SLOPES: Hardware–Software Cosynthesis of Low-Power Real-Time Distributed Embedded Systems With Dynamically Reconfigurable FPGAs

Li Shang, *Member, IEEE*, Robert P. Dick, *Member, IEEE*, and Niraj K. Jha, *Fellow, IEEE*

***Abstract*—In this paper, we present a multiobjective hardware–software cosynthesis system, called SLOPES, for multirate low-power real-time distributed embedded systems consisting of dynamically reconfigurable field-programmable gate arrays (FPGAs), processors, and heterogeneous communication resources. This cosynthesis algorithm simultaneously optimizes system price and average power consumption. First, we present an evolutionary algorithm that automatically determines the quantities and types of system resources, assigns tasks to different potentially reconfigurable processing elements, and assigns communication events to communication resources. Second, we propose a dynamic priority multirate scheduling algorithm to determine the times at which all the tasks and communication events in the system occur. This two-dimensional scheduling algorithm determines task priorities based on real-time constraints and detailed frame-by-frame FPGA reconfiguration overhead information. Experimental results indicate that the proposed method reduces schedule length by an average of 34.3% and reconfiguration energy by an average of 40.4%, compared to a method that does not consider the effect of partial reconfiguration during synthesis. SLOPES yields multiple system architectures that tradeoff system price and average power consumption under real-time constraints.**

***Index Terms*—Hardware–software co-design, low-power design, reconfigurable architectures, system-level synthesis.**

I. Introduction

THIS PAPER describes algorithms to synthesize low-power low-price embedded systems containing dynamically reconfigurable field-programmable gate arrays (FPGAs). This algorithm is most closely related to two research fields: hardware–software cosynthesis and reconfigurable computing. The proposed algorithms optimize a system-level hardware–software architecture, taking care to properly use dynamically reconfigurable FPGAs to reduce system price and average power consumption under hard real-time constraints.

Manuscript received June 11, 2004; revised June 2, 2005. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract DAAB07-00-C-L516 and in part by the National Science Foundation (NSF) under Award CNS-0347941. This paper was recommended by Associate Editor R. Gupta.

L. Shang is with the Department of Electrical and Computer Engineering, Queen's University, Kingston, ON K7L 3N6, Canada (e-mail: li.shang@queensu.ca).

R. P. Dick is with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208 USA (e-mail: dickrp@eecs.northwestern.edu).

N. K. Jha is with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA (e-mail: jha@princeton.edu).

Digital Object Identifier 10.1109/TCAD.2006.883909

Hardware–software cosynthesis algorithms automatically produce hardware–software architectures for distributed embedded systems. Ideally, they minimize multiple costs, such as execution time, price, and average power consumption. Given a specification, a hardware–software cosynthesis algorithm must select different processing elements (PEs) and communication resources to use in the embedded system (*allocation*), determine which resource will be used to carry out each portion of the specification's computation and communication (*assignment*), and produce a schedule for all of the specification's computation and communication (*scheduling*). Thus, given an embedded system specification, a cosynthesis algorithm produces a detailed description of an architecture that meets the design constraints and optimizes a set of costs. Prior to the work described in this paper, hardware–software cosynthesis algorithms had not considered the use of partially reconfigurable FPGAs for low-power embedded systems. However, researchers had considered using fully reconfigurable FPGAs in the synthesis of low-price embedded systems [1]–[7]. This previous work did not consider power consumption and did little to exploit partial reconfiguration.

FPGAs are commonly used in distributed embedded systems. They share many traits with application-specific integrated circuits (ASICs); they are parallel hardware platforms. However, they have the advantages of reducing design time and supporting dynamic (runtime) reconfiguration. Although FPGAs typically have lower performance, higher power consumption, and higher energy consumption when compared with ASICs, for many applications, they have substantially better performance, average power consumption, and energy than general-purpose processors [8]–[14]. When a design uses dynamic reconfiguration, it is important to minimize the overhead, i.e., time and energy consumption, associated with reconfiguration. To reduce this overhead, many new reconfigurable architectures have been proposed [15]–[19]. In modern dynamically reconfigurable FPGAs, the embedded configuration storage circuitry can be updated selectively in a few clock cycles without disturbing the execution of the remaining logic. These new designs have increased the potential benefit of using dynamically reconfigurable FPGAs in low-power embedded systems by dramatically reducing the performance and energy penalties of dynamic reconfiguration. However, these costs are still substantial.

With the success of battery-powered personal computing devices and wireless communication systems, reducing average power consumption has become a key goal in embedded system

design. For the systems under consideration, the total runtime depends on specifications provided by the designer, although individual tasks may be executed for different durations depending on synthesis decisions. Therefore, average power consumption and energy consumption within a period of operation are proportional; for the systems under consideration, reducing either one reduces the other in proportion.

Although their flexibility makes dynamically reconfigurable FPGAs a good solution for portable applications, their energy consumption cannot be neglected. Online reconfiguration consumes energy as well as time. Moreover, reconfiguration energy consumption can account for a substantial fraction of FPGA energy. FPGAs require energy for both execution and reconfiguration. This makes FPGA energy optimization more complex than processor and ASIC energy optimization, which only have execution energy consumption.

In summary, recent changes in FPGA technology have made the use of partially reconfigurable FPGAs in low-power embedded systems dramatically more promising. This paper describes a new dynamically reconfigurable embedded system synthesis algorithm based on an evolutionary hardware–software cosynthesis algorithm and a novel scheduling algorithm for partially reconfigurable FPGAs.

A. Previous Work

A number of researchers have worked on the hardware–software cosynthesis problem. In this section, we will survey some of the work most relevant to the proposed algorithm, i.e., the synthesis of heterogeneous distributed embedded systems without tight limits on resource allocations. Bender solved a simplified version of the hardware software cosynthesis problem with mixed-integer linear programming (MILP) [20]. He used a linear weighting sum to combine execution time, processor prices, and communication resource prices. Although the approach is optimal, it must use a suboptimal heuristic preprocessing stage to have any chance of solving complicated (realistic) problems in a reasonable amount of time. Dave *et al.* used a constructive algorithm to solve the classical multirate distributed system cosynthesis problem. This work was extended to target low-power embedded systems [21] and hierarchical embedded systems [22]. Hsiung developed a hardware–software cosynthesis algorithm for massively parallel homogeneous software applications that enumerates solutions in a small set [23]. Solutions that do not satisfy the specified constraints are eliminated. Jeong *et al.* developed a hardware–software cosynthesis algorithm that allows the use of incrementally dynamically reconfigurable hardware [4]. Karkowski and Corporaal allocated and partitioned an ANSI-C specification among homogeneous processors on a single chip [24]. Kuchcinski used constraint logic programming to minimize the price of an embedded system under time constraints [25]. The computational complexity of his algorithm may be reduced, as long as one is willing to tolerate suboptimal solutions. Lee *et al.* developed an A$\star$ search algorithm to optimize embedded system resource allocations [26]. This algorithm uses earliest deadline first scheduling integrated with a load balancing assignment algorithm borrowed from behavioral synthesis. It does not model intertask dependences. Oh and Ha developed an iterative algorithm targeting the heterogeneous distributed system cosynthesis problem [27]. Prakash and Parker (P&P) developed a MILP solver for the distributed hardware–software cosynthesis problem [28]. Schwiegershausen and Pirsch developed a MILP solver for the heterogeneous distributed system synthesis problem [29]. Srinivasan and Jha developed a heuristic constructive algorithm that synthesizes fault-tolerant real-time distributed embedded systems [30]. Teich *et al.* applied an evolutionary algorithm to the cosynthesis problem. They repaired bad solutions instead of avoiding their creation [31]. Their algorithm optimized period and price in the absence of hard real-time constraints. Wolf developed a fast greedy iterative improvement algorithm for the classical cosynthesis problem [32]. His algorithm may be used to model communication. However, in the presence of nonzero communication times, this algorithm is no longer guaranteed to produce the minimal cost solutions that meet deadlines. Yen and Wolf developed an iterative improvement algorithm for the hardware–software cosynthesis problem [33].

Most previous hardware–software cosynthesis algorithms do not consider the use of dynamically reconfigurable logic. In those that do [1]–[7], power consumption is not optimized, and partial dynamic reconfiguration is not fully exploited. In Dick and Jha's work, multiple tasks may not execute concurrently on the same FPGA [1]. Jeong *et al.* use an optimal MILP formulation and cannot synthesize large embedded systems in a practical amount of time [4]. The cosynthesis system proposed by Noguera and Badia uses prefetching techniques to minimize reconfiguration latency [7]. However, this approach overlooks power consumption. In addition, many algorithms make the simplifying assumption that the target embedded system consists of one processor and one FPGA [5], [6].

Past work has considered the use of dynamically reconfigurable FPGAs in high-level synthesis [34]–[36], [70]. However, in system-level synthesis, the problem is much more complex. The execution time, energy consumption, and reconfiguration overhead for each task, as well as the resource utilization and reconfiguration conditions of the FPGAs, must be considered. The complexity of the scheduling problem, which is known to be $\mathcal{NP}$-complete [37], is increased because it must be extended from the time domain to both time and space domains. The problem is further complicated by making use of partial, instead of full, dynamic reconfiguration.

B. Our Approach and Contributions

We propose to use an evolutionary algorithm to tackle the problems of allocation and assignment. Algorithms in this class have been shown to rapidly produce high-quality solutions for the cosynthesis problem [1], [38], [39]. Multiobjective system requirements can be simultaneously optimized. No arbitrary limitations are imposed on the quantities and types of system resources. However, resource use is minimized as a consequence of minimizing energy consumption and price. The optimization infrastructure described in this paper shares features with an evolutionary algorithm appearing in a coauthor's doctoral dissertation [40]. However, unlike that work, the algorithm

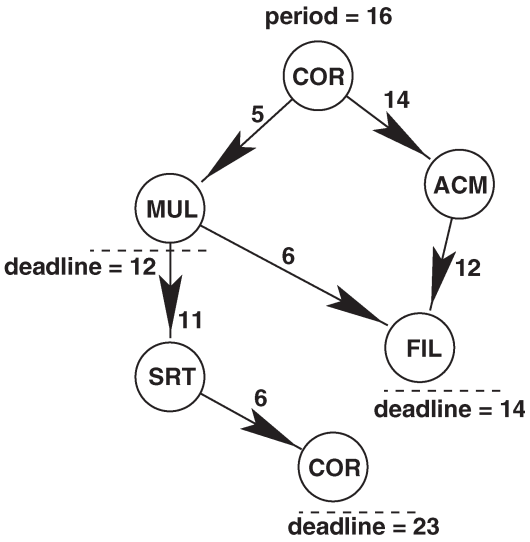


Fig. 1. Task graph.

described in this paper supports partial dynamic reconfiguration of FPGAs.

Since scheduling is performed in the inner loop of cosynthesis, a scheduler with a low time complexity is required. In addition, efficient methods for reducing the delay and energy overheads of dynamic reconfiguration are required. We propose a 2-D multirate cyclic scheduling heuristic. The scheduler uses resource and frame-by-frame reconfiguration information in its efforts to globally minimize the time and energy consumption resulting from reconfiguration, execution, and communication.

The proposed reconfigurable hardware–software cosynthesis algorithm tightly integrates the allocation, assignment, and scheduling optimization problems. It simultaneously optimizes system price and energy consumption under real-time constraints, producing multiple solutions that trade off these costs. The rest of this paper is organized as follows. In Section II, we define the terms and models used in our cosynthesis system. In Section III, we present an overview of the cosynthesis system and optimization infrastructure. In Section IV, we describe the scheduling algorithm. We present experimental results in Section V and conclude in Section VI.

II. Preliminaries

In this section, we define the concepts and models used in our cosynthesis system.

A. Input Specification

The input specification is a set of task graphs (see Fig. 1). A task graph is a directed acyclic graph in which a node is a task, i.e., a portion of the computation an embedded system is required to carry out. Matrix multiplication is an example of a task type. Edges represent communication events. An edge between tasks represents a data dependence and is labeled with the quantity of data transmitted. Each task graph has a period, which represents the interval between the earliest start times of its consecutive executions. In real-time systems, hard deadlines are associated with some tasks. A *multirate* embedded system contains task graphs with different periods. The least common multiple of these periods is the *hyperperiod*. A valid and irredundant static schedule spans a system's hyperperiod [41].

