand computing. A device with partial RTR capability is shared by multiple dynamically invoked applications. Each dynamically invoked application is assigned a set of logic resources depending upon system capacity and resource requirements of other applications concurrently active on the same device, and partial RTR makes it feasible for the application to obtain higher performance from a limited set of resources [1]. In this context, our overall goal is to maximize performance of individual applications represented as precedence-constrained task directed acyclic graphs (DAGs) on *single-context* architectures with partial RTR (Xilinx Virtex-II is a commercial instance of such architectures). Some key issues in mapping applications onto such devices are the significant reconfiguration delay overhead, physical (placement) constraints, etc.

In this paper, we focus on precedence-constrained *task chains*, common in image-processing applications such as JPEG encoding, Sobel filters, Laplace filters, etc., [2], [3]. In such applications, area-execution time characteristics of key tasks such as IDCT, Quantize, etc., are predictable because of complete pipelining. Additionally, many computation-intensive tasks such as DCT are completely *data-parallel*, i.e., results of task execution on a block of data are invariant even if the task processed some other disjoint block of data before the current data block.[1] On an architecture with partial RTR, it is possible to improve application execution time by *dynamically* adjusting the parallelism granularity of such data-parallel tasks, i.e., reconfiguring the architecture to instantiate multiple copies of such tasks *during application execution*—each copy (instance) uses an identical amount of hardware logic resources, but processes only part of the data. Due to complete pipelining, execution time of such tasks is directly proportional to the volume of data processed, and thus, reducing the data volume proportionately improves (reduces) the application execution time. Note that on architectures with no partial RTR, the scope of exploiting such data-parallelism is much more limited—partial RTR enables resource reuse, significantly expanding the potential of exploiting data-parallelism.

As an example, we consider a simple chain with two tasks, as shown in Fig. 1. Assuming that there are enough resources to simultaneously execute three copies of task T_1 *or* two copies of task T_2, Fig. 1(b) and (c) show some possible task graph configurations after such a transformation. However, such a transformation can be quite costly on architectures with partial RTR—each new task instance (copy) adds a significant reconfiguration overhead. Therefore, the transformations need to be guided by selecting the right granularity of parallelism that masks the reconfiguration overhead and maximizes performance. One important issue is that because of the recon-

Manuscript received June 21, 2007; revised November 30, 2007. Current version published January 14, 2009.

S. Banerjee is currently with Liga Systems, Sunnyvale, CA 94089 USA (e-mail: banerjee@ics.uci.edu).

E. Bozorgzadeh and N. Dutt are with the Center for Embedded Computer Systems, University of California, Irvine, CA 92697 USA.

Digital Object Identifier 10.1109/TVLSI.2008.2003490

[1] Huffman encoding is a well-known example of a task that *does not* have data-parallelism property.

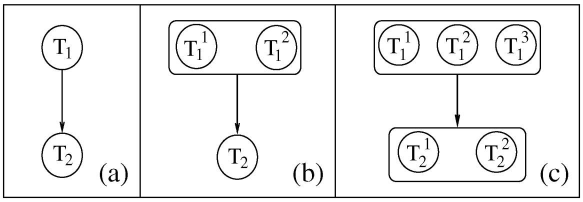


Fig. 1. Granularity of individual data-parallel tasks.

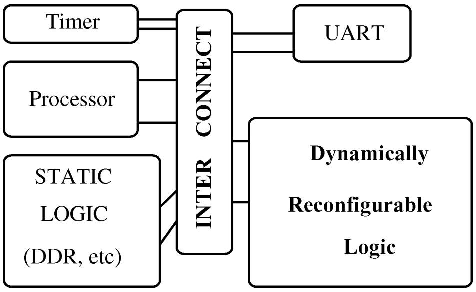


Fig. 2. On-demand computing environment.

figuration overhead, multiple instances of a task are typically unable to start execution at the same time—thus, individual execution time (*workload*) of the multiple instances may vary.

As the capacity of modern reconfigurable architectures continues to increase rapidly, suitably sized devices can accomodate many of our target applications without needing partial RTR. In this context, it is important to clarify that our work is motivated by the following applications:

- maximizing performance from available smaller (and cheaper) devices—this aspect includes *huge* multimedia applications such as the MPEG-4;
- more importantly, maximizing performance in a true on-demand computing environment such as that shown in Fig. 2. An embedded system includes a dynamically reconfigurable component **shared by multiple on-demand applications**. When an on-demand application is ready to execute, it is granted resources (logic area, etc.) subject to resource usage of other currently active applications, and partial RTR enables us to maximally exploit the available resources. Note that such an embedded system may be implemented completely on a large modern device such as the Xilinx Virtex-5 XC5VLX330. Part of the logic in such a device is statically configured for invariant functionality (such as timers, memory controllers, etc.) and the other part is dedicated to accelerating applications on demand using partial RTR.

In such an on-demand computing environment, we additionally require a *semi-online* application mapping (scheduling) approach to maximize application performance with partial RTR. We define a *semi-online* application execution scenario as follows.

1) The application structure is known statically, i.e., predecessor and successor of each individual task *does not* change during application execution. This property is satisfied by typical image-processing applications such as Sobel filtering [4], JPEG decoding, etc. Additionally, key scheduling parameters such as logic resource requirement of each individual task are also available statically.
2) When the application is invoked dynamically, it is allocated a set of logic resources depending upon its runtime environment.
3) A *semi-online* scheduling approach generates an application execution schedule based on the static scheduling parameters and two *run-time* parameters: (a) allocated logic resources and (b) input image size.
4) The scheduled application starts execution.

That is, a *semi-online* scheduling approach allows an application to adapt to key changes in its runtime environment—as examples, change in available logic resources and change in input image size directly affect the potential for performance improvement. This necessitates that the execution time of a semi-online approach is low enough to be included in an operating system for dynamically reconfigurable architectures [5], [6].[2] Thus, measure of viability for a semi-online approach is *cumulative execution time* defined as sum of *schedule length generated by approach* and *execution time of approach on an embedded processor*.

A. Paper Contributions

1) Granularity selection, scheduling, placement: In this paper, we propose such a *semi-online* scheduling approach, parallelism granularity selection (PARLGRAN), that maximizes application performance on architectures with partial RTR by choosing the right parallelism granularity for each individual data-parallel task. We define granularity as *both* the ***number of instances*** (copies) of that task, and, the ***workload*** (execution time) of each instance. Our approach considers physical (placement) constraints, and utilizes configuration prefetch [7] to reduce the latency. The key constraints of such architectures necessitate joint scheduling and placement [8], [9]. Our approach therefore, incorporates granularity selection as an integral part of simultaneous scheduling and placement. To the best of our knowledge, ours is the first effort towards this difficult problem of *semi-online application restructuring*.

2) ILP, Detailed experiments, Case Study: As a key step in understanding the problem and to validate quality of schedules generated by our approach, we additionally present an exact strategy (ILP). We present extensive experimental evidence to validate our proposed approach. First, we demonstrate that PARLGRAN generates results close to that of the exact (ILP) formulation with a set of small test cases. Since the exact strategy is very time consuming, we next present results on a very large set of over a thousand synthetic experiments where we compare against a simpler heuristic that tries to statically maximize performance gain from data parallelism based on resource availability only—average improvement in schedule length is over 20%. We follow-up our experiments on synthetic test cases with a detailed case study of JPEG encoding—the experiments demonstrate that a simple (RTR-unaware) static parallelization approach can end up generating schedules much worse than a RTR-aware approach completely oblivious to data-parallelism.

3) Semi-Online Capability: Finally, we have obtained detailed execution time estimates of our approach on a typical embedded processor, the PPC405 processor operating at a clock

[2] In scenarios where the image size and allocated resources are invariant at run-time, we would of course precompute the schedule statically with much less consideration for the one-time computation overhead.

frequency of 400 MHz. The data indicates that execution time of our approach is comparable to that of task execution time. Equally importantly, for our application case study, cumulative execution time *monotonically improves*. For a given image size, as available area increases, (execution time of heuristic + schedule length generated by heuristic) monotonically decreases. Thus, PARLGRAN is well-qualified for semi-online scheduling, i.e., for inclusion in an operating system for dynamically reconfigurable devices.

II. Related Work

While there exists a large body of work in mapping task chains typical in image processing to reconfigurable architectures, a significant amount of work such as [2] does not consider dynamic reconfiguration. More recently, there has been a spurt in work focussed on exploiting the powerful capabilities of partial dynamic reconfiguration for image-processing/multimedia applications [4], [10]–[12], etc. Our work is closely related to work such as [4], [10], etc., that focus on task graph scheduling with RTR-related constraints.

Recent work on scheduling application task graphs with RTR-related constraints [4], [10], often do not focus on the critical role played by placement on such architectures. Our work focusses on joint scheduling and placement required on architectures with partial RTR, similar to [8] and [13]. However, prior work in joint scheduling and placement typically ignore key architectural constraints such as the resource contention due to a single reconfiguration controller, configuration prefetch to reduce the reconfiguration latency, etc. Ignoring these key issues makes the problem closer to the rectangle packing problem [14] and does not realistically exploit RTR. Other recent work such as [15] focus on the problem of configuration reuse as an alternative strategy to reduce the reconfiguration overhead, an aspect we do not address in this work.

Additionally, work on task-graph scheduling for such architectures [9], [10] typically does not include application restructuring considerations. While [4] presents some application restructuring considerations, their work is completely oblivious to placement concerns. Also, their target device is a *multicontext* architecture with multiple concurrently active reconfiguration processes. Commercially available devices with partial RTR are *single context* architectures where only a single reconfiguration process is active at any instant. (True multicontext architectures such as Morphosys [16] incur a significant area overhead.) One work that exploits data-parallelism in a single-context architecture [17] simply considers the problem of maximizing performance for a single task, without considering dependencies between tasks in a task graph—it also does not include detailed placement considerations. To the best of our knowledge, this work is the first effort that focuses specifically on techniques for transforming applications on *single-context* architectures and includes very detailed consideration of ***all*** partial RTR related constraints such as placement, resource contention due to the sequential reconfiguration mechanism, etc.