B. Resource Library Model

In addition to task graphs, a cosynthesis algorithm needs information from a resource library. This library contains general-purpose processors, dynamically reconfigurable FPGAs, communication links, and memories that can be used for cosynthesis. We model two types of PEs: processors and FPGAs

1) *Processor:* A processor is a general-purpose PE used to carry out tasks. A processor may represent a microprocessor, a microcontroller, a digital signal processor (DSP), or an application-specific instruction processor. Each processor has a price, average static power consumption, and a variable indicating whether or not it has a communication buffer. Processors with communication buffers may concurrently execute tasks and communicate data with other PEs. For each task–processor pair, there is a worst-case execution time and memory load. *Worst-case execution time* is the maximum amount of time a processor may require to carry out the task. *Memory load* is the amount of memory required by the task during execution. This variable accounts for instruction and data space. The information for each task, e.g., execution time and average power consumption, can be determined using techniques presented in the literature [42]–[44].
2) *Dynamically reconfigurable FPGAs:* FPGAs are high-density programmable logic devices for which programming is typically conducted by streaming a configuration into static random access memory distributed within the FPGA. For appropriate tasks, FPGA-based implementations improve energy/time efficiency by an order of magnitude in comparison with general-purpose processors [8]–[14]. Designs based on instruction processors and FPGAs have been widely used in embedded systems. In most such designs, the FPGA is used to implement the tasks that consume the most time and energy. Runtime reconfiguration will make it more practical for FPGAs to support numerous tasks. Many types of FPGAs have long supported dynamic in-circuit reprogramming. However, high reconfiguration energy consumption, long reprogramming times, and the difficulty of manually designing multimode systems have limited the use of dynamic reconfiguration in the industry. Recent improvements in FPGA technology, such as partial reconfiguration, have made dynamic reconfiguration significantly faster and reduced the associated energy consumption. Commercially available reconfigurable devices supporting partial dynamic reconfiguration include Virtex [19] from Xilinx, FPSLIC from Atmel [45], and XPP64-A1 from PACT [46]. We will focus on examples of the Virtex family.

In 1993, Xilinx acquired Algotrinix Ltd. and adapted their Configurable Array Logic architecture, producing the XC6200 family of FPGAs. This was the first widely sold product

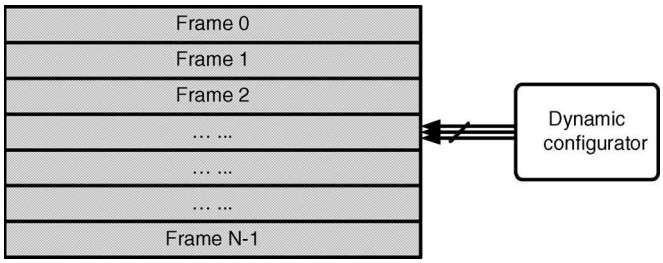


Fig. 2. Dynamically reconfigurable FPGA model.

supporting partial reconfiguration. The XC6200 uses a 2-D reconfiguration model in which the configuration of each individual on-chip logic component, for instance, configurable logic blocks (CLBs), can be modified at runtime. The XC6200 family was discontinued, partially due to the complexity of this 2-D reconfiguration architecture. Instead, in the Virtex and Virtex-II families, Xilinx uses a 1-D reconfiguration model, as shown in Fig. 2, to achieve a better tradeoff between flexibility and hardware complexity. In this model, FPGAs are reconfigured at the frame level, i.e., the frame is the atomic reconfiguration unit. Table I shows the number of frames required by various on-chip resources in Virtex XV50 FPGA. The reconfiguration of a frame does not disturb the execution of other frames. A task may reuse a configuration pattern left behind by an earlier task. Only one frame may be reconfigured at a time. Each ready task needs to be loaded into contiguous frames in the FPGA reconfiguration memory before its execution. For each frame, the task has a specific configuration pattern. If the required configuration pattern cannot be found in the corresponding frame in the FPGA, a *pattern miss* is said to occur. FPGA configuration storage areas are analogous to memory caches in computers: Compulsory, conflict, and capacity misses can also occur in the reconfiguration memory of FPGAs.

The power consumption of FPGAs may be divided into two categories: execution power and reconfiguration power. Estimating FPGA execution power consumption is similar to ASIC power analysis. Inside FPGAs, the logic cells and interconnect fabric have very regular structures. Therefore, the capacitance of each individual reconfigurable component can be determined [47]. Capacitance and timing information can be extracted from a low-level specification or implementation. Combining this information with real-delay timing simulation [47]–[49], which is used to characterize switching activity, allows the execution power of FPGAs to be predicted with high accuracy. For frame-level dynamically reconfigurable FPGAs [19], reconfiguration power is proportional to configuration frequency.

The following example provides some insight into the performance, power consumption, and energy consumption consequences of using dynamically reconfigurable FPGAs. Using the Xilinx Virtex XVC1000 FPGA, a 32-bit multiplier using parallel adders requires approximately 1600 CLBs. It functions at 50 MHz, and the power consumption is approximately 430 mW. Although many FPGAs contain built-in multipliers, we have used a multiplier in this example to illustrate the challenge of runtime reconfiguration with a commonly understood functional unit. Note that Virtex FPGAs only support 1-D frame-based reconfiguration. Therefore, all of the CLBs in seven columns must be configured together, even if only a subset is used. The Virtex XVC1000 contains 72×108 CLBs. Even assuming all the 72 CLBs in each column are used, which is typically not the case, $\lceil 1600/72 \rceil = 23$ columns are required. Each CLB column requires at least 48 frames, and each frame contains 1376 configuration bits. Therefore, it is necessary to transmit more than 1.52 million configuration bits. Using the SelectMap reconfiguration interface with the highest configuration throughput, i.e., 400 Mb/s, still requires 3.6 ms of configuration time. Configuring the whole FPGA pushes the configuration time to 18 ms. The power consumption during reconfiguration is approximately 250 mW. Thus, it is clearly critical to minimize configuration overhead by using partial reconfiguration, reusing configuration frames, and prefetching frames, etc.

The following parameters are defined for each dynamically reconfigurable FPGA in the resource library: price, number of configuration frames, reconfiguration bandwidth, number of reconfiguration bits for each frame, number of inputs/outputs, idle power, and reconfiguration power per frame. For each task, the worst-case execution time, average power consumption, and memory requirement to store reconfiguration and computation data on each FPGA type in the resource library are specified.

- *Communication resources:* Communication resources connect PEs to each other. Each communication resource is described by a price, packet size, average energy consumption per packet, worst-case communication time per packet, idle power consumption, and contact count. Contact count is the number of PEs to which a communication resource is capable of connecting. Communication resources are sequential resources: Only one pair of communicating PEs may use a communication resource at a time. Busses (to minimize price) and point-to-point links (to minimize communication contention) are supported. The communication network is heterogeneous. However, bus segmentation is not modeled.
- *Memory:* Our memory model assigns each memory block a price and capacity. The memory requirements for computation and communication are specified for each task type.

C. Target Architecture

We assume a target architecture composed of computation and storage elements, e.g., processors, FPGAs, and memory modules. These devices are connected by an arbitrary topology network of communication resources, i.e., the processors and FPGAs are nodes, and communication resources are (hyper) edges in a hypergraph. Instruction processor and FPGA memories are not shared, i.e., communication is explicit. To guarantee that all hard real-time deadlines are met, we generate a system-wide schedule at synthesis time. Therefore, devices must be prevented from starting the execution of tasks too early. This can be accomplished by using a global controller that signals PEs at the appropriate times or by simple local timing logic, such as counters, which may be synchronized using a global signal. Note that fine-grained synchronization among PEs is not

TABLE I
CONFIGURATION FRAMES IN VIRTEX XV50

Column Type	Center	CLB	IOB	Block SelectRAM interconnect	Block SelectRAM content
Number of frames	8	48	54	27	64

necessary. A low-frequency synchronization clock can be used by adding the duration of the worst-case discretization error (the synchronization period) to task execution times.

D. Optimization Terms

Our tool, SLOPES, determines the following portions of an embedded system architecture in an attempt to meet the specified timing constraints while minimizing price and average power consumption: PE and communication resource allocations, communication resource connectivity, assignments of tasks and communication events to resources, and schedules of tasks and communication events. A *PE/communication resource allocation* is the number of each type of general-purpose processor and FPGA/communication resource in an architecture. *Task/communication event assignment* indicates the PE or communication resource upon which each task or communication event is executed. *Communication resource connectivity* indicates the PEs to which each communication resource is connected. In addition, SLOPES generates a schedule for the tasks assigned to each PE and the communication events assigned to each communication resource. Resource contention is explicitly modeled. A static schedule based on worst-case execution times and communication times is used to determine whether all tasks meet their deadlines. When tasks are executed on general-purpose processors, memory is required for instructions and data. Similarly, FPGAs require memory to store configurations and data. SLOPES accounts for these requirements when computing an architecture's price and average power consumption.

An architecture's *cost set* characterizes the quality of the architecture. A cost set contains the number of tasks that could not be scheduled within the system hyperperiod, the number of communication events that could not be scheduled within the system hyperperiod, the degree to which the specification's task deadlines were violated, the degree to which FPGAs were overused (overallocated), as well as the price and average power consumption of the system architecture.

E. Energy Minimization

The energy consumption of the synthesized system is minimized throughout the synthesis process. Allocation/assignment determine a power/energy-efficient system architecture. Note that, for each computation/communication task, using different resources results in different power/energy consumption values. Our scheduler minimizes the schedule length to make power/energy-efficient solutions feasible and minimizes the average reconfiguration energy consumption.

PE static and dynamic power dissipation, communication resource power dissipation, and FPGA reconfiguration power consumption are considered; the energy impact of resource allocation, assignment of tasks to PEs, assignment of communication events to communication resources, and scheduling is calculated, allowing appropriate global decisions for energy consumption minimization.

In addition to using energy consumption to guide the global optimization algorithm when changing solutions, energy-aware heuristics are used in a number of stages within the optimization algorithm to increase the probability that high-quality solutions will be provided to multiobjective Boltzmann trials for selection. For example, the relative energy consumption and performance of tasks executing on the available PEs is used to guide task assignment. Note that these heuristics are not necessary to ensure high solution quality. However, they accelerate the global optimization algorithm by focusing exploration on the most promising areas of the solution space, allowing higher solution quality for a given amount of optimization time.

The task/PE and communication resource energy consumption models are general and easily customizable. The FPGA power model was briefly introduced in Section II-B. Note that the process of generating these models is beyond the scope of this paper but may be found in other works [47], [50], [51].

III. HARDWARE–SOFTWARE COSYNTHESIS OVERVIEW

The hardware–software cosynthesis problem is hard; large instances have only been solved using potentially suboptimal heuristics. This section explains the reasons for using a parallel recombinative simulated annealing algorithm [52] for synthesis in SLOPES. In addition, we describe this optimization infrastructure. The curious reader is directed to the literature for a more detailed introduction to multiobjective optimization of heterogeneous distributed systems, a more complete survey of the hardware–software cosynthesis research area, and a tutorial on its relationship with search and optimization [40].

The hardware–software cosynthesis problem is composed of multiple interdependent $\mathcal{NP}$-complete problems. The allocation/assignment problem and the scheduling problem are both known to be $\mathcal{NP}$-complete [37]. To our knowledge, all existing optimal hardware–software cosynthesis algorithms solve a constrained version of the problem (e.g., one-CPU one-ASIC systems [53]–[56]) or face insurmountable performance degradation when run on large problem instances; although useful for small problem instances, optimal solvers are not capable of tackling large problem instances, e.g., those containing more than ten or so tasks [28], [29]. Existing hardware software cosynthesis systems capable of handling large instances of the general heterogeneous system synthesis problem rely on potentially suboptimal heuristics [25], [57], [58]. The optimization infrastructure used by SLOPES falls in this class: It does not guarantee optimality. However, as we will describe in the following section, for problem instances to which optimal solutions are known, the optimization infrastructure used by SLOPES also arrives at optimal solutions.