There is of course a vast body of knowledge in the compiler domain on extracting parallelism from programs at different levels of granularity [18]. Such *compile-time* techniques [19] are typically unaware of partial RTR constraints (such as placement, reconfiguration overhead)—equally importantly, compile-time techniques also incur a high execution overhead, since they are not intended for execution in an embedded environment. Another related work [10] has proposed a strategy based on Pareto points to precompute schedules when there are a (known) limited number of parameter variations at run-time. Along with being placement-unaware, this strategy is not fully capable of exploiting a true on-demand computing environment. In our paper, we explicitly focus on a low execution-complexity approach capable of maximally exploiting variability in task execution parameters (data size) and resource allocation in a run-time scheduling environment.

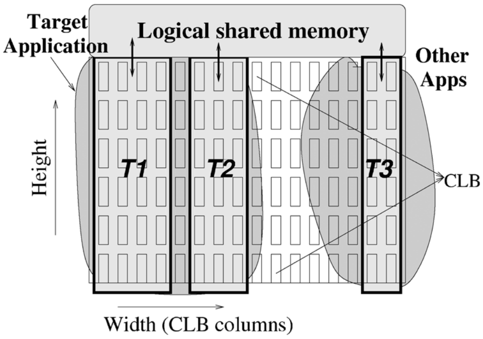


Fig. 3. Target dynamic architecture.

III. Problem Overview

A. Target Architecture

Our target dynamically reconfigurable device as shown in Fig. 3 consists of a set of configurable logic blocks (CLB) arranged in a 2-D matrix. The basic unit of configuration for such a device is a frame spanning the height of the device. A column of resources consists of multiple frames. A task occupies a contiguous set of columns. The reconfiguration time of a task is directly proportional to the number of columns (frames) occupied by the task implementation. One key constraint is that only one task reconfiguration can be active at any time instant. An example of our target device is the Xilinx Virtex-II architecture where constraints such as dynamic tasks occupying a contiguous set of columns are critical for realization of partial run-time reconfiguration.

Note that our target device is capable of supporting multiple concurrently executing applications, as shown in Fig. 3. When an application is invoked, it is allocated a set of resources (CLB columns) dependent on the current system state—partial RTR enables the application to maximize performance from the limited set of allocated logic resources. Also, while we do not explicitly show it in Fig. 3, part of the target device is used for *static* system functionality such as UARTs, operating system functionality (such as a scheduler), etc.

B. Application Specification

A task T_i executing on such a system can be represented as a three-tuple (c_i, t_i, r_i) where c_i is the number of resource columns occupied by the task, t_i and r_i are the execution time and reconfiguration overhead, respectively. Each task needs to be reconfigured before its execution is scheduled. The physical constraints on such a device necessitates joint scheduling and placement [8], [9].

In image processing applications, we often find chains (linear sequences) of such tasks. For a chain of n tasks, $(T_1, \ldots, T_n)$, each task in the chain has exactly one predecessor and one successor. Of course, the first task, T_1, has no predecessor, and the last task, T_n, has no successor. A predecessor task utilizes a shared memory mechanism to communicate necessary data to its successor—this shared memory can be physically mapped to local on-chip memory and/or off-chip memory depending upon memory requirements of the application. Detailed discussion on the specifics of memory organization, including strategies for on-chip versus off-chip data mapping are beyond the scope of this paper—we simply assume that the *logical shared memory* provides sufficient bandwidth for our target applications. One important aspect of our shared memory abstraction is that communication time between two tasks is independent of their physical placement. (An example of a system architecture capable of providing such an abstraction is available in [20].)

Such a chain of tasks can be executed in a pipelined fashion when sufficient resources are available to instantiate all tasks in the chain. In this work, we specifically focus on scenarios with limited logic when all tasks *cannot* be simultaneously placed in the available logic. Detailed considerations of inter-task pipelining versus data-parallelism (when insufficient resources are available to place all tasks in a pipeline) is out of scope of this work, and will be considered in future work.

C. Problem Objective

Our overall goal is to maximize performance (minimize schedule length) under physical and architectural constraints, given a resource constraint of C_{cons} columns available for the application, where C_{cons} is less than that required to map the entire application,[3] i.e., $C_{\text{cons}} < \sum_{i=1}^{n}(c_i)$. An additional key goal is that our approach should have a low computational overhead suitable for implementation on a typical embedded processor.

IV. Detailed Problem Specification and Exact Mathematical Formulation (ILP)

In this section, we first motivate our problem and follow-up with a detailed problem specification. Next, we provide an exact mathematical formulation (ILP).

A. Motivation and Detailed Problem Specification

Ideally, the degree of parallelism for a data-parallel task is limited only by the availability of HW resources. Let us consider a chain with only a single data-parallel task T_1 that executes in time t_1 using c_1 columns, as shown in Fig. 4(a). Given a resource constraint of C_{cons} columns, we expect performance to be maximized (schedule length minimized) when this task is instantiated $\lfloor C_{\text{cons}}/c_1 \rfloor$ times, as in Fig. 4(b). In these figures, the X-axis represents the columnar area constraint C_{cons} and the Y-axis represents the schedule length. For sequential tasks (0 degree of data-parallelism), the execution of task T_i is represented as E_i as in Fig. 4(a). Each individual task T_i requires reconfiguration before execution—however, for ease of presentation, we show all our schedules (and corresponding equations) as starting from execution of the first task T_1 in the chain. For data-parallel tasks, we additionally denote execution of jth instance (copy) of task T_i as E_i^j, as in Fig. 4(b).

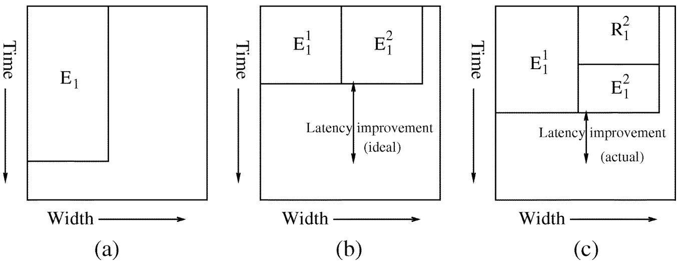


Fig. 4. Effect of significant reconfiguration overhead. (a) Sequential. (b) Ideal parallel. (c) Actual parallel.

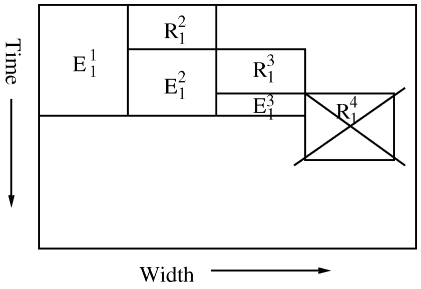


Fig. 5. Parallelism degree determined by reconfiguration overhead.

Unfortunately, as we discuss next, the ideal performance gain in Fig. 4(b) is typically not achievable while considering realistic issues on such architectures.

1) Significant Reconfiguration Overhead: For modern *single-context* architectures that support partial RTR, the large reconfiguration delay is a key bottleneck in achieving ideal parallelism. To illustrate this, we consider Fig. 4(c). In this figure, reconfiguration for jth instance (copy) of T_1 is denoted as R_i^j. Similar to our convention of not explicitly showing reconfiguration R_1 for task T_1 in Fig. 4(a), we do not explicitly show reconfiguration R_1^1 for the first data-parallel instance T_1^1 in Fig. 4(c). We simply assume that the reconfiguration controller is available at the beginning of the execution of the first instance T_1^1. Next, we attempt to maximize performance by instantiating an additional copy T_1^2 and distributing the workload (execution time) equally between the two instances T_1^1 and T_1^2. However, execution of the second instance E_1^2 can start only after the reconfiguration overhead, r_1. Thus, instead of the ideal workload of $t_1/2$, the workload of the second task instance is only $(t_1 - r_1)/2$ leading to less performance improvement than expected. The actual schedule length is $(t_1 + r_1)/2$ instead of the *ideal* schedule length of $t_1/2$.

For a single task, a simple equation suffices to compute the *optimal workload* leading to maximum performance improvement, as shown in the following lemma.

Lemma 1: For parallelizing a task T_i into j instances, and given that the reconfiguration controller is available at the beginning of execution of the first instance, the best performance (least execution time) is obtained when the workload (execution time) of the jth instance is: $(t_i - r_i \times j \times ((j-1)/2))/j$.

Proof: The proof follows directly from the simple example of parallelizing task T_1 into two task instances $(j = 2)$.

[3] We assume of course that the largest task fits into the available area, i.e., $C_{\text{cons}} \geq \max(c_i)$.

When the jth task instance is ready for execution (reconfiguration for T_i^j is complete), workload completed by T_i^1 is $(j-1)\times r_1$, workload completed by T_i^2 is $(j-2)\times r_1,\ldots,\ldots,$ workload completed by T_i^{j-1} is r_1. The aggregate workload completed *before* T_i^j starts is

$$r_1 \times ((j-1)+(j-2)+\cdots+1) = r_1 \times j \times \frac{j-1}{2}.$$

To maximize performance (minimize schedule length), the remaining workload is distributed equally between all j task instances, i.e., workload assigned to instance T_i^j is

$$\frac{t_i - r_i \times j \times \frac{j-1}{2}}{j}.$$

■

Lemma 1 clearly demonstrates that maximizing performance involves *unequal* workload (execution time) distribution between multiple copies of a task to compensate for the significant reconfiguration delay and the sequential reconfiguration mechanism.

Along with reducing expected performance, the large reconfiguration delay also potentially prevents performance improvement if more than a few copies of a task are instantiated, as shown in Fig. 5. Even though enough resources are available to instantiate four copies of task T_1, instantiating the fourth copy does not improve the schedule length any further. In fact, assigning any nonzero workload to the fourth instance leads to a longer schedule than a schedule with only three instances. Similar to the previous lemma for computing *optimal workload*, a simple equation suffices to compute the *optimal number of instances* leading to maximum performance improvement, as shown in the following.

Lemma 2: For parallelizing a task T_i and given that the reconfiguration controller is available at the beginning of execution of the first instance, the best performance (least execution time) is obtained when there are exactly n_i^{opt} instances

$$n_i^{\text{opt}} = \text{MIN}\left(\left\lfloor \frac{C_{\text{cons}}}{c_i} \right\rfloor, \left\lceil \frac{1+\sqrt{1+8\times\frac{t_i}{r_i}}}{2} \right\rceil - 1\right).$$

Proof: The first term $\lfloor C_{\text{cons}}/c_i \rfloor$ states that one trivial bound on the number of instances is simply the maximum number of task copies that fit in the available area. The second term follows from Lemma 1 as shown in the following.

If the j_wth instance *does not* improve performance, the aggregate workload completed *before* $T_i^{j_w}$ starts execution is *greater than* the task workload, i.e.,

$$r_i \times j_w \times \frac{j_w - 1}{2} > t_i.$$

Solving the previous quadratic equation, performance *does not* improve if

$$j_w \geq \left\lceil \frac{1+\sqrt{1+8\times t_i/r_i}}{2} \right\rceil.$$

Thus, the maximum number of instances n_i^{opt} such that performance *definitely improves* is given by $n_i^{\text{opt}} = j_w - 1$, leading to the second term in the lemma. ■

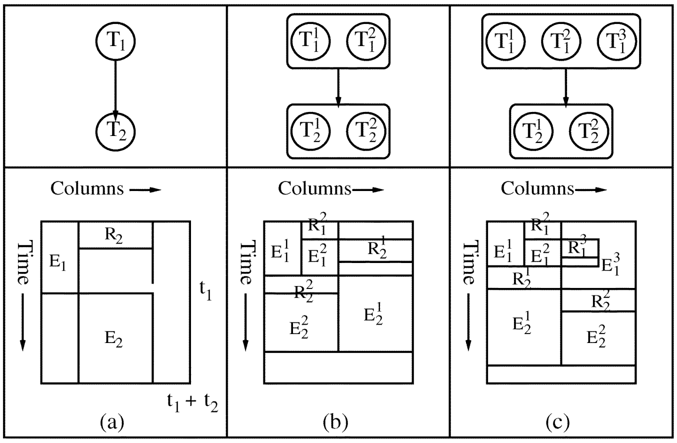


Fig. 6. Problem space explosion with precedence constraints.

Thus, the granularity of data-parallelism, that includes selecting number of instances, is determined by two factors: along with the very obvious factor of the number of instances that fit in the given space, the other key factor is the ratio of task execution time to task reconfiguration time. In the experimental section, we have conducted experiments on images with different sizes—for such experiments, the reconfiguration time for individual tasks is invariant, while the task execution time is proportional to the image size. The experimental results validate this lemma, i.e., there is more performance improvement with increasing image size (resulting from potentially more instances for each data-parallel task).

The simple equations in Lemmas 1 and 2 provide the underlying principles for our proposed approach. However, they are not sufficient to compute the schedule length for a precedence-constrained *task chain*. We next consider the additional complications introduced in our problem due to precedence constraints.

2) Precedence Constraints: For precedence-constrained application task chains, there is interaction of the resource demands of parent and child tasks, as shown in Fig. 6 for a simple chain with two tasks T_1 and T_2. The HW resource constraint allows three copies (instances) of T_1, **or**, two copies of T_2 to be executing simultaneously. One possible approach is to exactly follow Lemmas 1 and 2, i.e., instantiate all three copies of T_1 to maximize performance of T_1, and then instantiate two copies of T_2, as in Fig. 6(c). However, it is potentially possible to improve the schedule length further by instantiating only two copies of T_1 and using the remaining space to reconfigure (instantiate) one copy of T_2—once the two copies of T_1 end, the first instance T_2^1 of T_2 is able to start execution immediately, as in Fig. 6(b). Note that in our execution model, *all instances of a parent task must finish execution before any instance of a child task starts execution*.

Obviously, the problem space explodes with the introduction of precedence constraints. Effectively for a chain with n tasks, we want to determine the best possible performance from

$$\lfloor C/c_0 \rfloor \times \lfloor C/c_1 \rfloor \times \ldots \lfloor C/c_n \rfloor$$

candidate transformed task graphs.

To better understand the problem difficulty, note that scheduling a simple *task chain* under partial RTR constraints is NP-complete even without any data-parallelism considerations. The detailed proof based on reduction from set-partitioning is out of scope for this paper—the interested reader should contact the author [21] for further details. Also, configuration prefetch [7] plays a critical role—in Fig. 6(b), the *gap* introduced between completion of reconfiguration R_2^1 and start of execution E_2^1 for task instance T_2^1 is crucial to improving latency in the presence of significant reconfiguration delay. Thus, our detailed problem specification is shown in the following.

Problem Inputs:

- precedence-constrained application task chain $(T_1- > T_2- > \cdots - > T_n)$ where *some* tasks T_i have data-parallelism property;
- strict bound C_{cons} on the number of contiguous columns available for mapping the task chain.

Problem Output:

- *number* of copies for each data-parallel task;
- *workload* (execution time) t_i^j of each (jth) copy of a data-parallel task $(\sum_j (t_i^j) = t_i)$;
- *placed task schedule* where every task (instance) is assigned an execution *start time*, and an execution *start column*.

Problem Objective:

- minimize schedule length (maximize performance).

As mentioned previously, an additional objective is that any solution should have low execution complexity suitable for implementation on typical embedded processors. However, for the sake of completeness (and as a key tool to evaluate the quality of our proposed heuristics), we next present an exact mathematical formulation (ILP) to the previous problem.

B. Mathematical Formulation (ILP) of Problem

In this subsection, we present an integer linear program (ILP) that provides an exact solution to our problem. Our underlying model is a 2-D grid where task placement is modelled along one axis while time is represented on the other axis. Previous work [9] has addressed the problem of exact scheduling (and placement) for a task graph with partial RTR related constraints (including configuration prefetch) based on such a grid representation. Unlike [9], our objective is to determine the structure of a task graph that maximizes performance—attempting to determine the *number of task instances* and *execution time of an instance* while satisfying all constraints related to columnar partial RTR makes the ILP formulation additionally challenging.

1) Core Principles: To formulate such an ILP, we essentially start with an expanded *series-parallel* graph. For each data-parallel task T_i, we implicitly instantiate as many task copies T_i^j as possible subject to the resource constraint C_{cons}. For each such task instance we add precedence edges to the child task T_{i+1} of T_i (or, to *every* instance of task T_{i+1}, if T_{i+1} is data-parallel).

Next, we introduce a Boolean (0-1) variable ID (invalid) for every task instance in the expanded graph—a nonzero value of this variable denotes that the corresponding task instance is **not** required for maximizing performance. To determine task instance execution time along with task instance start time, we introduce two sets of variables: sx (start execution) variable for a task instance denotes the execution start time of the task instance, while x (is executing) variable denotes that a task instance is processing data in a given time-step.

The following indices are key to properly specifying the ILP variables:

$i \in (1, \ldots,$ number of tasks in the chain$)$
$i' \in (1, \ldots,$ number of task instances in the expanded graph$)$
$l_i \in (1, \ldots,$ number of instances of task $T_i)$
$j \in (1, \ldots,$ upper bound on schedule length$)$
$k \in (1, \ldots, C_{\text{cons}})$.

2) ILP Variables: The complete set of 0-1 (decision) variables is

- $x_{i',j,k} = 1$, if task instance $T_{i'}$ **is executing at** time-step j, and k is left-most column occupied by $T_{i'}$. (=0, otherwise)
- $sx_{i',j,k} = 1$, if instance $T_{i'}$ **starts execution at** time-step j, and k is left-most column occupied by$T_{i'}$. (=0, otherwise)
- $fx_{i',j} = 1$, if instance $T_{i'}$ finishes execution in time-step j. (=0, otherwise)
- $ID_{i'} = 1$, if $T_{i'}$ is **not required** in an optimal solution. $(= 0$, otherwise)
- $r_{i',j,k} = 1$, if reconfiguration for $T_{i'}$ starts at time-step j, and k is left-most column occupied by $T_{i'}$. (=0, otherwise)

Some of the constraints necessitate introduction of additional binary variables to represent logical conditions. All such variables are represented as b.

3) Constraints:

1) Simple task execution constraints

a) Each valid task instance is executed exactly once

$$\forall i', \quad \sum_k \sum_j (sx_{i',j,k}) + ID_{i'} = 1, \quad \sum_j (fx_{i',j}) + ID_{i'} = 1. \tag{1}$$

b) Task instance execution-time is non-negative, i.e., execution finish time for a task instance is greater than or equal to execution start time

$$\forall i', \quad \sum_j \left(j * fx_{i',j} - \sum_k (j * sx_{i',j,k}) \right) \geq 0. \tag{2}$$

Note that this is true *for all* task instances. If a task instance is not required, its corresponding sx (start execution) and fx (finish execution) variables are all assigned a value of zero.