A. Optimization Infrastructure Design Considerations

SLOPES requires an optimization infrastructure with the following attributes: 1) fast enough to solve large problem instances, 2) well suited to solving multiobjective problems, 3) capable of using problem-specific information to accelerate optimization, and 4) capable of finding optimal or near-optimal solutions.

As noted above, the hardware–software cosynthesis problem requires multiple $\mathcal{NP}$-complete problems to be solved. In practice, guaranteed optimal solvers for this problem domain are capable of handling only small problem instances. Therefore, we ruled out the use of guaranteed optimal algorithms. Iterative improvement algorithms that maintain a pool of solutions, e.g., genetic algorithms, meet the second requirement. However, variants of more commonly used algorithms, e.g., simulated annealing, in which a pool of solutions simultaneously exist, can also be used to solve multiobjective problems. We ultimately implemented an optimization infrastructure based on genetic algorithms and simulated annealing that avoids reevaluating solutions it has previously encountered. It was straightforward to incorporate problem-specific heuristics into this algorithm by adding intelligence to the operators that change solutions. Note that, although we used heuristics to guide changes, we chose to make this guidance stochastic to avoid constraining the region of the solution space the algorithm is capable of exploring.

To test the quality of our evolutionary hardware–software cosynthesis algorithm, we compared it [38], [40] with a number of other algorithms, e.g., constructive, iterative improvement, MILP, and simulated annealing algorithms. For the problem instances that are small enough for optimal algorithms to solve (e.g., MILP [28]), our optimization infrastructure also arrives at optimal solutions, often in orders of magnitude less CPU time. When run on larger problem instances previously tackled by constructive [21] and iterative improvement algorithms [59], in all cases, our optimization infrastructure produces results that are as good as, and frequently better than, prior work. Our runtimes are either better (by up to four orders of magnitude) or low enough to render differences unimportant (CPU seconds). See Section V-B for more details.

B. Evolutionary Algorithm

In this section, we describe the evolutionary algorithm used by SLOPES to optimize resource allocations, task and communication event assignments, as well as communication resource connectivities.

A genetic algorithm maintains a pool of solutions that evolve in parallel over time. During each *generation*, genetic operators that allow randomized local changes, and the exchange of information between solutions, are applied to the solutions in the current pool to improve them. The lowest quality solutions are then removed from the pool [60].

In an iterative improvement algorithm, we define *greediness* as the probability that a cost-decreasing change to a solution will be preferred instead of a cost-increasing change. Simulated annealing algorithms are iterative improvement algorithms in which greediness increases during the run of the algorithm [61].

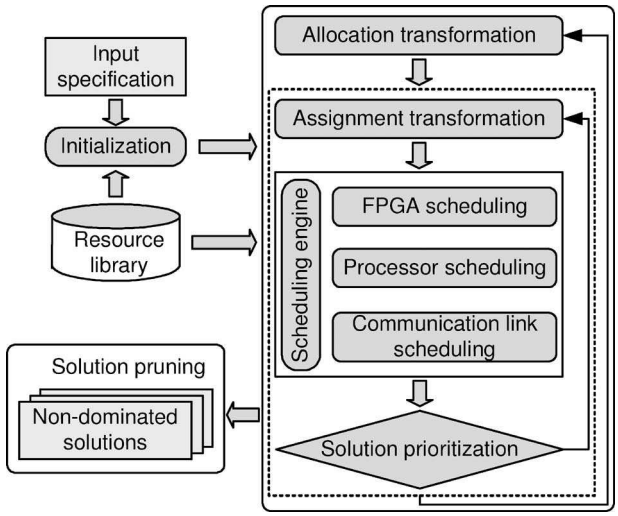


Fig. 3. Hardware–software cosynthesis overview.

Parallel recombinative simulated annealing (PRSA) algorithms [52] have some of the best attributes of both genetic algorithms and simulated annealing algorithms [52]. This class of algorithms is best understood to contain genetic algorithms that use Boltzmann trials between modified and existing solutions to select the solutions that will exist in the next generation. The greediness of a PRSA algorithm starts low and increases during an optimization run, allowing it to escape local minima in a fashion similar to simulated annealing. SLOPES uses a PRSA optimization algorithm. Moreover, it is a Pareto-rank-based multiobjective optimization algorithm [62], as described in Section III-D.

An overview of our cosynthesis system is shown in Fig. 3. Cosynthesis solutions are organized in clusters. Solutions within a cluster share the same allocation of hardware resources but have different assignments of tasks to resources. At the start of the algorithm's run, solutions are initialized. Then, evolution operators, i.e., reproduction, mutation, and crossover, are used to transform allocations and assignments, producing the next generation of solutions. Within each cluster, the assignment information may be mutated or traded between different solutions. Allocation information may be mutated or traded between different clusters. The ranks of solutions are determined using Pareto ranking in a multidimensional space within which system price, average power consumption, and other costs are dimensions. A solution's *rank* is equal to the number of other solutions that do not dominate it, i.e., a solution dominates another if it is better in both average power consumption and system price. When a prespecified number of generations has passed without improvement, invalid solutions, i.e., those that do not meet the deadlines, are pruned out, and the remaining nondominated solutions are reported to the system designer (the concept of domination is defined in Section III-D).

C. Mutation and Crossover

Mutation makes randomized local changes to an architecture. When an architecture mutates, SLOPES first determines whether the task assignment or communication resource connectivity will mutate. A random variable w between 0 and the

average number of contacts of a communication resource is selected. If w is greater than 1, the task assignment mutates; otherwise, the communication resource connectivity mutates. t_count is the number of tasks in the embedded system specification multiplied by the global temperature. Task assignment mutation causes a randomly selected set of t_count tasks to be reassigned to randomly selected PEs. c_count is the number of communication resources in the architecture multiplied by the global temperature. Communication resource connectivity mutation causes c_count communication resources to disconnect all of their contacts and randomly reconnect them to PEs.

When a cluster mutates, SLOPES first determines whether the PE allocation or communication resource allocation will mutate. A random variable x between 0 and the average number of contacts on a communication resource is selected. If x is greater than 1, the PE allocation mutates; otherwise, the communication resource allocation mutates. Cluster mutation may cause an instance of a randomly selected PE type to be added to all the architectures in the cluster, or it may cause a randomly selected PE to be removed from all the architectures in the cluster. The probability of an additive mutation is related to the global temperature maintained by SLOPES and varies from 1 to 0 during the run of the algorithm. All of the architectures in the cluster randomly change the parts of their task assignments and communication resource connectivities that depend on the lost PE such that none of the tasks or communication resources depend on the lost PE. Communication resource allocation mutation is analogous to PE allocation mutation.

Crossover is the process of trading or swapping information between two architectures. Crossover may be applied to PE allocations, communication resource allocations, task assignments, and link connectivities. During crossover, a portion of the information is swapped between the pair of solutions. The method of selecting the traded information is a matter of some importance in evolutionary algorithm design. However, it appears in other work [40] and has been omitted from this paper for the sake of brevity.

D. Ranking and Reproduction

In this section, we explain the Pareto ranking method of imposing a partial order on solution quality. Pareto ranking has a number of interesting properties elaborated on in the literature [38], [62].

The number of clusters and solutions maintained by SLOPES is conserved during one run of the algorithm. For each cluster or solution created via reproduction, another is eliminated. The number of solutions and clusters maintained during a run can be chosen at the start of the run. We typically use 20 clusters, each of which contains 20 solutions.

A solution *dominates* another if some of its features are equal and others (at least one) are better, i.e., given costs $a_1, a_2, \ldots, a_n$ for solution a and costs $b_1, b_2, \ldots, b_n$ for solution b, a dominates b ($\mathrm{dom}(a, b) = 1$) if and only if

$$\forall_{i=1}^{n} a_i \leq b_i \wedge \exists_{j=1}^{n} a_j \neq b_j. \tag{1}$$

A solution's *Pareto rank* is the number of other solutions, in the solution pool, that do not dominate it. Given a solution pool of size p, calculating Pareto rank is an $O(p^2)$ operation; each solution must be compared with every other solution. In Fig. 4, each circle represents a solution. Each solution's price and average power consumption are indicated by the position of its circle in the graph. The number in each circle indicates the Pareto rank of the associated solution.

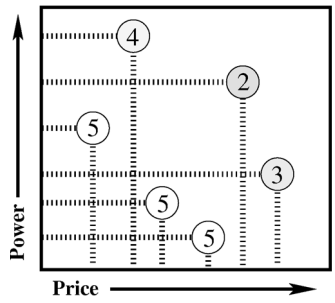


Fig. 4. Pareto rank.

Ranking clusters is more complicated than ranking solutions. Each solution has one set of costs. Thus, determining whether it dominates another solution is straightforward. Clusters, however, contain numerous solutions; each cluster is associated with many sets of costs. We extend the concept of domination to take partial domination into account. Cluster domination, cdom, is represented by a scalar instead of a Boolean value. The definition of rank must also be adjusted when it is applied to clusters. A *noninferior* solution is one that is not dominated by any other solution. Let x and y be clusters. $\mathrm{nis}(x)$ is the set of noninferior solutions in x. $\mathrm{dom}(a, b)$ is 1 if a is not dominated by b and 0 otherwise. cdom is a function of two clusters, i.e.,

$$\mathrm{cdom}(x, y) = \max_{a \in \mathrm{nis}(x)} \sum_{b \in \mathrm{nis}(y)} \mathrm{dom}(a, b) \tag{2}$$

and

$$\mathrm{rank}[x] = \sum_{y \in \text{all clusters},\ y \neq x} \mathrm{cdom}(x, y). \tag{3}$$

Once cluster ranks have been determined, cluster reproduction is analogous to solution reproduction. A prespecified number of clusters is removed to make room for high-rank clusters to reproduce. Clusters are selected for reproduction in the same manner as solutions. Cluster crossover and mutation are also analogous to solution crossover and mutation.

After cluster reproduction, mutation, and crossover, SLOPES ranks all clusters relative to each other. Every architecture in the system is ranked relative to every other architecture, as described above. Each cluster's rank is the sum of the ranks of the architectures contained within it.

E. Boltzmann Trials

Given two ranks J and K and the global temperature T, a Boltzmann trial preserves the architecture associated with

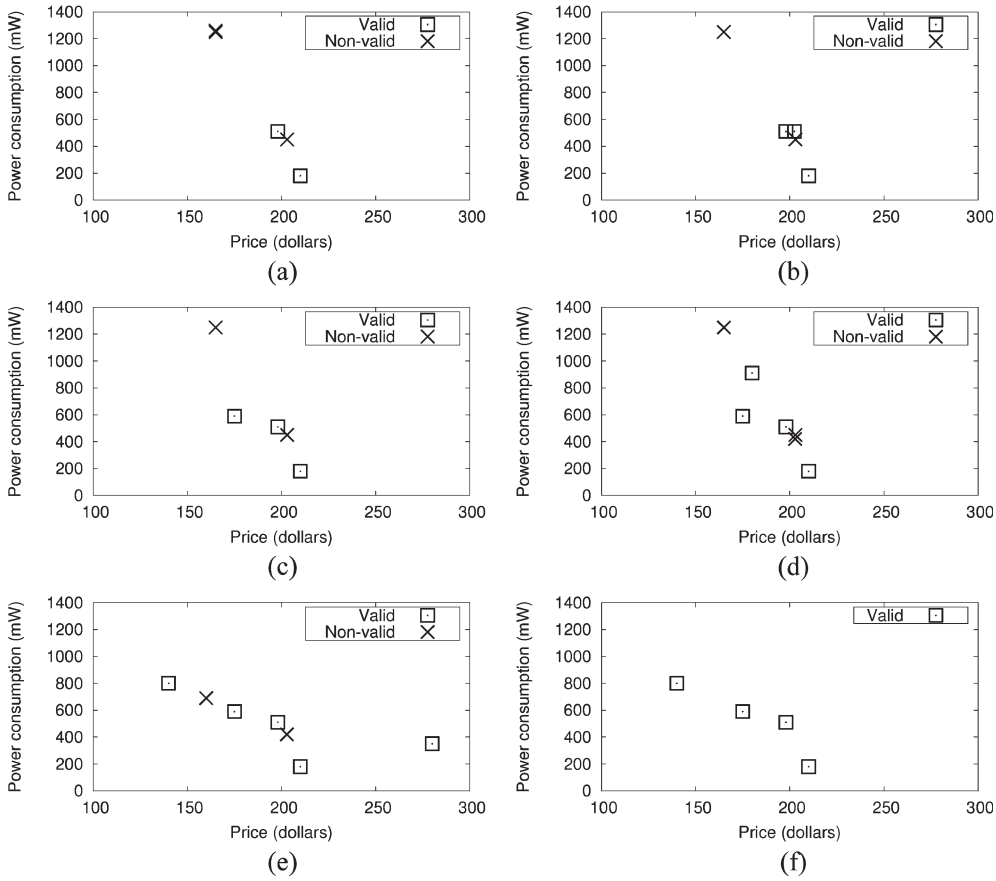


Fig. 5. Optimization process of our evolutionary algorithm.