2) Core data-parallelism constraints:

a) Execution time for a task equals the aggregate execution time of all instantiated copies

$$\forall i, \quad \sum_{l_i} \sum_j \sum_k (x_{l_i,j,k}) = t_i. \tag{3}$$

This equation holds trivially for all non-data-parallel tasks that have a single instance each.

b) Precedence constraints between task instances: Each valid instance of task T_i $(i > 1)$ starts execution after any instance of T_{i-1} finishes execution

$$\forall i > 1, \forall l_i, \forall l_{i-1},$$
$$(ID_{l_i} = 0) \Longrightarrow \sum_j \left(\sum_k (j * sx_{l_i,j,k}) - j * fx_{l_{i-1},j} \right) \geq 1. \tag{4}$$

We can rewrite the equation in the following form:

$$\forall i > 1,\ \forall l_i,\ \forall l_{i-1},$$
$$\textbf{if}\ ((1 - ID_{l_i}) > 0)\ \textbf{then}$$
$$\sum_j \left(\sum_k (j * sx_{l_i,j,k}) - j * fx_{l_{i-1},j} \right) - 1 \geq 0.$$

This enables us to apply the *if-then* transformation as in [22].[4]

3) Core column-based partial RTR constraints:

a) Each valid task instance needs to be reconfigured—also, the start column for reconfiguration is same as start column for execution

$$\forall i', \forall k, \quad \sum_j (r_{i',j,k}) - \sum_j (sx_{i',j,k}) = 0. \tag{5}$$

b) Each valid task instance can start processing data only after the task is reconfigured, i.e., *reconfiguration delay* time-steps after start of reconfiguration

$$\forall i', \quad (ID_{i'} = 0) \Longrightarrow \sum_j e \sum_k (j * sx_{i',j,k} - j * r_{i',j,k}) \geq t_i^{rf} \tag{6}$$

where t_i^{rf} denotes the reconfiguration time for task T_i. We can apply the *if-then* transform similar to that for (4).

c) Resource constraints on field-programmable gate array (FPGA): total number of columns being used for task instance executions and number of columns being reconfigured is limited by the total number of FPGA columns

$$\forall j, \sum_{i'} \sum_k \sum_{n=k-c_i+1}^{k} \left(x_{i',j,n} + \sum_{m=j-t_{i'}^{rf}+1}^{j} (r_{i',m,n}) \right) \leq C_{\text{cons}}. \tag{7}$$

d) At every time-step j, at most single task instance is being reconfigured

$$\forall j, \quad \sum_{i'} \sum_{m=j-t_{i'}^{rf}+1}^{j} \sum_k (r_{i',m,k}) \leq 1. \tag{8}$$

e) At every time-step j, mutual exclusion of execution and reconfiguration for every column

$$\forall j, \forall k, \quad \sum_{i'} \sum_{n=k-c_{i'}+1}^{k} \left(x_{i',j,n} + \sum_{m=j-t_{i'}^{rf}+1}^{j} (r_{i',m,n}) \right) \leq 1. \tag{9}$$

f) For every column, at every time-step, total number of reconfigurations is at most 1 less than the number of executions started using that column

$$\forall j, \forall k, \quad \sum_{i'} \sum_{n=k-c_{i'}+1}^{k} \sum_{m=1}^{j} (r_{i',m,n} - sx_{i',m,n}) \leq 1. \tag{10}$$

g) Simple placement constraint: a task can start execution only if there are sufficient available columns to the right

$$\forall i, \forall j, \forall k \in (C_{\text{cons}} - c_i + 1, \ldots, C_{\text{cons}}), \quad x_{i',j,k} = r_{i',j,k} = 0. \tag{11}$$

4) Equations relating task execution variables:

a) For each task instance, if it *starts execution* in time-step j (sx variable is "1"), variables denoting task *is executing* are zero in prior time-steps and "1" in time-step j

$$\forall i', \forall j > 1,$$
$$\left(\sum_k (sx_{i',j,k}) = 1 \right) \Longrightarrow \sum_k \sum_{m=1}^{m=j-1} (x_{i',m,k}) = 0 \tag{12}$$
$$\forall i', \forall j > 1,$$
$$\left(\sum_k (sx_{i',j,k}) = 1 \right) \Longrightarrow \sum_k (x_{i',j,k}) = 1. \tag{13}$$

b) For each task instance, if it *is executing* in column k, the corresponding *starts execution* variable is true for this column

$$\forall i', \forall k, \quad \left(\sum_j (x_{i',j,k}) \geq 1 \right) \Longrightarrow \sum_j (sx_{i',j,k}) = 1. \tag{14}$$

c) For each valid task instance, the task instance execution time equals the number of nonzero *is executing* variables

$$\forall i', \forall k, (ID_{i'} = 0) \Longrightarrow \sum_j \sum_k (x_{i',j,k})$$
$$= \sum_j \left(\sum_k (j * sx_{i',j,k}) - j * fx_{i',j} \right) + 1. \tag{15}$$

All the previous equations can be simplified using the *if-then* transform described earlier.

5) Objective function to minimize schedule length: This is equivalent to minimizing the end time for any instance of the last task in the chain T_n. By introducing a new sink task T_{sink} and precedence edges from all instances of the last task in the chain T_n, the objective function is simply the execution start time for this new task T_{sink}. If we additionally assign a width of C_{cons} columns to this new task, the objective function is further simplified to

$$\text{minimize} \sum_j (j * sx_{1,\text{sink},1}).$$

6) Additional constraints: Along with the necessary constraints, we also introduce *additional constraints* such as simple timing ASAP/ALAP constraints to help reduce the search space (and correspondingly reduce the time required by the ILP solver to find a solution).

V. Heuristic Approaches

In this section, we first present MFF, a heuristic for scheduling simple task chains. While MFF is oblivious to data-parallelism,

[4] if-then transform for the constraint if $(f(X) > 0)$ then $g(X) \geq 0$ is $-g(X) \leq Mb$; $f(X) \leq M(1-b)$; $b \in (0,1)$, where M is a large number such that $f(X) \leq M$, $-g(X) \leq M$ for X satisfying all other constraints.

it provides the core concepts underlying PARLGRAN, our proposed approach for chains with *some* data-parallel tasks.

A. *Modified First Fit (MFF)*

For architectures with partial RTR, the physical (placement) constraints and, the architectural constraint of the single reconfiguration mechanism, make it difficult to achieve the ideal schedule length $L_{\text{ideal}} = \sum_{i=1}^{n}(t_i)$. In fact, this *simple* problem of minimizing schedule length for a chain, under constraints related to partial RTR, is actually NP-complete, as mentioned in Section IV (the detailed proof by reduction from set-partitioning is out of scope for this paper). MFF, our proposed heuristic to solve this problem, essentially tries to satisfy task resource constraints, and, attempts simple local optimizations to *reduce fragmentation*, and, hence, the schedule length.

```
Approach: MFF (Modified First Fit)
Place task $T_1$ starting from leftmost column
for each task ($T_i$, i > 1)
  $F_i^S$  =
$\mathrm{earliest\ time - slot\ enough\ space\ is\ available(last - fit)}$
  $F_i^R$  =
$\mathrm{earliest\ time - slot\ reconfiguration\ controlleris\ available}$
  $R_i^{start} = \mathrm{MAX}(F_i^S, F_i^R)$
  $E_i^{start} = \mathrm{MAX}(R_i^{start} + r_i, E_{i-1}^{end})$
  if ($T_i$ aligned with rightmost column)
    local optimization: Adjust immediate ancestor placement
    (and start time) if possible to improve start time of $T_i$
endfor
```

MFF is based on a first-fit approach. To get intuition behind why a first-fit approach works well in practical scenarios, we take a look at Fig. 7(b). The tasks are essentially laid out in the form of diagonals running from the top-left of the placed schedule towards the bottom-right (the diagonal in the figure results from each task in the chain being placed to the right of its predecessor and can start only after its predecessor completes). As long as a task does not "fall off" the diagonal, it is possible to overlap at least part of the reconfiguration overhead with the execution of its immediate ancestor. Once a task "falls off" the diagonal and is placed at the left-most column C_{cons}, it is essentially trying to reuse the area of ancestor tasks higher up in the chain. Given that for tasks in a chain the execution components have to be in sequence, a more distant ancestor is guaranteed to finish earlier than a closer ancestor. This increases significantly the possibility of being able to overlap reconfiguration of this task with the execution of ancestors that are closer to it in the chain. Effectively the chain property causes a "window" of tasks: tasks within a window affect each other much more strongly than tasks outside the window.

1) Simple Fragmentation Reduction: One simple modification for reducing fragmention in MFF compared to pure first-fit is shown in Fig. 7. Our observations indicate that in tightly-constrained scenarios (few columns available for task mapping), placing the second task T_2 adjacent to task T_1, as in Fig. 7(b), often leads to immediate fragmentation—though enough area is available to reconfigure task T_3 in parallel with execution of task T_2, this area is not contiguous, and thus task T_3 gets delayed. MFF takes care of this by placing T_2 at the right-hand corner. Of course, this simple technique does not improve fragmentation in all possible scenarios.

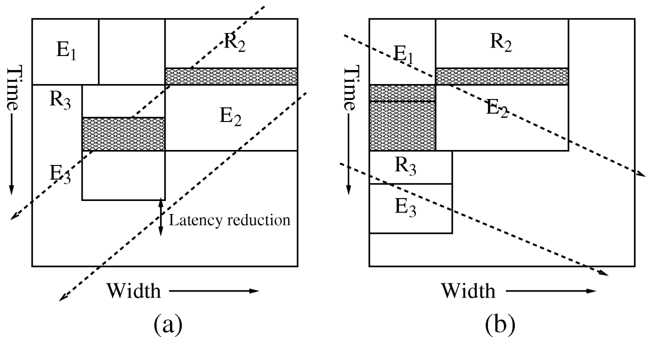


Fig. 7. Simple chain-right placement of task 2. (a) Less fragmentation. (b) More fragmentation.

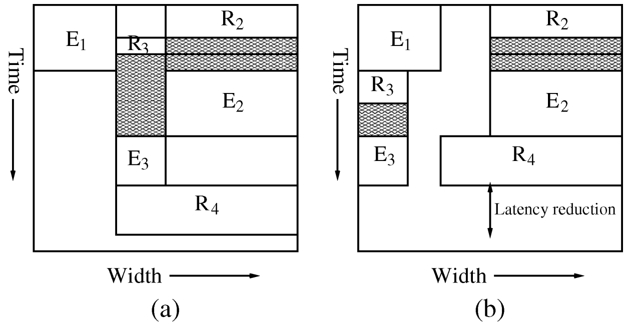


Fig. 8. Exploiting slack in reconfiguration controller. (a) More fragmentation. (b) Less fragmentation.

2) Local Optimization—Exploiting Slack in Reconfiguration Controller: A more interesting local optimization to reduce fragmentation is shown in Fig. 8(a). While scheduling task T_4, we notice that it is possible to exploit slack in the reconfiguration mechanism to *postpone* the reconfiguration R_3 of task T_3 without delaying the actual execution E_3 of task T_3. We can thus make better use of the available area (HW resources) to reschedule (and change placement of) task T_3—as a result, reconfiguration R_4 of task T_4 can now execute in parallel with E_3, leading to a reduction in schedule length, as shown in Fig. 8(b).

Before proceeding to PARLGRAN, it is important to understand that the fragmentation problems we try to address in MFF (and PARLGRAN) are because we are trying to jointly schedule and place while satisfying a host of other constraints—thus, other free space coalescing techniques for partially reconfigurable architectures, such as [23], are not directly applicable.

B. *PARLGRAN*

We use the insights obtained from the chain-scheduling problem as the basis for our granularity selection approach. Detailed analysis of chain-scheduling shows that applying local optimizations can improve the performance. We additionally want to design an approach such that the algorithm execution time is comparable to the execution time of the tasks. Thus, our proposed granularity selection approach is simple and greedy, but, uses specific problem properties to try and improve the solution quality.

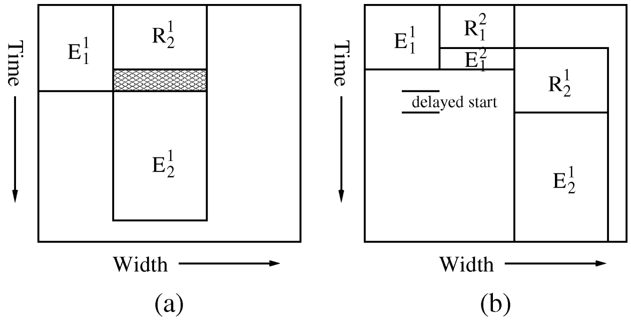


Fig. 9. Static pruning based on timing.

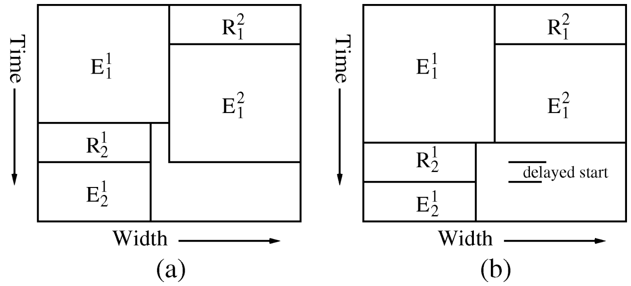


Fig. 10. Uneven finish times.

Our approach consists of the following two steps:

- static pruning;
- dynamic granularity selection.

1) Static Pruning: First, we utilize some simple facts to statically prune regions of the search space. As an example of pruning, consider Fig. 9. If we schedule exactly one copy each for tasks T_1 and T_2, then task T_2 can start as soon as T_1 ends, i.e., at t_1, as in Fig. 9(a). If we schedule another copy of task T_1, the execution time of T_1 improves. However, now the reconfiguration controller becomes the bottleneck, as shown in Fig. 9(b). Now, task T_2 can start only at $(r_1 + r_2)$, which is greater than t_1. Effectively, the number of copies of a task is limited by the impact of its reconfiguration overhead on its successors. Note that such simple *static pruning* is based primarily on timing considerations.

2) Dynamic Granularity Selection: We next consider work distribution (load balancing) issues for the multiple task copies.

a) Uneven Finish Times: From our initial discussion on data-parallelism (as shown earlier in Fig. 5), it seems that it is a good idea to always generate as many copies as possible subject to performance improvement and get them to finish at the same time instant. However, with the introduction of task dependencies, it is necessary to modify this strategy in certain cases to improve performance, as shown in Fig. 10. In Fig. 10(a), let FT_1^1 denote the time instant the earlier copy of task T_1, that is E_1^1 ends. Task T_2 can start at: $ST_2^1 = FT_1^1 + r_2$. However, if both copies of T_1 end at the same time instant as shown in Fig. 10(b), this time-instant is given by

$$FT_1^{\text{equal}} = FT_1^1 + r_2/2.$$

As a result, reconfiguration R_2 for task T_2 gets delayed and execution E_2 for task T_2 can only start at$FT_1^{\text{equal}} + r_2 = FT_1^1 + 3 * r_2/2$.

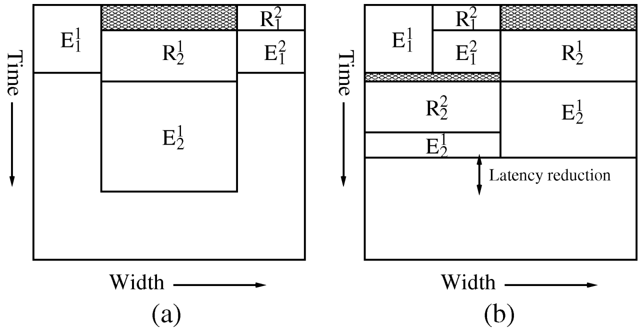


Fig. 11. Left placement for instances of first task. (a) More fragmentation. (b) Less fragmentation.

Of course, if the area of task T_2 is greater than the area of task T_1, letting both copies of T_1 end at the same time instant would lead to a shorter schedule.

b) Adjacent Task Instances: Another simple observation to improve MFF specifically for parallelism granularity selection is shown in Fig. 11. Placing multiple copies of a task adjacent to each other intuitively helps reduce fragmentation.

PARLGRAN is an adaptation of MFF that essentially tries to greedily add multiple copies of data parallel tasks as long as it estimates that adding a new copy is beneficial for performance (shorter schedule length). The concepts of dynamically adjusting the workload combined with local optimizations makes it effective. We summarize our PARLGRAN approach shown as follows.

PARLGRAN (Parallelism Granularity Selection)

```
Place first copy of task $T_1$ starting from left-most column for
each task $(T_i, i > 1)$
    Compute earliest execution start of task (space search
    by last-fit)
    if (parent task is data-parallel)
        while (no degradation in start time of $T_i$)
            add new copy of parent (assign start time,
            physical location)
            adjust workload of existing scheduled copies
            of parent
    Schedule (and place) $T_i$
    apply local optimizations from MFF for improving
    schedule
endfor
```

In the previous code segment, one task is considered in each iteration of the outer loop. Using a last-fit-based placement strategy, we obtain the earliest time-instant that this task can start execution. The inner loop selects the granularity of its parent task, i.e., selects the number of instances and workload of the parent task such that the current task can start execution earlier (of course this is applicable only to parent tasks that are data-parallel). As each new instance of a parent task is added, the algorithm uses the fragmentation reduction strategies described earlier to try and reduce the schedule length. When it is not possible to improve the start time of the current task any further, the inner loop terminates and the next task in the chain is selected. Note that granularity selection is very tightly integrated with placement—at each step, the schedule length is computed based on the physical location of tasks on the device.

While this approach appears to be simplistic, experimental results in the following section show it typically does better than statically deciding to parallelize each task to its maximum degree. For real image-processing applications such as JPEG encoding, blind parallelization can lead to *significantly inferior* results, even worse than RTR-aware first-fit, because of the reconfiguration overhead and the physical (placement) constraints.

VI. Experiments

We conducted a wide variety of experiments to validate our proposed approach. We demonstrate the quality of schedules generated by our heuristics with a very large set of synthetic experiments (consisting of over a thousand data-points) along with a detailed application case study. Additionally, we demonstrate the *semi-online* applicability of PARLGRAN with detailed analysis of estimated execution time on a typical embedded processor, the PPC405 (PowerPC) processor with an operating frequency of 400 MHz.

It is important to remember that our goal is to maximize performance (minimize schedule length) for an application task chain in an on-demand computing scenario where a dynamically invoked application is assigned logic resources (area for mapping application task graph) depending on the number and resource requirement of other applications simultaneously executing on the reconfigurable device. Thus, while it is possible to fit our applications onto suitably sized target devices, we assume for experimental purposes that the hard resource constraint C_{cons} is less than the aggregate size of all tasks.

A. Experimental Setup

We assumed a target device with 24 columns, similar to Xilinx XC2V2000.[5] From the XC2V2000 data sheet, we estimate that the reconfiguration overhead for the smallest task occupying one column on our architecture is 0.38 ms at the maximum suggested reconfiguration frequency of 66 MHz. We obtained area and timing data for well-known tasks such as Huffman encoding, DCT, etc., by synthesizing them with columnar placement and routing constraints on the XC2V2000, similar to the Xilinx methodology suggested for *reconfigurable modules*.[6]

We explored a large set of scenarios with the following strategy:

1) We generated task chains of varying chain length, ranging from 4 to 15 tasks in the chain.
2) For a task chain of given chain length, each individual task was assigned a set of parameters (execution time, reconfiguration delay, number of columns) randomly selected from our database of synthesized tasks. Thus, we generated multiple task chains for a given chain length.
3) Finally, for each individual task chain, we conducted multiple experiments by varying the area constraint across a wide range, to represent situations with less area, as well as situations with more area available for mapping the application.