J and eliminates the architecture associated with K with probability

$$\frac{1}{1+e^{(K-J)/T}}. \tag{4}$$

After ranking architectures, SLOPES conducts interarchitecture Boltzmann trials between randomly selected pairs of architectures within a cluster, eliminating the loser, until the cluster contains the same number of architectures as it did before reproduction. Intercluster Boltzmann trials are analogous to interarchitecture Boltzmann trials. The use of a global temperature-dependent criterion for eliminating solutions allows SLOPES to escape local minima early in its run while the global temperature is still high. As the global temperature decreases, SLOPES becomes increasingly greedy.

After reproduction, crossover and mutation are carried out on the solutions that were copied. The number of crossovers and mutations per generation, for each type of string, is specified by user-defined parameters. Crossover is applied to solution pairs that are randomly selected from the solutions created by reproduction. Mutation is applied to solutions that are also randomly selected from the solutions created by reproduction.

F. Evolutionary Optimization Demonstration

We demonstrate the optimization process of our evolutionary algorithm with a set of system-level specifications generated by task graphs for free (TGFF) [63], a structured random task graph generator. The characteristics of the processors, memory blocks, and communication resources were taken from past work, publicly available examples [63], and various benchmarks [64]. The dynamically reconfigurable FPGA model is based on Xilinx Virtex-E FPGAs [19].

Fig. 5 traces an example system synthesis and optimization run. Typically, the evolutionary algorithm maintains hundreds of solutions in the solution pool. For simplicity, we only show a subset of the solutions in this figure, in which squares denote valid solutions for which hard deadlines are all met. Crosses depict invalid solutions, in which some hard deadlines are not met. The initial solutions are shown in Fig. 5(a). Then, from Fig. 5(b)–(e), during each generation, new solutions are produced, and good solutions are kept. The final solutions are shown in Fig. 5(f). These solutions are all valid and nondominated. The optimization algorithm moves solutions toward the Pareto-optimal curve. System designers can choose the desired solution based on the relative importance of different costs in their target applications, i.e., average power consumption and system price. It is sometimes best to defer this decision until the end of the optimization run, at which time the available tradeoffs between costs are known.

IV. Scheduling Algorithms

The scheduling algorithm is invoked in the inner loop of cosynthesis, in a cycle with evolutionary changes to allocation and assignment. Tasks and communication events need to be

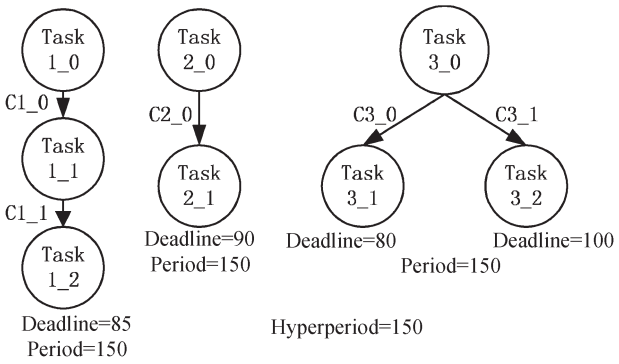


Fig. 6. Task graphs.

TABLE II
ALLOCATION AND ASSIGNMENT INFORMATION

Processor 1	Processor 2	FPGA	Bus
2_1	3_2	other tasks	C2_0, C3_1

TABLE III
TASK EXECUTION (IN MICROSECONDS)

Task	1_0	1_1	1_2	2_0	2_1	3_0	3_1	3_2
Worst-case execution time	33	11	25	50	20	26	33	37

scheduled on different processors, FPGAs, and communication resources. Processors and communication resources must be used sequentially. Therefore, each has a 1-D schedule. However, scheduling for dynamically reconfigurable FPGAs is a 2-D problem: FPGA schedules span both time and space.

A. Motivational Example

We next present an example to introduce and illustrate our scheduling algorithm.

Example 1: Consider a system specification with the three example task graphs shown in Fig. 6. The allocation and assignment information for each task and communication event is shown in Table II. Tasks 1_0 and 3_1 have the same configuration patterns, whereas the configuration patterns for other tasks differ. The reconfiguration time for each frame is 3.4 μs. Based on the allocated PEs, the worst-case execution time for each task is shown in Table III. The communication events C3_1 and C2_0 are executed on the bus that links the three PEs. We explicitly model contention for shared communication resources. The communication event durations are 15 and 10 μs, respectively. We follow the traditional assumption in distributed computing that communication between tasks assigned to the same PE takes negligible time. Two different scheduling approaches are applied to these task graphs, as will be described later. The first approach is based on prior work, and the second is our proposed approach.

Scheduling Approach I: The *scheduling sequence*, i.e., the order in which tasks are scheduled, is based on static slack-based priority [21]. Task i's priority is given by

$$P_i = -(\mathrm{LST}_i - \mathrm{EST}_i) \tag{5}$$

where LST_i is the latest start time of task i, and EST_i is its earliest start time. These two values are computed by conducting forward and reverse topological sorts of the task graphs starting from both the source and sink nodes.

Configuration patterns may be loaded into the FPGA before the task ready time, in a process analogous to cache prefetching. Configuration patterns remaining after the execution of earlier tasks may be used by later tasks. If there are several candidate positions in the FPGA where a task may be placed, the heuristic finds a position that allows the task to start as soon as possible. This location assignment policy is similar to the greedy heuristic proposed in [4].

Table IV (first row) shows the schedule length, reconfiguration resource utilization (lower is better), and reconfiguration power/energy consumption. The deadline is violated in this case. Fig. 7 shows the FPGA, processor, and bus schedules. The shaded blocks represent framewise reconfiguration. Reconfigurations caused by compulsory misses are not shown, as they occur only once in the beginning of the first hyperperiod. The numbers in brackets indicate the sequence in which the tasks are scheduled.

Scheduling Approach II: The scheduling sequence is determined dynamically by task priorities that consider both real-time constraints and frame-by-frame reconfiguration overhead information (see Section IV-B for additional details).

When deciding the location in the FPGA at which a task will be executed, the global reconfiguration information for all the tasks assigned to the FPGA and the FPGA's current configuration are considered. Table IV (second row) and Fig. 8 indicate the schedule quality for this approach.

From the above example, we find, not surprisingly, that different FPGA scheduling policies may dramatically influence the schedule quality, i.e., the satisfaction of real-time constraints, reconfiguration resource utilization, and reconfiguration energy consumption. First, since reconfiguration itself consumes a significant amount of energy, minimizing reconfiguration overhead is important for reducing system energy consumption. Second, solutions that cannot satisfy real-time constraints necessitate faster (and generally more expensive) PEs. This increases system price. A good scheduling approach indirectly reduces system price and average power consumption.

B. 2-D FPGA Scheduling Algorithm

In this section, we describe our 2-D scheduling algorithm for dynamically reconfigurable FPGAs. Scheduling sequence and location assignment are the most important problems solved during scheduling.

Scheduling Sequence: As in Approach I in Example 1, static slack-based priorities are commonly used to order tasks for scheduling on processors. The intuitive idea behind this approach is that a task with more slack can tolerate some delay and should yield to another task with less slack. This approach works well for sequential resources. However, it is not suitable for FPGAs, which can execute multiple tasks concurrently. In the static slack-based priority approach, tasks along the critical path of one task graph might always be scheduled before tasks

TABLE IV
SCHEDULING RESULTS

Approach	Deadline	Schedule length (μs)	Reconfiguration utilization (%)	Avg. reconfig. power (mW)	Avg. reconfig. energy (μJ)
I	Violation	117	48	127	19.1
II	Satisfied	80	23	61	9.2

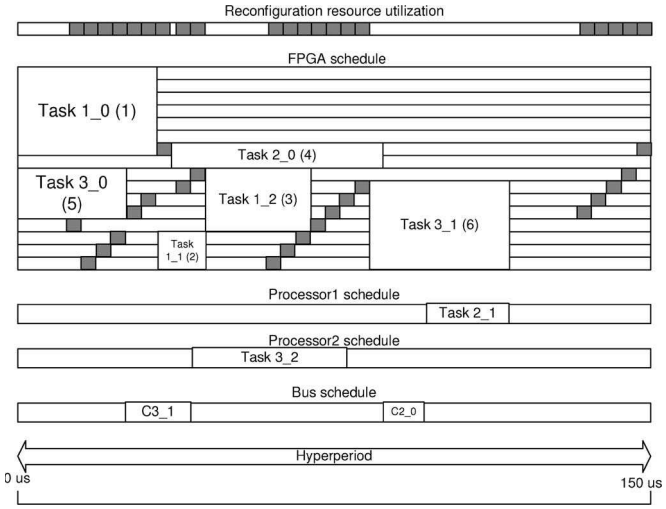


Fig. 7. Scheduling result for Approach I.

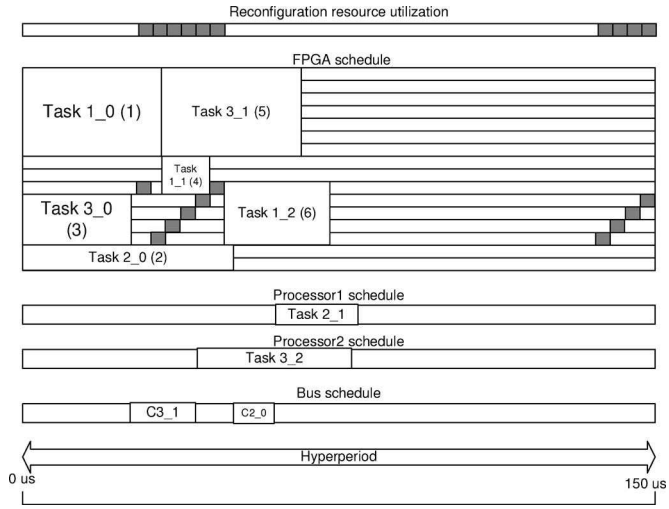


Fig. 8. Scheduling result for Approach II.

in other task graphs. This can be suboptimal for FPGAs. Our experimental results show that interleaving the execution of tasks from different graphs on FPGAs leads to better global schedules by allowing additional opportunities for amortizing reconfiguration overhead over multiple tasks.

Another difference between processors and FPGAs is that in FPGAs, reconfiguration degrades performance and increases energy consumption. Hence, to reduce the reconfiguration overhead, the ready tasks that require configuration patterns already residing in an FPGA should be preferred. This means that the reconfiguration overhead should also influence task priority. We propose a dynamic priority-based approach, which dynamically updates task i's priority, i.e., p_i, as follows:

$$p_i = -\text{LFT}_i + d_i + r_i - r_{ij} \tag{6}$$

where LFT_i is the latest possible finish time for task i, which is computed by conducting a backward topological sort of the task graph based on the deadline information. d_i is the worst-case execution time for task i on the PE to which it is assigned. r_i is the reconfiguration overhead for task i. r_{ij} is the intertask reconfiguration time between adjacent tasks i and j. r_{ij} equals 0 if these two tasks share the same configuration pattern. This approach considers both the real-time constraints and reconfiguration overhead. Moreover, tasks from different task graphs are treated fairly.

Location Assignment Policy: When a task is selected using the above approach, multiple candidate locations may exist in the FPGA. The location assignment policy for a task influences the scheduling results for other tasks, as well as those of the current task. Several factors need to be considered in this context.