Our overall strategy resulted in a set of over a thousand individual experiments. Note that the database of task parameters included information corresponding to images of various sizes—since each individual task is completely pipelined, the reconfiguration delay and number of columns occupied by the task is independent of the image size, but the execution time is directly proportional to the image size.

In subsequent discussions, the following notation denotes schedule length generated by various approaches, including our proposed approach, the exact formulation, and other heuristics implemented to evaluate the quality of schedules generates by our approach.

- L_{exact}: corresponds to our exact (ILP) formulation.
- L_{mff}: corresponds to our MFF approach for scheduling chains with no data-parallelism considerations.
- L_{pgran}: corresponds to our proposed PARLGRAN approach.
- L_{ff}: corresponds to a simple first-fit (FF) approach [9] for scheduling chains with no data-parallelism considerations.
- L_{maxp}: corresponds to maximum parallelization (MAXPARL) approach.

Along with our implementation of MFF, PARLGRAN, and the ILP, we additionally implemented MAXPARL to evaluate the quality of our schedules. MAXPARL attempts to maximize parallelization by statically selecting the maximum number of copies possible for each task subject to resource constraints only, and assigning equal workload to each such task instance. Note that MAXPARL includes detailed configuration prefetch considerations. Because of equal workload distribution, multiple instances of a task finish at different time-instants, unlike Lemma 1—however, the freed-up area is utilized to instantiate multiple copies of any data-parallel child task. Thus, the schedules generated by MAXPARL are of reasonable quality and significantly better than an approach with no configuration prefetch considerations.

B. Schedule Quality on Synthetic Experiments

1) Schedule Quality of MFF (Compared to FF): Our first set of experiments consisted of comparing schedule lengths generated by MFF with that of first-fit, on the set of experiments as described above.

The experimental data confirmed that schedules generated by MFF were almost always equal to or better than FF. The schedule lengths generated by MFF were better in 207 out of 1096 tests, i.e., approximately 19% of the tests, worse in 6 out of 1096 tests. In 114 tests, around 10% of the total, MFF was better by at least 3%. In the worst experiment for MFF, first-fit generated a schedule longer by 0.44%. Overall, on longer chains (more tasks) and looser constraints (more columns), both algorithms were almost equally able to hide the reconfiguration overhead. However, on more constrained problems with shorter chains and tighter area constraints, MFF tends to generate better schedules.

2) Comparing PARLGRAN Schedule Length With ILP for Small Tests: Our next set of experiments consisted of comparing

[5] While the XC2V2000 datasheet specifies a 56 × 48 matrix of logic blocks, architectural constraints for partial RTR necessitate that one dynamically reconfigurable column equals *two* columns of logic on the physical device as shown online http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf.

[6] [Online]. Available: http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf

TABLE I
PARLGRAN VERSUS ILP FOR SMALL TESTS

Testcase	L_{exact}	L_{pgran}
test2	25	25
test3	23	23
test5	19	22
test7	25	27
test8	23	24
jpg3	48	49

TABLE II
REDUCTION IN SCHEDULE LENGTH FOR COMPLETELY DATA PARALLEL CHAINS WITH PARLGRAN

Chain	PARLGRAN Vs FF	PARLGRAN Vs MAXPARL		
length	Avg	Avg	Best	Worst
4-6	44%	7.1%	93.1%	-49.6%
7-9	55%	20.5%	139.2%	-31.2%
10-12	63%	31.8%	142.7%	-27.3%
13-16	71%	38.9%	125%	-7.1%
Avg gain	$> 50\%$	$> 20\%$		

the schedule length generated by PARLGRAN with that generated by the exact formulation. The implementation of the ILP using the commercial solver CPLEX[7] requires hours for even very small testcases on our implementation platform (SunOS 5.9 with a 502 MHz Sparcv9 processor). Thus, for experiments involving the exact formulation, we report results on a small set of synthetic experiments with short chains where chain length varies between 3 to 5 tasks (the experiments also include one testcase (jpg3) from the detailed case study presented in the next subsection).

In Table I, the second column represents schedule lengths generated by the ILP, while the third column represents schedule lengths generated by PARLGRAN. For this set of experiments, the schedule length is reported in time-steps where one time-step corresponds to the reconfiguration delay for a single CLB column.

As the table shows, the schedules generated by PARLGRAN for small experiments (short chains) are reasonably close to that of the exact approach.

3) Overall Schedule Quality of PARLGRAN: Next, in Table II, we present a summary of results covering the entire set of synthetic experiments. The data in each row of the table corresponds to experiments on chains of corresponding length—as an example, data in the second row (chain length 7–9) was obtained from experiments on chains with at least 7 tasks and at most 9 tasks. Note that this set of experiments is identical to that we used to validate MFF—we additionally assume that each task in the chain is completely data-parallel. For comparison with MAXPARL and FF, our quality measure is simply the percentage increase in schedule length generated by the other approach compared to PARLGRAN. As an example, for comparison with MAXPARL, the quality measure is simply

$$((L_{\mathrm{maxp}} - L_{\mathrm{pgran}})/L_{\mathrm{pgran}}) * 100.$$

The second column in Table II represents the *Average* percentage improvement of PARLGRAN as compared to FF. Each entry in the second column is an average of a large number of experiments conducted on chains of corresponding length. The third, fourth and fifth columns, respectively, represent the *Average*, the *Best*, and the *Worst* performance of our approach compared to MAXPARL. As an example, the data in the second row, fourth column, states that on a large number of experiments with chain length between 7 and 9 tasks, the best result generated by our approach corresponds to an experiment where MAXPARL generated a schedule 139% longer.

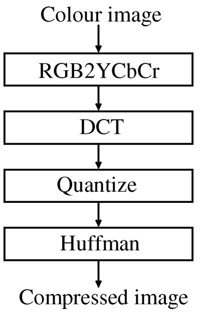


Fig. 12. JPEG encoder task graph.

TABLE III
CASE STUDY OF JPEG ENCODING: SCHEDULE LENGTH WITH DIFFERENT IMAGE SIZE AND AREA CONSTRAINTS

Case	C_{cons}	L_{mff} (ms)	L_{maxp} (ms)	L_{pgran} (ms)
256*X*256 JPG	5	12.71	12.73	**12.36**
	6	11.24	12.52	**10.81**
	7	11.24	11.38	**10.05**
	8	11.24	12.11	**9.08**
	9	10.10	12.79	**9.08**
512*X*512 JPG	5	42.86	40.68	40.30
	6	41.34	35.32	35.13
	7	41.34	34.18	34.37
	8	41.34	29.08	28.60
	9	40.20	28.38	27.71

Expectedly, there is significant improvement in schedule length with PARLGRAN compared to the sequential (FF) approach, as shown in the second column of the table. More importantly, the data in the third column clearly shows that our proposed *granularity selection* heuristic, PARLGRAN, generates increasingly better results compared to MAXPARL when more space is available. Intuitively, with more available area, it is possible to make more instances of the data-parallel tasks. However, with each additional instance, the workload (execution time) decreases per instance, resulting in execution time comparable to the reconfiguration overhead—PARLGRAN is better capable of deciding when to stop instantiating multiple copies, as opposed to MAXPARL. The local optimizations in PARLGRAN play an active role in such circumstances to help improve the schedule length.

One key aspect of the data in Table II is that for smaller chains, our presented results cover a very large range of varying area constraints—for longer chains, the presented results cover the scenarios where the available HW area is at most 40%–45% of the aggregate HW area of the tasks. For chains with more than 9–10 tasks, a loose area constraint results in even more significant improvement with PARLGRAN compared to other approaches.

[7] [Online]. Available: http://www.ilog.com/products/cplex

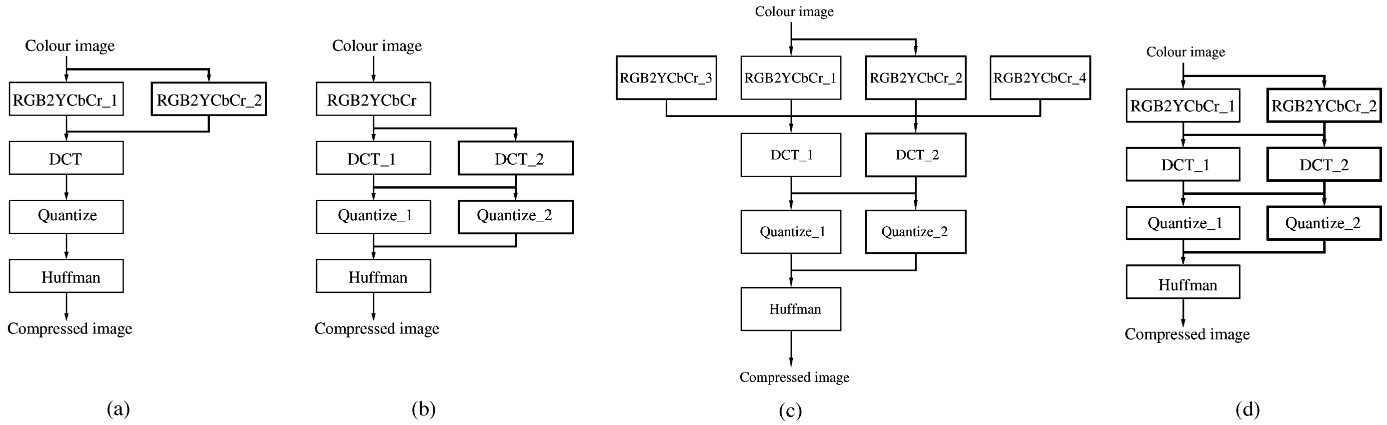


Fig. 13. Transformed JPEG task graph: (a) Image size: 256 × 256, $C_{\mathrm{cons}} = 5$. (b) 256 × 256, $C_{\mathrm{cons}} = 8$. (c) maximum parallelization, $C_{\mathrm{cons}} = 8$. (d) Image size: 512 × 512, $C_{\mathrm{cons}} = 8$.