- *Reconfiguration prefetch:* Each task needs to be loaded into the FPGA before execution. When a task's implementation is large, the reconfiguration overhead may be substantial even in dynamically reconfigurable FPGAs. Reconfiguration prefetch can be used to alleviate this problem. The system can try loading the task earlier and finish reconfiguration before the ready time of the task. This may allow the reconfiguration time for the task to be hidden.
- *Configuration pattern reuse:* When a new task needs to be loaded into an FPGA, its configuration patterns must be mapped into a set of contiguous frames. If subsets of the requisite configuration patterns already reside in the FPGA, loading them can be avoided. This helps reduce reconfiguration overhead.
- *Eviction candidate:* If an FPGA has insufficient free space for a new configuration pattern, resident configuration patterns must be evicted from the device. This problem is similar to the paging problem [65] and the weighted caching problem [66]. However, for our problem, all the frames assigned to a task need to be contiguous, further increasing the complexity of the problem. The frames that need to be reconfigured for the incoming task may contain configuration patterns from different tasks, each executing at a different relative frequency (the number of times the task executes in the system hyperperiod). When a configuration pattern with a higher relative frequency is evicted, this may result in additional reconfiguration overhead later in the hyperperiod. We define the eviction cost

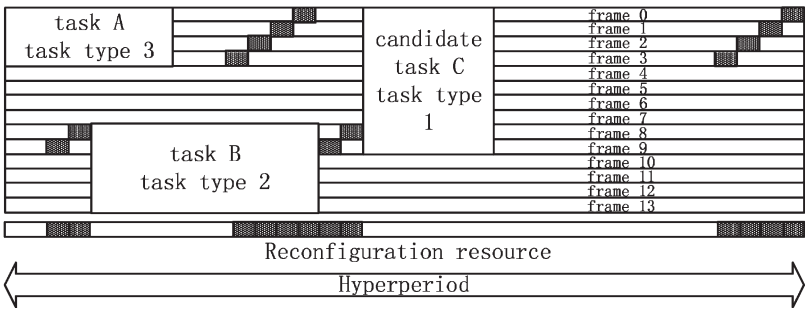


Fig. 9. Task scheduling example.

for a candidate position for this task to be a weighted sum of all the configuration patterns that need to be replaced, as follows:

$$\text{eviction\_cost} = \sum_{i=\text{start\_frame}}^{\text{end\_frame}} \text{recurrent\_freq}_{\text{frame}_i} \quad (7)$$

where $\text{recurrent\_freq}_{\text{frame}_i}$ is the recurrent frequency of the configuration pattern in frame_i. The eviction_cost is the weighted cost for this candidate position. The candidate positions with lower eviction_cost should be preferred.

- *Fitting policy:* The algorithm should try to avoid fragmentation of the FPGA configuration memory when choosing the candidate position from the FPGA.
- *Slack utilization:* Some of the possible candidate positions for a ready task may already have configuration patterns similar to the newly required ones. Using these positions lowers the eviction costs. However, tasks may not be able to start execution immediately if assigned to such candidate positions. A greedy policy may neglect such candidate positions, adversely impacting the schedule quality for other tasks because reconfiguration hardware is a sequential resource. Reconfiguration of one frame delays reconfiguration of others. Therefore, reconfiguration overhead minimization should have a high priority. Thus, it is sometimes necessary to make a suboptimal local scheduling decision for a task to allow a better global system schedule. The slack of a task indicates the extent to which an inferior scheduling result for it can be tolerated. Since the task may share slack with other tasks that have not yet been scheduled, slack should be conserved when possible. The portion of the slack allocated to a given task is its slack divided by the depth, i.e., the longest path, of the subtask graph (the root vertex of the subtask graph is the current task), as follows:

$$\text{tolerate\_start\_time}_j = \text{start\_time}_j + \frac{\text{slack}_{\text{task}_j}}{\text{depth}_{\text{sub\_graph}}} \quad (8)$$

- where start_time_j is the ready time of task_j, $\text{depth}_{\text{sub\_graph}}$ is the depth of the subtask graph in terms of the number of tasks, and $\text{slack}_{\text{task}_j}$ is the slack of task_j. $\text{Tolerate\_start\_time}_j$ is the latest delayed start time that task_j can tolerate.

Our FPGA location selection policy is based on the above analysis. The influence of reconfiguration overhead on the start time of each task is minimized. Candidate positions with lower weighted reconfiguration overhead and tolerable delay are always chosen. Reconfiguration data can be effectively shared among tasks with similar reconfiguration patterns. The reconfiguration overhead is, therefore, effectively reduced and sometimes hidden. This approach also minimizes reconfiguration energy consumption, which is a significant portion of the energy consumption in FPGAs.

Algorithm: The pseudocode for the 2-D scheduling algorithm is shown in Algorithm 1. First, root nodes from all the task graphs are put into the candidate pool (line 1). The priority of each task in the candidate pool is updated dynamically (line 3), and the task, i.e., task_i, with the highest priority is chosen (line 4). Since the parent tasks of task_i may be assigned to PEs other than task_i, the corresponding communication events need to be scheduled on the communication resource first (line 5). Then, task_i is scheduled on the candidate PE (line 6). Finally, scheduling task_i leads to other tasks becoming ready (line 7). The key part of the scheduling algorithm is schedule_task(task_i),

Algorithm 1 Scheduling_algorithm():
1: candidate_pool $\leftarrow$ root_nodes
2: **while** pending_tasks $\neq \emptyset$ **do**
3: priority_calculation(candidate_pool)
4: $\text{task}_i \leftarrow$ extract(candidate pool)
{task with the highest priority}
5: sched_input_communication(task_i)
6: schedule_task (task_i)
7: candidate_pool $\leftarrow$ introduce_ready_task(task_i)
8: **end while**

Consider task C in the partial FPGA schedule shown in Fig. 9. When this task is being loaded into the FPGA, the reconfiguration overhead may be introduced before and/or after the task, shown as shaded blocks.

Two issues need to be considered for the reconfiguration blocks introduced before task C. First, the time spans of the empty slots in the different frames among the possible candidate positions for task C may differ. Since the reconfiguration hardware is a sequential resource, reconfiguration of one frame will delay the reconfiguration of other frames and even the start time of the task. Second, the reconfiguration slots left unused between the reconfiguration events and task C cannot be used by tasks with different configuration patterns. We define

P_{frame_i}, i.e., the priority used to determine the reconfiguration sequence of frames, as follows:

$$P_{\text{frame}_i} = \begin{cases} -(r\_t_{\text{task}} - s\_t_{\text{frame}_i}) \\ -(\text{hyperperiod} - s\_t_{\text{frame}_i} + r\_t_{\text{task}}) \\ r\_t_{\text{task}} = \text{ready\_time}_{\text{task}} \text{ modulo hyperperiod} \\ s\_t_{\text{frame}_i} = \text{start\_time}_{\text{frame}_i} \text{ modulo hyperperiod} \end{cases} \tag{9}$$

where $r\_t_{\text{task}} \geq s\_t_{\text{frame}_i}$, $r\_t_{\text{task}} < s\_t_{\text{frame}_i}$, and $\text{start\_time}_{\text{frame}_i}$ is the start time of the empty time slot in frame_i. If the duration between the reconfiguration slot start time and the task ready time is short, reconfiguration of the corresponding frame needs to be scheduled first. Otherwise, reconfiguration may not be completed before the task ready time and the task may be delayed. The reconfiguration slots in each frame are scheduled before the ready task in order of decreasing priority. To hide the reconfiguration overhead whenever possible, a function called schedule_back() is used. This function looks backward for the first available reconfiguration slot from r_t_{task} to $s\_t_{\text{frame}_i}$ in the current frame. If the function returns false, this indicates that reconfiguration cannot start during the interval $[s\_t_{\text{frame}_i}, r\_t_{\text{task}}]$. In this case, another function, i.e., schedule_front(), is invoked. This function locates the first available reconfiguration slot in the current frame from r_t_{task} to the finish time of the empty time span. With this approach, reconfiguration events are scheduled as soon as possible before the task ready time and as closely as possible to the task to allow a good fit and, therefore, high slack utilization. In Fig. 9, before candidate task C, frames 8 and 9 are scheduled first, then frames 0–3.

We next discuss the issues involved in scheduling reconfiguration slots after the task. To leave enough flexibility for future tasks, the reconfiguration slots need to be placed as close to the next task as possible. In addition, a scheduling order priority must be determined for all the necessary frames, in accordance with their relationships, as follows:

$$P_{\text{frame}_i} = \begin{cases} -(f\_t_{\text{frame}_i} - r\_\text{task}) \\ -(\text{hyperperiod} - r\_t_{\text{task}} + f\_t_{\text{frame}_i}) \\ r\_t_{\text{task}} = \text{ready\_time}_{\text{task}} \text{ modulo hyperperiod} \\ f\_t_{\text{frame}_i} = \text{finish\_time}_{\text{frame}_i} \text{ modulo hyperperiod} \end{cases} \tag{10}$$

where $f\_t_{\text{frame}_i} \geq r\_t_{\text{task}}$, $f\_t_{\text{frame}_i} < r\_t_{\text{task}}$, and $\text{finish\_time}_{\text{frame}_i}$ is the finish time of the empty time span in frame_i. The schedule_back() function is called for each frame in a nonincreasing priority order. It chooses the first available reconfiguration slot from $f\_t_{\text{frame}_i}$ to r_t_{task} in the current frame. With this approach, in Fig. 9, in frames 0–3, the reconfiguration slots after task C are scheduled close to task A (note that tasks repeat after the hyperperiod). In frames 8 and 9, the reconfiguration slots are scheduled close to task B.

The schedule_task(task_i) function contains two steps. First, candidate_position_sort(task_i) calculates the priority of each candidate position. Its pseudocode is shown in Algorithm 2. In lines 2–6, the algorithm calculates the priorities of the frames in each candidate position. Then, for each candidate position, it schedules reconfiguration slots before the task based on the frame priorities (lines 7–10). From all the frames in this candidate position, it chooses the latest reconfiguration finish time as the actual task ready time for this position. Then, it uses the location assignment policy described earlier in this section to calculate the priority of each candidate position (line 11). Second, function schedule_task_p(task_i) is invoked to schedule the task. Its pseudocode is shown in Algorithm 3. The candidate position with the highest priority is chosen from the candidate_position_pool (line 2). The reconfiguration slots before the task are first scheduled (line 3), followed by the reconfiguration slots after the task (line 4). Finally, the task itself is inserted into the schedule (line 5). If any of these three steps fails, the frame at which the failure occurs is chosen. Starting from this time slot, the next time slot is located, and a new priority is calculated based on this frame (line 7). The candidate position is inserted into the priority queue at the appropriate location (line 8), and a new candidate position is chosen to attempt scheduling of the task (line 9).

Algorithm 2 Candidate_position_sort(task_i)
1: **for** $i \in \{0 \cdots \text{num\_candidate\_positions} - 1\}$ **do**
2: **for** $j \in \{\text{position\_start}_i \cdots \text{position\_finish}_i\}$ **do**
3: $\text{slot}_j \leftarrow$ candidate_time_slot find()
4: $\text{slot\_priority}_i \leftarrow$ slot_priority_calculation(slot_j)
5: $\text{slot\_priority\_}pl_i$.insert($\text{slot\_priority}_i$)
6: **end for**
7: **for** $j \in \{\text{slot\_priority\_}pl_i.\text{begin} \cdots$ $\text{slot\_priority\_}pl_i.\text{end} - 1\}$ **do**
8: **if** $\text{reconfig\_frame}_i = \text{false}$ **then**
9: schedule_reconfig()
10: **end if**
11: update_position_priority($\text{candidate\_position}_i$)
12: **end for**
13: **end for**

Algorithm 3 Schedule_task_p(task_i)
1: **while** candidate_position_pool $\neq \emptyset$ **do**
2: candidate_position $\leftarrow$ extract(candidate_position_pool)
3: stage1_flag $\leftarrow$ schedule_reconfig_before_task(task_i)
4: stage2_flag $\leftarrow$ schedule_reconfig_after_task(task_i)
5: stage3_flag $\leftarrow$ schedule_task_exec(task_i)
6: **if** stage1_flag or stage2_flag or stage3_flag **then**
7: calculate_priority(candidate_position.next_slot())
8: candidate_position_pool.insert(candidate_position)
9: next_candidate_position_chosen()
10: **end if**
11: **end while**

Complexity Analysis: For the FPGA scheduling algorithm, assuming an $O(n \log n)$ sorting algorithm, the time complexity is $O(MKn^2 \log n)$, where M is the FPGA size, K is the average number of frames required by a task, and n is the number of tasks. However, on average, for the specifications on which we ran it, the algorithm behaves like an $O(MKn \log n)$ algorithm because it usually finds a feasible reconfiguration slot early in the candidate pool. Note that our implementation uses

quicksort instead of an $O(n \log n)$ sorting algorithm because its average-case performance is superior.

Scheduling Algorithms for Other Resources: FPGA scheduling is compatible with scheduling for processors and communication resources. We use the same approach to schedule tasks and communication events on different processors and communication resources by setting reconfiguration times to 0 and using only one frame for each device. In other words, we revert to a 1-D scheduler for processors and communication resources. Intertask data dependences are used to coordinate the schedules of tasks and communication events in different computation and communication resources.

V. Experimental Results

In this section, we present experimental results for our hardware–software cosynthesis system. We first compare the proposed multidimensional scheduler for partially dynamically reconfigurable hardware with existing techniques that do not support partial reconfiguration (see Section V-A). Next, we compare the underlying distributed heterogeneous distributed system synthesis algorithm with related work (see Section V-B). Finally, in Section V-C, we present the results of using the proposed algorithm to synthesize hardware–software systems containing partially dynamically reconfigurable logic.

A. Scheduling Algorithm for Partially Reconfigurable Hardware

The system is implemented in C++. The resource library consists of various system resources available from the industry and academia. We use processors, memory blocks, and communication resources provided in previous work [63] as well as publicly available benchmarks [64]. The parameters of our dynamically reconfigurable FPGA model are based on Xilinx Virtex-E FPGAs [19]. We use the SelectMAP reconfiguration mode, i.e., a bidirectional parallel interface.

The writing reconfiguration time for task i, i.e., r_i, is estimated based on the number of configuration frames (N_frames_i) used by each task, the frame size ($M_$bits), the width of the configuration interface ($K_$bits), which is 8 bits, and the configuration frequency f, as follows:

$$r_i = N\_\text{frames}_i \times M\_\text{bits}/(K\_\text{bits} \times f) + C \qquad (11)$$

where C is a constant overhead due to initial synchronization and setting of the address and other configuration registers.

The first ten sets of example task graphs, which are input to the cosynthesis system, were generated by TGFF [63]. The next two are based on digital signal processing examples from the literature [67], [68]. An additional three benchmarks, for telecommunication, automotive, and networking applications, were obtained from E3S [64]. These last three were derived from characterizations of a wide range of typical embedded system tasks running on a large set of embedded processors and microcontrollers available from the Embedded Microprocessor Benchmarking Consortium (EEMBC) [69]. The benchmarks provide performance information. Therefore, we know the execution time of each task on each processor. We derived power consumption and other information from data sheets and conversations with processor manufacturers. Therefore, for the current set of benchmarks, all tasks have the same power consumption on a given processor. Note that our synthesis system accepts performance and power consumption information for each task type running on each processor type. Therefore, task-specific power numbers may be used when EEMBC begins to report them. For each task, FPGA power consumption is linearly scaled from the power consumption in processors. All the experiments were performed on a Pentium-III 667-MHz PC with 512 MB of memory running the Linux operating system.

We first demonstrate the performance of our FPGA scheduling algorithm. We compare the results of scheduling for Approach I (Section IV-A), based on static slack-based priority, configuration prefetch, and preconfiguration utilization [4], with our Approach II. The results are shown in Table V, which includes schedule length, average reconfiguration power consumption, and CPU time for the 15 examples described above. Note that reconfiguration energy consumption is exactly proportional to reconfiguration power consumption, i.e., the relative improvements are equivalent. Compared with Approach I, the improvements in schedule length and average reconfiguration power are shown in columns 4 and 7, respectively, and also in Fig. 10, in which bars represent schedule length and lines represent average reconfiguration power.

In contrast to Approach I, our algorithm always meets real-time performance constraints. When using Approach I, only the solutions for Examples 3, 5, and 9 meet real-time performance constraints. The average reduction in schedule length is 34.3%, and the average reduction in reconfiguration energy consumption is 40.4%. Recall that reconfiguration energy consumption is frequently as high as task energy consumption. Hence, it is very important to reduce it. Reducing schedule length helps the cosynthesis system choose lower cost (and potentially slower) PEs without violating real-time constraints, thus reducing the system price. Although our approach improves average reconfiguration power consumption in 13 examples, it increases average power consumption for Example 9. In this example, there are tight FPGA resource constraints. Therefore, little flexibility remains for our scheduling algorithm to search for a globally optimal solution. Since our approach may not choose a locally optima solution for each task, it may at times get a worse result than the greedier Approach I. In addition to producing better results than Approach I for most problems, our algorithm needs less runtime: 28.0% less on average. Our algorithm predicts the needs of future tasks and makes it easier to schedule them. Approach I is greedy and makes locally optimal choices. Therefore, it needs more time to schedule tasks encountered later.

B. Heterogeneous Distributed System Synthesis

To compare the allocation and assignment optimization algorithm used in SLOPES with other work, we used it for hardware–software cosynthesis of systems that do not contain partially dynamically reconfigurable hardware.

TABLE V
FPGA SCHEDULING RESULTS

Example	Schedule length (μs)			Avg. reconf. power (mW)			CPU time (seconds)		
	I	II	Imp.	I	II	Imp.	I	II	Imp.
1	4815	1625	66.3%	101.4	12.0	88.2%	3.2	2.2	31.3%
2	12530	5302	57.7%	186.7	88.1	52.8%	0.7	0.3	57.1%
3	8353	5488	34.3%	114.8	81.3	29.2%	7.5	3.6	52.0%
4	5992	2392	60.1%	88.4	37.3	57.8%	3.2	1.4	56.3%
5	9139	6903	24.5%	120.2	94.0	21.8%	5.9	4.3	27.1%
6	3282	2852	13.1%	223.3	193.3	13.4%	1.2	1.1	8.3%
7	2066	1351	34.6%	33.1	19.9	39.9%	2.4	1.5	33.3%
8	4270	1600	62.5%	99.3	33.1	66.7%	0.7	0.5	28.6%
9	4600	4717	-2.5%	67.9	74.7	-10.0%	3.8	3.2	15.8%
10	6444	2588	59.8%	110.3	0	100.0%	0.5	0.3	40.0%
11	4403	4114	6.5%	216.9	204.8	5.6%	2.9	2.5	13.8%
12	1392	995	28.5%	22.5	16.6	26.2%	0.3	0.3	0
13	28190	15796	44.0%	206.6	126.0	39.0%	1.1	0.7	36.4%
14	475	357	24.8%	6.8	1.7	75.0%	0.5	0.4	20.0%
15	17442	17442	0%	232.3	232.3	0%	0.5	0.5	0

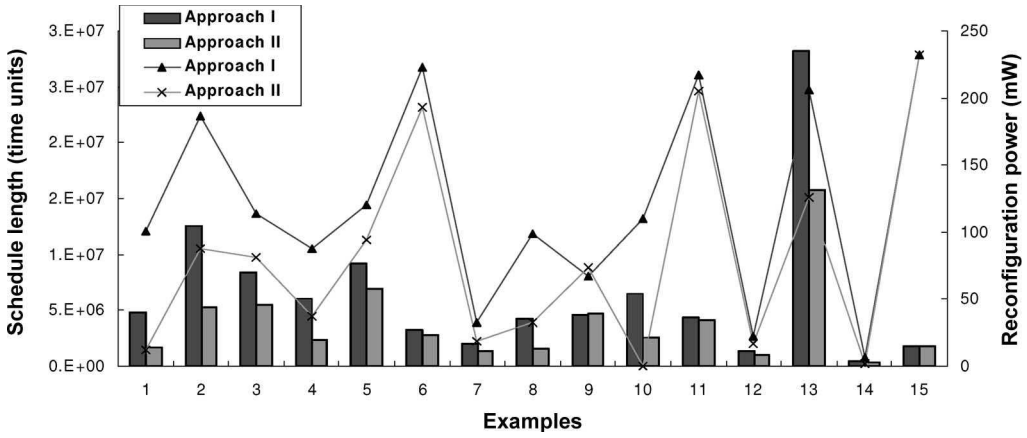


Fig. 10. FPGA scheduling results.

Table VI shows the results of running the allocation and assignment optimization algorithm on all of the P&P SOS examples [28]. The first column shows the names of the examples and the performance constraints (i.e., deadlines). The second column shows the prices of the solutions found by our optimization algorithm. The other columns show the prices of the solutions found by algorithms developed by other researchers. The third column is for SOS, the P&P MILP algorithm [28]. This algorithm has the advantage of guaranteeing optimality. However, its runtime increases dramatically with increasing problem complexity. The fourth column is for COSYN, a constructive algorithm [21]. The fifth column is for Oh and Ha's heuristic [27]. The P&P 2 $\langle 5 \rangle$ and P&P 2 $\langle 7 \rangle$ entries are explained in the next few paragraphs.

The P&P examples contained no soft deadlines or power information. Therefore, we ran our optimization algorithm in single-objective price optimization mode. We used the same optimization parameters for each of these examples and for those in the next section. A 1.4 GHz AMD Athlon Thunderbird CPU was used to solve these problems. Each example took between 12 and 35 min of CPU time. Note that it is possible for our optimization algorithm to produce good solutions to the simpler P&P examples in significantly less than 12 min of CPU time. However, we wanted to use the same optimization parameters for all of the P&P examples, as well as all of the Hou and Wolf's examples (considered later). Therefore, we selected a solution pool size and halting conditions sufficient for more complicated problems, i.e., we granted the optimization algorithm more CPU time than was necessary for simple problems so that it would have good performance on complicated problems. The dependence of our optimization algorithm's CPU time requirements upon problem complexity stands in contrast with

TABLE VI
PRICES FOR THE PARAKASH AND PARKER EXAMPLES

Example	Optimization algorithm within SLOPES	SOS	COSYN	Oh & Ha's algorithm
P&P 1 ⟨2.5⟩	14	14	n.a.	n.a.
P&P 1 ⟨3⟩	13	13	n.a.	n.a.
P&P 1 ⟨4⟩	7	7	n.a.	7
P&P 1 ⟨7⟩	5	5	5	5
P&P 2 ⟨5⟩	14 (15) †	15	n.a.	n.a.
P&P 2 ⟨6⟩	12	12	n.a.	n.a.
P&P 2 ⟨7⟩	7 (8) †	8	n.a.	n.a.
P&P 2 ⟨8⟩	7	7	n.a.	7
P&P 2 ⟨15⟩	5	5	5	5
P&P 3 ⟨6⟩	10	10	10	n.a.
P&P 3 ⟨7⟩	6	6	n.a.	n.a.
P&P 3 ⟨15⟩	5	5	5	n.a.

n.a.: not available

the requirements of SOS. Although SOS took only 11 s of CPU time, on a Solbourne Series5e/900 (similar to a SPARC 4/490), for a simple problem, P&P 1 $\langle 2.5 \rangle$, its runtime increased dramatically with increased problem complexity; it took 106.7 h of CPU time for P&P 2 $\langle 15 \rangle$. There is no particular problem with taking a large amount of CPU time to solve a synthesis problem well. However, dramatic increases in optimization time with increasing problem complexity imply that an algorithm may not halt in an acceptable amount of time for large problems.

The P&P behavioral specifications are somewhat unconventional. They contain tasks with precomputation and postcomputation. We implemented an extension to conventional task graphs to precisely model this. Our model does vary from that used by P&P in one way: Our point-to-point communication links are bidirectional, and theirs are directed, i.e., they allow communication to occur over a point-to-point link in only one direction during the life of an embedded system. In our model, communication via a point-to-point link can occur in either direction, although only one communication event can be carried by the point-to-point link at a time. This results in some apparently unusual results for the P&P 2 $\langle 5 \rangle$ and P&P 2 $\langle 7 \rangle$ examples. Our slightly different model for point-to-point communication links allows our algorithm to get lower prices than SOS in a few instances. Although a price comparison is still legitimate, it requires some explanation. We will now describe the impact of this difference on the solutions to each of the P&P examples.