C. Detailed Application Case Study: JPEG Encoding

After conducting a wide range of experiments on synthetic graphs, we conducted a detailed application case study on the JPEG encoding algorithm, represented as a chain of four key tasks $(\mathrm{RGB2YCbCr-} > \mathrm{DCT-} > \mathrm{Quantize-} > \mathrm{Huffman})$, shown in Fig. 12. Note that Huffman is a sequential task (no data-parallelism) while the remaining 3 tasks are data-parallel. Table III presents some results from our case study. Entries in the first column, Case, denote the image size—256 × 256 denotes experiments on a 256 × 256 color image. For each case, we varied the number of columns and observed the resulting schedule lengths (the aggregate area requirement of all tasks in the chain is 11 columns). The second column C_{cons} represents the area constraint in columns. The third, fourth and fifth columns correspond to schedule lengths (in milliseconds) generated by MFF, MAXPARL, and PARLGRAN, respectively.

The data in Table III demonstrates that as available area increases, our proposed approach PARLGRAN consistently generates shorter schedules. As an example, for the 256 × 256 image, we consider the data corresponding to $C_{\mathrm{cons}} = 5$, and the data corresponding to $C_{\mathrm{cons}} = 8$. The corresponding transformed task graphs are shown in Figs. 13(a) and (b), respectively. The DCT task is the most computation-intensive task in the chain (maximum execution time). However, a tighter area constraint $(C_{\mathrm{cons}} = 5)$ does not allow multiple instances of the DCT task. Thus, PARLGRAN improves performance by adding one instance of the RGB2YCRCB task, as shown in Fig. 13(a). However, with more area $(C_{\mathrm{cons}} = 8)$, PARLGRAN is capable of deciding that it is more beneficial to instantiate two copies of the DCT and only have a single instance of the RGB2YCRCB task. For comparison, we note that an approach oblivious to partial RTR constraints would generate four instances of the RGB2YCRCB task with $C_{\mathrm{cons}} = 8$, as shown in Fig. 13(c).

Next, we observe how our approach adapts to varying data size with Figs. 13(b) and (d). For the same area constraint $(C_{\mathrm{cons}} = 8)$, the transformed task graph for the 256 × 256 image has *six* tasks while that for the 512 × 512 image has *seven* tasks. For the larger image, the task execution time is significantly higher than the task reconfiguration time, resulting in more scope for exploiting data-parallelism, as in Lemma 2.

Next we focus on experimental results for the 256 × 256 image. For this set of experiments, where the reconfiguration overheads are comparable to the task execution times, our approach frequently does much better than statically parallelizing everything (MAXPARL). Additionally, the data demonstrates that such blind parallelization can lead to results worse than a simple sequential scheduling approach. For an area constraint of eight columns, schedule length of FF is longer than PARLGRAN by $(11.24 - 9.08)/9.08 = 23.5\%$. Blind (static) parallelization leads to significantly worse schedule longer by $(12.11 - 9.08)/9.08 = 33.3\%$. This is in spite of the fact that the effective transformed graph from MAXPARL [see Fig. 13(c)] consists of nine tasks with apparently more parallelism, while the transformed graph from PARLGRAN [Fig. 13(b)] consists of six tasks only.

For the 512 × 512 image, each task execution time is significantly greater than the reconfiguration overhead. In such a scenario, where, additionally, the chain length is short, MAXPARL generates good results—of course, PARLGRAN typically does somewhat better. But, both parallelizing approaches result in significant speedups.

D. Applicability in Semi-Online Scenario

The experimental data clearly demonstrates that PARLGRAN generates high-quality schedules. However, our objective is for PARLGRAN to be applicable in a *semi-online* scenario where the task precedence relations, and the task area-timing characteristics are available at compile-time, while the available HW area for mapping the application is known only at run-time. Task management under such dynamic resource availability is a key issue in modern operating systems for reconfigurable architectures [5]. So, we next obtained detailed execution time estimates for MFF, PARLGRAN, and MAXPARL on the PPC405 operating at 400 MHz—such a processor is available in the Xilinx Virtex-II Pro platform.

We obtained heuristic execution time estimates for the JPEG encoding application with three different image sizes: 256 × 256, 384 × 384, 512 × 512. For each image size, we varied the area constraint and obtained *cumulative* execution time as shown in Figs. 14(a)–(c). In each of these figures, the X-axis represents the area constraint as a percentage of the aggregate area required by all tasks. The Y-axis represents the cumulative execution time (schedule length computed by heuristic + execution time of heuristic).

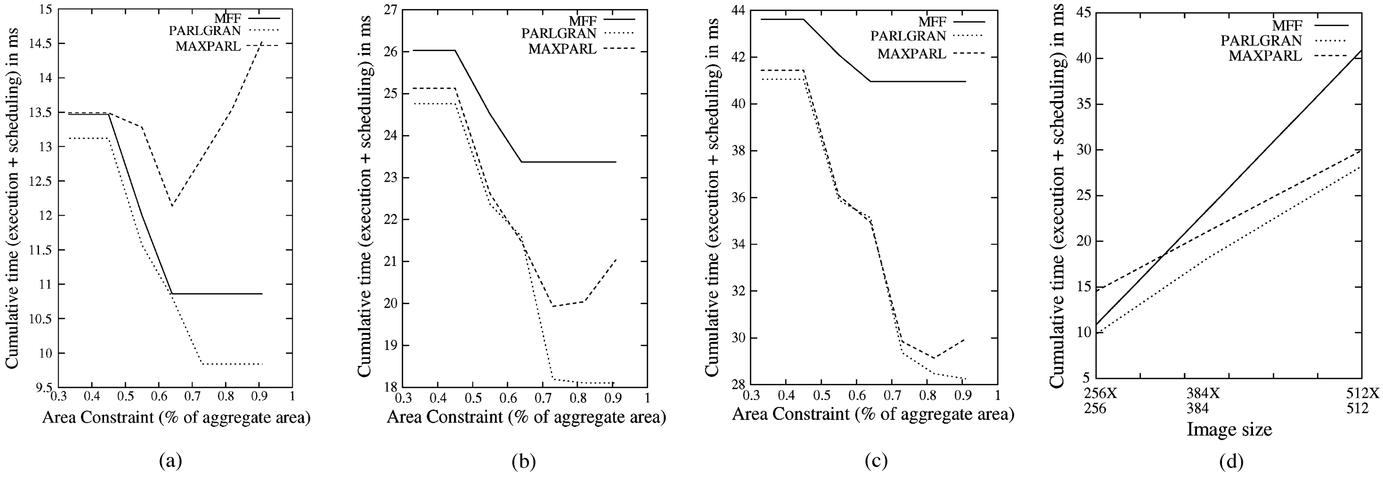


Fig. 14. Schedule length + heuristic execution time: (a) JPEG encoding 256 × 256. (b) JPEG encoding 384 × 384. (c) JPEG encoding 512 × 512. (d) Loose area constraint.

MFF of course has the least execution time overhead. Thus, for short chains with a very tight area constraint, cumulative execution time with MFF is comparable to other heuristics, as in Fig. 14(a). However, as available area increases or, image size increases, scope for exploiting data-parallelism increases. In such scenarios, PARLGRAN and MAXPARL generate shorter schedules that more than compensate for the increased heuristic execution time.

This is explicitly demonstrated in Fig. 14(d) where we present data for three different image sizes scheduled with the same (relaxed) area constraint. Note that increase in image size results in increased ratio of task execution time to reconfiguration overhead, making more data-parallel instances feasible (as in Lemma 2). As image size increases, cumulative execution time with MFF increases almost linearly, i.e., cumulative execution time for a 512 × 512 image is almost 4 times that of the 256 × 256 image. However, with approaches that attempt to exploit data-parallelism, the cumulative execution time increases at a slower rate—for PARLGRAN, the cumulative execution time for the 512 × 512 image is only around **2.5 times** that for the 256 × 256 image.

Heuristic execution time for all approaches increase as more area is available for mapping the application. However, MAXPARL is significantly more sensitive, as shown in Figs. 14(a) and (b). This is because MAXPARL attempts to maximize parallelism by scheduling a graph with the maximum number of tasks possible in the given area.

Comparing the data in Table III with that in Fig. 14(a) shows that PARLGRAN execution time overhead is approximately $(9.85 - 9.08) = 0.77$ ms for the 256 × 256 image with $C_{\text{cons}} = 8$. This is quite low compared to the schedule length $L_{\text{pgran}} = 9.08$ ms. Much more importantly, for all experiments on the JPEG application, *cumulative execution* time for PARLGRAN **monotonically decreases** confirming its viability in a semi-online environment.

Our wide range of experiments and case studies confirm that PARLGRAN generates high-quality schedules in all situations—tightly constrained problems with shorter chains, fewer columns, as well as problems with more degrees of freedom, i.e., longer chains, more available columns. Additionally, the estimated run-time of our approach on a typical embedded processor is comparable to the HW task execution times.

VII. Conclusion

In this paper, we proposed PARLGRAN, a *semi-online* scheduling approach that selects granularity of data-parallelism to maximize performance of application *task chains* executing on architectures with partial RTR capability. Our approach selects both the number of instances of a data-parallel task, and, the execution time (workload) of each such instance—it is integrated in a joint scheduling and placement formulation, necessitated by the underlying physical and architectural constraints imposed by partial RTR.

To evaluate our proposed heuristic, we have formulated and implemented an exact (ILP) approach, and a simpler strategy that attempts to *statically* maximize data-parallelism. Results on smaller experiments show that PARLGRAN generates schedules reasonably close in quality to that of the exact approach. Experimental results on a very large space with over a thousand synthetic experiments confirm that PARLGRAN generates schedules that are on an average better by 20% compared to the simpler strategy that attempts to statically maximize data-parallelism based on logic availability only.

A detailed case study on JPEG encoding confirms that in realistic scenarios, the simpler approach that tries to maximize data parallelism without accounting for the underlying RTR-related constraints can end up generating schedules *much worse* than even a data-parallelism-oblivious (but RTR-aware) approach. Finally, detailed execution-time estimates indicate that our approach is suitable for integration in a *semi-online* scheduling methodology where the goal is to maximize performance of an application given an area constraint and input characteristics (image size) available only at run-time.

While our approach demonstrates the potential for significant performance improvement, there are some key aspects that we want to address in our future work. Most importantly, we have assumed in this work that we are not constrained by memory/communication bandwidth. Our initial estimates indicate that even with increased parallelism, the additional bandwidth requirement for realistic applications (including the JPEG application) can be satisfied by a typical memory-bus configuration such as a PC3200 DDR memory integrated with a suitable bus. However, we agree that with increased task granularity (more instances) and ever-increasing device sizes (enabling more applications to execute concurrently), the data transfer to and from memory, both on-chip and off-chip, has the potential to become a bottleneck, and will be considered in future work. Last, but not the least, we have focussed on exploiting data-parallelism with partial RTR. An operating system [5] that handles resource management in a true on-demand computing environment requires a toolbox that can suitably mix and match a variety of techniques including but not limited to inter-task pipelining, clustering, etc.

References

[1] M. J. Wirthlin, "Improving functional density through run-time circuit reconfiguration," Ph.D. dissertation, Elect. Comput. Eng. Dept., Brigham Young Univ., Provo, UT, 1997.

[2] H. Quinn, L. A. S. King, M. Leeser, and W. Meleis, "Runtime assignment of reconfigurable hardware components for image processing pipelines," in *Proc. IEEE Symp. Field Program. Custom Comput. Mach. (FCCM)*, 2003, pp. 173–182.

[3] J. Noguera and R. M. Badia, "Power-performance trade-offs for reconfigurable computing," in *Proc. IEEE/ACM/IFIP Int. Conf. Hardw.-Softw. Codesign Syst. Synth. (CODES+ISSS)*, 2004, pp. 116–121.

[4] J. Noguera and R. M. Badia, "Performance and energy analysis of task-level graph transformation techniques on dynamically reconfigurable architectures," in *Proc. Int. Conf. Field Program. Logic Appl. (FPL)*, 2005, pp. 563–567.

[5] C. Steiger, H. Walder, and M. Platzner, "Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks," *IEEE Trans. Computers*, vol. 53, no. 11, pp. 1393–1407, Nov. 2004.

[6] G. Brebner, "A virtual hardware operating system for the Xilinx XC6200," in *Proc. Int. Workshop Field-Program. Logic*, 1996, pp. 327–336.

[7] S. Hauck, "Configuration pre-fetch for single context reconfigurable processors," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, 1998, pp. 65–74.

[8] S. P. Fekete, E. Kohler, and J. Teich, "Optimal FPGA module placement with temporal precedence constraints," in *Proc. Des. Automat. Test Europe (DATE)*, 2001, pp. 658–667.

[9] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Integrating physical constraints in HW-SW partitioning for architectures with partial dynamic reconfiguration," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 11, pp. 1189–1202, Nov. 2006.

[10] J. Resano, D. Mozos, and F. Catthoor, "A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable architectures," in *Proc. Des. Automat. Test Europe (DATE)*, 2005, pp. 106–111.

[11] C. Bobda, M. Majer, A. Ahmadiniya, T. Haller, A. Linarth, and J. Teich, "The Erlangen slot machine: Increasing flexibility in FPGA-based reconfigurable platforms," in *Proc. Field-Program. Technol. (FPT)*, 2005, pp. 37–42.

[12] N. Sedcole, P. Y. K. Cheung, G. A. Constantinides, and W. Luk, "A reconfigurable platform for real-time embedded video image processing)," in *Proc. Field Program. Logic Appl. (FPL)*, 2003, pp. 606–615.

[13] P.-H. Yuh, C.-L. Yang, Y.-W. Chang, and H.-L. Chen, "Temporal floorplanning using the T-tree formulation," in *Proc. Int. Conf. Comput.-Aided Des. (ICCAD)*, 2004, pp. 300–305.

[14] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani, "Rectangle-packing based module placement," in *Proc. Int. Conf. Comput.-Aided Des. (ICCAD)*, 1995, pp. 472–479.

[15] J. Harkin, T. M. Mcginnity, and L. P. Maguire, "Modeling and optimizing run-time reconfiguration using evolutionary computation," *ACM Trans. Embedded Comput. Syst.*, vol. 3, no. 4, pp. 661–685, 2004.

[16] H. Singh, G. Lu, E. M. C. Filho, R. Maestre, M.-H. Lee, F. J. Kurdah, and N. Bagherzadeh, "MorphoSys: Case study of a reconfigurable computing system targeting multimedia applications," in *Proc. Des. Automat. Conf. (DAC)*, 2000, pp. 573–578.

[17] K. N. Vikram and V. Vasudevan, "Mapping data-parallel tasks onto partially reconfigurable hybrid processor architectures," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 9, pp. 1010–1023, Sep. 2006.

[18] S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA: Morgan Kaufmann, 1997.

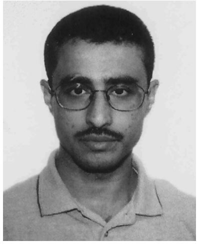
[19] T. Stefanov, B. Kienhuis, and E. Deprettere, "Algorithmic transformation techniques for efficient exploration of alternative application instances," in *Proc. Int. Symp. Hardw./Softw. Codesign (CODES)*, 2002, pp. 7–12.

[20] J. Noguera, "Energy-efficient hardware/software co-design for dynamically reconfigurable architectures," Ph.D. dissertation, Dept. Comput. Arch., Techn. Univ. Catalonia, Barcelona, Spain, 2005.

[21] J. Augustine, *Personal Communication.* 2005. [Online]. Available: john.augustine@tcs.com

[22] W. L. Winston and M. Venkataraman, *Introduction to Mathematical Programming*, 4th ed. Boston, MA: Thomson Brooks Cole, 2003.

[23] M. Handa and R. Vemuri, "An efficient algorithm for finding empty space for online FPGA placement," in *Proc. Des. Automat. Conf. (DAC)*, 2004, pp. 960–965.



Sudarshan Banerjee (M'99) received the B.Tech. and M.Tech.degrees in computer science and engineering from the Indian Institute of Technology, Kharagpur, India, and the Ph.D. degree in information and computer science from the University of California, Irvine.

He is currently working on hardware-assisted simulation with Liga Systems, Sunnyvale, CA. He has extensive experience in development of industry-leading logic verification tools as an employee of Synopsys and Cadence. His current research interests include partitioning and scheduling for HW-SW codesign, dynamically reconfigurable FPGAs.

Elaheh Bozorgzadeh (S'00–M'03) received the B.S. degree in electrical engineering from Sharif University of Technology, Iran, in 1998, the M.S. degree in computer engineering from Northwestern University, Evanston, IL, in 2000, and the Ph.D. degree in computer science from the University of California, Los Angeles, in 2003.

She is currently as Assistant Professor with the Department of Computer Science, University of California, Irvine. Her research interests include design automation for embedded systems, reconfigurable computing, and VLSI/FPGA CAD. She has coauthored over 45 conference and journal papers.

Prof. Bozorgzadeh was a recipient of a Best Paper Award from the IEEE FPL 2006. She is a member of ACM.

Nikil Dutt (F'08) received the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign, Urbana-Champaign, in 1989.

He is currently a Chancellor's Professor with the University of California, Irvine, with academic appointments in the Computer Science and Electrical Engineering and Computer Science Departments. His research interests include embedded systems, electronic design automation, computer architecture, optimizing compilers, system specification techniques, and distributed embedded systems.

Prof. Dutt was a recipient of Best Paper Awards from CHDL89, CHDL91, VLSIDesign2003, CODES+ISSS 2003, CNCC 2006, and ASPDAC-2006. He currently serves as Editor-in-Chief of the *ACM Transactions on Design Automation of Electronic Systems (TODAES)* and as an Associate Editor of the *ACM Transactions on Embedded Computer Systems (TECS)* and of the IEEE Transactions on Very Large Scale Integration (VLSI) Systems (IEEE T-VLSI). He was an ACM SIGDA Distinguished Lecturer during 2001–2002, and an IEEE Computer Society Distinguished Visitor for 2003–2005. He has served on the steering, organizing, and program committees of several premier CAD and Embedded System Design conferences and workshops, including ASPDAC, CASES, CODES+ISSS, DATE, ICCAD, ISLPED, and LCTES. He serves or has served on the advisory boards of ACM SIGBED and ACM SIGDA, and previously served as Vice-Chair of ACM SIGDA and of IFIP WG 10.5. He is an ACM Distinguished Scientist and an IFIP Silver Core Awardee.