None of the solutions produced by SOS for any of the deadlines associated with the P&P 1 examples contain a pair of point-to-point communication links that connect the same pair of PEs and have different directions, i.e., *back-to-back links*. Therefore, a bidirectional communication model will not allow back-to-back links to be merged, thereby reducing price, in any of these solutions. For every P&P 1 example, our optimization algorithm arrived at a solution with the same price as SOS. The P&P 3 example resource database does not contain directed point-to-point communication links. As a result, the difference in communication models has no impact on the results for this example. Our optimization algorithm arrived at a solution with the same price as SOS for every P&P 3 example.

The differing communication models had an impact on some of the solutions to the P&P 2 examples. For the problems with deadlines of five and seven, the optimal solutions produced by SOS contain exactly one pair of back-to-back links. A bidirectional communication model might allow one of these links to be removed, thereby reducing the solution price by, at most, one unit. Our algorithm arrived at such solutions. If one were to reinsert that link, the solution prices would be equal to those of SOS, which we have shown in parentheses in Table VI. For the problems with deadlines six, eight, and 15, SOS's solutions contain no back-to-back links. Therefore, a bidirectional communication model would not improve the solutions.

Our optimization algorithm produced a solution with the same price as SOS's optimal solution for every example in which a bidirectional communication model would not allow a pair of back-to-back links to be merged in the optimal solution produced by SOS. In cases for which our communication model could potentially allow point-to-point communication links to be merged, thereby reducing price, our algorithm arrived at a solution that had a price exactly equal to that of SOS, minus the savings that might result from merging of back-to-back links. In practice, our optimization algorithm finds solutions that are substantially equivalent to those produced by SOS, an optimal algorithm, although our optimization algorithm has CPU time requirements that do not increase rapidly with increasing problem complexity.

Table VII compares the results produced by running our optimization algorithm on Hou and Wolf's examples [71] with those produced by other hardware–software cosynthesis algorithms. The first column states whether or not the example in question is clustered. Task clustering is the process of using a prepass to collapse multiple tasks into a cluster of tasks. This cluster is treated like a single task during assignment, i.e., all the tasks in a cluster are executed on the same PE. Clustering reduces the complexity of the cosynthesis problem by decreasing the number of tasks that must be assigned. Clustered graphs have a similar structure to unclustered graphs but contain fewer tasks. The second column contains the names of Hou and Wolf's examples. The third column shows the prices of the solutions produced by our optimization algorithm. The other columns show the prices of the solutions found by algorithms developed by other researchers. The fourth column is for COSYN [21]. The fifth column is for Yen's iterative improvement algorithm [33]. The sixth column is for Oh and Ha's heuristic [27].

The above examples contained no soft deadlines or power information. Therefore, we ran our optimization algorithm in single-objective price optimization mode. We did not contract the periods and deadlines of these examples to reduce the hyperperiod: These are precisely Hou and Wolf's examples. We used the same optimization parameters for each of these examples and for the examples in the previous section. These examples were run on a 1.4-GHz AMD Athlon Thunderbird CPU. Each example took approximately 10 min of CPU time,

TABLE VII
OPTIMIZATION FOR THE HOU AND WOLF'S EXAMPLES

Clustering	Example	Optimization algorithm within SLOPES	COSYN	Yen's algorithm	Oh & Ha's algorithm
Unclustered	H&W 1&2	140	170	170	170
	H&W 1&3	170	170	240	170
	H&W 3&4	140	n.a.	210	170
Clustered	H&W 1&2	140	n.a.	170	170
	H&W 1&3	170	n.a	170	n.a.
	H&W 3&4	170	n.a.	170	n.a.

n.a.: not available

with the exception of H&W 1&3 unclustered, which took 74 min of CPU time. For all of Hou and Wolf's problems, our optimization algorithm arrived at solutions with prices that are equal to or lower than those produced by past work.

It is interesting to note the implications of these results for clustering research. Task clustering converts a task graph into another task graph with fewer nodes by grouping some nodes together and treating them as a single node. This has the potential to improve the solutions produced by a cosynthesis algorithm by eliminating unpromising areas from the search space. For example, if all of a problem's promising solutions assign two tasks to the same PE and schedule them concurrently, converting them into a single task will concentrate a search on the most promising areas of the solution space. However, although task clustering can simplify a hardware–software cosynthesis problem and eliminate unpromising potential solutions from the search space, it can also eliminate promising solutions from the search space. Note that for the H&W 3&4 example, our optimization algorithm was able to find a superior solution to the unclustered version of the problem. For the unclustered version of this problem, our optimization algorithm found a solution with a price of 140. However, for the clustered version, such a solution is not possible. In this example, clustering forced tasks that would ideally be assigned to different PEs to be assigned to the same PE. It is important for a clustering algorithm not to eliminate the possibility of finding a good solution in its attempts to simplify a problem. It is our opinion that the best place to carry out such pruning and simplification is within a synthesis algorithm, where additional information is available about allocation and assignment, not as a prepass.

In summary, when compared with the four results of the optimal SOS MILP solver [28], for each problem, the optimization infrastructure used in SLOPES also produces optimal solutions. For large problem instances, the CPU time required was 1/100 000 of that required by the MILP solver. For very small and tightly constrained problem instances, the MILP solver was faster than the optimizer used in SLOPES. However, for these problem instances, CPU time requirements were also low for SLOPES (approximately 15 min). The difference in CPU times increased dramatically with increasing problem size. Note that, unlike SOS, SLOPES makes no guarantee of finding optimal solutions for all problem instances. When compared with two iterative improvement algorithms [27], [59] as well as a constructive algorithm [21], the optimization infrastructure used in SLOPES found results that were the same, in three cases, and better, in three cases, than the best results found by any of these other heterogeneous distributed system synthesis algorithms. The required CPU time was similar to that of the constructive algorithm and ranged from equal to 1/1000 that of the iterative improvement algorithm, with larger performance improvements for larger problem instances.

C. Synthesis of Partially Reconfigurable Heterogeneous Distributed Systems

The results for our hardware–software cosynthesis system are shown in Table VIII. In this table, columns 2 and 3, respectively, show the corresponding system price and average power consumption of all the nondominated solutions. The last column shows the CPU time required for cosynthesis. The system price is calculated by summing the prices of all the processors, FPGAs, communication resources, and memory in the synthesized distributed embedded system. The average system power consumption is calculated by summing all the execution, reconfiguration, communication, and idle energy consumption in the hyperperiod and dividing by the hyperperiod. Table VIII illustrates the ability of our cosynthesis system to explore the design space. Our multiobjective optimization approach allows a range of tradeoffs between system price and average power consumption. All real-time constraints are satisfied. The runtime indicates that large task graphs can be handled in a reasonable amount of time.

VI. CONCLUSION

This paper described SLOPES, which is a multiobjective hardware–software cosynthesis algorithm for real-time distributed embedded systems. A 2-D (space and time) multirate cyclic scheduling algorithm was proposed to tackle the scheduling problem in dynamically reconfigurable FPGAs. This algorithm not only minimizes schedule length (thus allowing cheaper PEs) but also significantly reduces reconfiguration energy. Reconfiguration delay and energy are the main bottlenecks in exploiting dynamic reconfiguration in modern FPGAs; SLOPES addresses both of these bottlenecks. The cosynthesis system optimizes both the price and average power consumption of distributed embedded systems containing partially dynamically reconfigurable FPGAs.

TABLE VIII
HARDWARE–SOFTWARE COSYNTHESIS RESULTS

Example	Price (dollar)	Avg. Power consumption (mW)	CPU time (minutes)
1	209	144.7	99.7
	389	66.1	
2	42	394.5	133.6
	212	253.6	
3	57	619.7	19.8
	153	305.5	
	173	271.1	
	198	121.9	
	525	108.4	
4	159	745.5	54.2
	174	626.9	
	209	503.6	
5	153	815.6	28.8
	385	699.8	
	420	489.4	
6	232	922.7	14.9
	367	829.6	
	394	557.5	
7	156	684.2	3.0
	353	462.9	
8	156	790.5	18.0
	204	345.6	
9	209	852.0	39.2
	238	345.8	
	250	265.3	
10	156	353.8	2.1
11	145	223.1	16.4
	228	202.9	
12	82	69.2	12.2
	252	57.4	
13	167.7	1474.5	11.9
	227.1	101.2	
14	143.1	1387.7	8.2
15	144.2	1270.0	0.5
	164.2	769.3	
	288.1	400.7	

REFERENCES

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

[2] B. Dave, "CRUSADE: Hardware/software co-synthesis of dynamically reconfigurable heterogeneous real-time distributed embedded systems," in *Proc. Des. Autom. and Test Eur. Conf.*, Mar. 1999, pp. 97–104.

[3] N. Shenoy, A. Choudhary, and P. Banerjee, "An algorithm for synthesis of large time-constrained heterogeneous adaptive systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 6, no. 2, pp. 207–225, Apr. 2001.

[4] B. Jeong, S. Yoo, S. Lee, and K. Choi, "Hardware–software cosynthesis for run-time incrementally reconfigurable FPGAs," in *Proc. Asia and South Pacific Des. Autom. Conf.*, Jan. 2000, pp. 169–174.

[5] Y. B. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, "Hardware–software co-design of embedded reconfigurable architectures," in *Proc. Des. Autom. Conf.*, Jun. 2000, pp. 507–512.

[6] J. Noguera and R. Badia, "A HW/SW partitioning algorithm for dynamically reconfigurable architectures," in *Proc. Des. Autom. and Test Eur. Conf.*, Mar. 2001, pp. 729–734.

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

[8] A. Dandalis, V. K. Prasanna, and J. D. P. Rolim, "A comparative study of performance of AES final candidates using FPGAs," in *Proc. Int. Workshop Cryptographic Hardware and Embedded Syst.*, Aug. 2000, pp. 125–140.

[9] J. Goodman and A. P. Chandrkasan, "An energy efficient reconfigurable public-key cryptography processor architecture," in *Proc. Int. Workshop Cryptographic Hardware and Embedded Syst.*, Aug. 2000, pp. 175–190.

[10] S. Choi, R. Scrofano, V. K. Prasanna, and J.-W. Jang, "Energy-efficient signal processing using FPGAs," in *Proc. Int Symp. Field Programmable Gate Arrays*, Feb. 2003, pp. 225–234.

[11] G. Stitt and F. Vahid, "The energy advantages of microprocessor platforms with on-chip configurable logic," in *Proc. Des. Test Comput.*, Dec. 2002, pp. 36–43.

[12] R. Lysecky and F. Vahid, "A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning," in *Proc. Des. Autom. and Test Eur. Conf.*, Mar. 2005, pp. 18–23.

[13] R. Scrofano, S. Choi, and V. K. Prasanna, "Energy efficiency of FPGAs and programmable processors for matrix multiplication," in *Proc. Int. Conf. Field Programmable Technol.*, Dec. 2002, pp. 422–425.

[14] O. Mencer, M. Morf, and M. J. Flynn, "Hardware software tri-design of encryption for mobile communication units," in *Proc. Int. Conf. Acoust., Speech and Signal Process.*, May 1998, pp. 3045–3048.

[15] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *Proc. Symp. FPGAs Custom Comput. Mach.*, Apr. 1997, pp. 12–21.

[16] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A time-multiplexed FPGA," in *Proc. Symp. FPGAs Custom Comput. Mach.*, Apr. 1997, pp. 22–28.

[17] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit," in *Proc. Int. Symp. Comput. Architecture*, Jun. 2000, pp. 225–232.

[18] T. Fujii *et al.*, "A dynamically reconfigurable logic engine with a multi-context/multi-mode unified-cell architecture," in *Proc. Int. Solid-State Circuits Conf.*, Feb. 1999, pp. 22–28.

[19] Xilinx Inc., *Virtex Series Data Sheet*. [Online]. Available: http://www.xilinx.com

[20] A. Bender, "Design of an optimal loosely coupled heterogeneous multiprocessor system," in *Proc. Eur. Des. and Test Conf.*, Mar. 1996, pp. 275–281.

[21] B. P. Dave, G. Lakshminarayana, and N. K. Jha, "COSYN: Hardware–software co-synthesis of heterogeneous distributed embedded systems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 7, no. 1, pp. 92–104, Mar. 1999.

[22] B. P. Dave and N. K. Jha, "COHRA: Hardware–software cosynthesis of hierarchical heterogeneous distributed embedded systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 17, no. 10, pp. 900–919, Oct. 1998.

[23] P.-A. Hsiung, "CMAPS: A cosynthesis methodology for application-oriented parallel systems," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 5, no. 1, pp. 51–81, Jan. 2000.

[24] I. Karkowski and H. Corporaal, "Design space exploration algorithm for heterogeneous multi-processor embedded system design," in *Proc. Des. Autom. Conf.*, Jun. 1998, pp. 82–87.

[25] K. Kuchcinski, "Embedded system synthesis by timing constraints solving," in *Proc. Int. Symp. Syst. Synthesis*, Sep. 1997, pp. 50–57.

[26] C. Lee, M. Potkonjak, and W. Wolf, "Synthesis of hard real-time application specific systems," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 4, no. 4, pp. 215–242, 1999.

[27] H. Oh and S. Ha, "Hardware–software cosynthesis technique based on heterogeneous multiprocessor scheduling," in *Proc. Int. Workshop Hardware/Software Co-Design*, May 1999, pp. 183–1878.

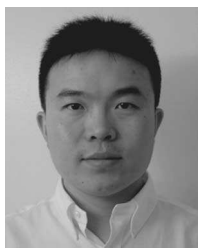
[28] S. Prakash and A. Parker, "SOS: Synthesis of application-specific heterogeneous multiprocessor systems," *J. Parallel Distrib. Comput.*, vol. 16, no. 4, pp. 338–351, Dec. 1992.

[29] M. Schwiegershausen and P. Pirsch, "Formal approach for the optimization of heterogeneous multiprocessors for complex image processing schemes," in *Proc. Eur. Des. Autom. Conf.*, Sep. 1995, pp. 8–13.

[30] S. Srinivasan and N. K. Jha, "Hardware–software co-synthesis of fault-tolerant real-time distributed embedded systems," in *Proc. Eur. Des. Autom. Conf.*, Sep. 1995, pp. 334–339.

[31] J. Teich, T. Blickle, and L. Thiele, "An evolutionary approach to system-level synthesis," in *Proc. Int. Workshop Hardware/Software Co-Design*, Mar. 1997, pp. 167–171.

[32] W. H. Wolf, "An architectural co-synthesis algorithm for distributed, embedded computing systems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 5, no. 2, pp. 218–229, Jun. 1997.
[33] T.-Y. Yen and W. H. Wolf, "Communication synthesis for distributed embedded systems," in *Proc. Int. Conf. Comput.-Aided Des.*, Nov. 1995, pp. 288–294.
[34] Z. Li, K. Compton, and S. Hauck, "Configuration caching techniques for FPGA," in *Proc. Symp. FPGAs Custom Comput. Mach.*, Apr. 2000, pp. 22–36.
[35] X. Tang, M. Aalsma, and R. Jou, "A compiler directed approach to hiding configuration latency in Chameleon processors," in *Proc. Int. Conf. Field-Programmable Logic and Appl.*, Aug. 2000, pp. 29–38.
[36] M. Kaul, R. Vemuri, S. Govindarajan, and I. Ouaiss, "An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications," in *Proc. Des. Autom. Conf.*, Jun. 1999, pp. 616–622.
[37] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-Completeness*. New York: Freeman, 1979.
[38] R. P. Dick and N. K. Jha, "MOGAC: A multiobjective genetic algorithm for hardware–software co-synthesis of distributed embedded systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 17, no. 10, pp. 920–935, Oct. 1998.
[39] E. Zitzler, "Evolutionary algorithms for multiobjective optimization: Methods and applications," Ph.D. dissertation, Swiss Federal Inst. Technol. Zurich, Zurich, Switzerland, Nov. 1999.
[40] R. P. Dick, "Multiobjective synthesis of low-power real-time distributed embedded systems," Ph.D. dissertation, Dept. Elect. Eng., Princeton Univ., Princeton, NJ, Jul. 2002.
[41] E. L. Lawler and C. U. Martel, "Scheduling periodically occurring tasks on multiple processors," *Inf. Process. Lett.*, vol. 12, no. 1, pp. 9–12, Feb. 1981.
[42] S. Malik, M. Martonosi, and Y.-T. S. Li, "Static timing analysis of embedded software," in *Proc. Des. Autom. Conf.*, Jun. 1997, pp. 147–152.
[43] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step toward software power minimization," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 2, no. 4, pp. 437–445, Dec. 1994.
[44] S. Gupta and F. Najm, "Power modeling for high-level power estimation," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 8, no. 1, pp. 18–29, Feb. 2000.
[45] Atmel Inc., *AT94K05/10/40AL Complete*. [Online]. Available: http://www.atmel.com
[46] P. Technologies, *XPP64-A1 Reconfigurable Processor*. [Online]. Available: http://www.packcorp.com
[47] L. Shang, A. S. Kaviani, and K. Bathala, "Dynamic power consumption in Virtex-II FPGA," in *Proc. Int. Symp. Field Programmable Gate Arrays*, Feb. 2002, pp. 157–164.
[48] Xilinx Inc., *Xilinx Power Tools*. [Online]. Available: http://www.xilinx.com
[49] K. K. W. Poon, A. Yan, and S. J. E. Wilton, "A flexible power model for FPGAs," in *Proc. Int. Conf. Field-Programmable Logic and Appl.*, Sep. 2002, pp. 312–321.
[50] L. Shang and N. K. Jha, "High-level power modeling of CPLDs and FPGAs," in *Proc. Int. Conf. Comput. Des.*, Sep. 2001, pp. 46–51.
[51] R. P. Dick, G. Lakshminarayana, A. Raghunathan, and N. K. Jha, "Analysis of power dissipation in real-time operating systems," *IEEE Trans. Comput.-Aided Design Integrated Circuits Syst.*, vol. 22, no. 5, pp. 615–627, May 2003.
[52] S. W. Mahfoud and D. E. Goldberg, "Parallel recombinative simulated annealing: A genetic algorithm," *Parallel Comput.*, vol. 21, no. 1, pp. 1–28, Jan. 1995.
[53] K. S. Chatha and R. Vemuri, "An iterative algorithm for hardware–software partitioning, hardware design space exploration and scheduling," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 3/4, pp. 281–293, Aug. 2000.
[54] J. D'Ambrosio and X. Hu, "Configuration-level hardware/software partitioning for real-time systems," in *Proc. Int. Workshop Hardware/Software Co-Design*, Aug. 1994, pp. 34–41.
[55] P. V. Knudsen and J. Madsen, "PACE: A dynamic programming algorithm for hardware/software partitioning," in *Proc. Int. Workshop Hardware/Software Co-Design*, Mar. 1996, pp. 82–95.
[56] R. K. Gupta and G. De Micheli, "Hardware–software cosynthesis for digital systems," *IEEE Des. Test Comput.*, vol. 10, no. 3, pp. 29–41, Sep. 1993.
[57] T. Benner and R. Ernst, "An approach to mixed systems co-synthesis," in *Proc. Int. Workshop Hardware/Software Co-Design*, Mar. 1997, pp. 9–14.
[58] D. Saha, R. S. Mitra, and A. Basu, "Hardware software partitioning using genetic algorithm," in *Proc. Int. Conf. VLSI Des.*, Oct. 1998, pp. 155–159.
[59] T.-Y. Yen, "Hardware–software co-synthesis of distributed embedded systems," Ph.D. dissertation, Dept. Elect. Eng., Princeton Univ., Princeton, NJ, Jun. 1996.
[60] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989.
[61] E. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines*. Chichester, U.K.: Wiley, 1989.
[62] C. M. Fonseca and P. J. Fleming, "Multiobjective genetic algorithms made easy: Selection, sharing and mating restrictions," in *Proc. Genetic Algorithms Eng. Syst.: Innovations and Appl.*, Sep. 1995, pp. 45–52.
[63] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: Task graphs for free," in *Proc. Int. Workshop. Hardware/Software Co-Design*, Mar. 1998, pp. 97–101.
[64] *E3S: The Embedded System Synthesis Senchmarks Suite*. E3S. [Online]. Available: http://www.eecs.northwestern.edu/.dickrp
[65] L. A. Belady, "A study of replacement algorithms for virtual storage computers," *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966.
[66] N. Young, "The k-server dual and loose competitiveness for paging," *Algorithmica*, vol. 11, no. 6, pp. 525–541, Jun. 1994.
[67] M. Woodside and G. G. Monforton, "Fast allocation of processes in distributed and parallel systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 2, pp. 164–174, Feb. 1993.
[68] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 5, pp. 506–521, May 1996.
[69] Embedded Microprocessor Benchmark Consortium. [Online]. Available: http://www.eembc.org
[70] P. Merino, M. Jacome, and J. C. Lopez, "A methodology for task based partitioning and scheduling of dynamically reconfigurable systems," in *Proc. Symp. FPGAs for Custom Computing Machines*, Apr. 1998, pp. 324–325.
[71] J. Hou and W. Wolf, "Process partitioning for distributed embedded systems," in *Proc. Int. Wkshp. Hardware/Software Co-design*, Mar. 1996, pp. 70–76.



Li Shang (S'99–M'04) received the B.E. and M.E. degrees from Tsinghua University, Beijing, China, and the Ph.D. degree from Princeton University, Princeton, NJ.

He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, Queen's University, Kingston, ON, Canada. He has published in the areas of computer architecture, behavioral synthesis, thermal/power modeling and optimization, reconfigurable computing, and mobile computing.

Prof. Shang was the recipient of the Best Paper Award at PDCS'02. He is the Walter F. Light Scholar of Queen's University.

Robert P. Dick (S'95–M'00) received the B.S. degree from Clarkson University, Potsdam, NY, and the Ph.D. degree from Princeton University, Princeton, NJ.

He was a Visiting Researcher with NEC Laboratories America and a Visiting Professor with the Department of Electronic Engineering, Tsinghua University, Beijing, China. He is currently an Assistant Professor with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL. He has published in the areas of embedded system synthesis, mobile *ad hoc* network protocols, reliability, behavioral synthesis, data compression, embedded operating systems, and thermal analysis of integrated circuits.

Prof. Dick won his department's Best Teacher of the Year Award in 2004.

Niraj K. Jha (S'85–M'85–SM'93–F'98) received the B.Tech. degree in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur, India, in 1981, the M.S. degree in electrical engineering from the State University of New York at Stony Brook in 1982, and the Ph.D. degree in electrical engineering from the University of Illinois at Urbana–Champaign in 1985.

He is a Professor of electrical engineering with Princeton University, Princeton, NJ. He has coauthored three books entitled *Testing and Reliable Design of CMOS Circuits* (Kluwer, 1990), *High-Level Power Analysis and Optimization* (Kluwer, 1998), and *Testing of Digital Systems* (Cambridge Univ. Press, 2003). He has also authored four book chapters and has authored or coauthored more than 280 technical papers. He is the holder of 11 U.S. patents. His research interests include nanotechnology, power/thermal analysis and optimization, computer-aided design of integrated circuits and systems, digital system testing, and embedded system security.

Prof. Jha is a Fellow of the Association for Computing Machinery. He currently serves as an Associate Editor of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—PART II: ANALOG AND DIGITAL SIGNAL PROCESSING, IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, and *Journal of Low Power Electronics*. He has served as an Editor of the *Journal of Electronic Testing: Theory and Applications* (JETTA) and as the Program Chairman of the 1992 Workshop on Fault-Tolerant Parallel and Distributed Systems and the 2004 International Conference on Embedded and Ubiquitous Computing (EUC). He has served as the Director of the Center for Embedded System-on-a-chip Design funded by the New Jersey Commission on Science and Technology. He is the recipient of the AT&T Foundation Award and NEC Preceptorship Award for research excellence, NCR Award for teaching excellence, and Princeton University Graduate Mentoring Award. He has coauthored six papers that have won the Best Paper Award at ICCD'93, FTCS'97, ICVLSID'98, DAC'99, PDCS'02, and ICVLSID'03. A paper of his was selected for *The Best of ICCAD: A Collection of the Best IEEE International Conference on Computer-Aided Design Papers of the Past 20 Years* and another for *IEEE Micro* Top Picks of Computer Architecture Conferences. He was invited to deliver the keynote speech at EUC'05.