COWLS: Hardware–Software Cosynthesis of Wireless Low-Power Distributed Embedded Client–Server Systems

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

***Abstract*—In this paper, we present COWLS, a hardware–software cosynthesis algorithm that targets embedded systems composed of servers and low-power clients that communicate with each other through a channel of limited bandwidth, e.g., a wireless link. A novel scheduling algorithm is used to pipeline the execution of tasks that serve multiple clients associated with a given server. COWLS simultaneously optimizes the price of the client–server system, the power consumption of the clients, and the response times of tasks that have only soft deadlines, while meeting all of the hard deadlines. It produces numerous solutions that trade off different architectural features, e.g., price, power consumption, and response time, of an embedded client–server system. As far as we know, this is the first synthesis algorithm of its kind. We present the experimental results for numerous pseudorandom examples, a low-power client–server camera system, as well as the rest of the benchmarks within a publicly released embedded system synthesis benchmark suite.**

***Index Terms*—Client–server systems, embedded systems, genetic algorithms, hardware–software cosynthesis, multiobjective optimization, processor scheduling, real time systems, wireless communication.**

I. INTRODUCTION

A WIRELESS embedded client–server system is a special-purpose computer in which clients and servers communicate with each other via a channel of limited bandwidth. Clients are frequently consumer products, e.g., portable communication devices, for which price is often particularly important. Server price is also an important factor, although it is usually less important than client price because clients typically outnumber servers. In this work, we assume that servers have access to high-capacity power supplies. In order to maintain mobility, clients may be small and battery powered. Therefore, client-power consumption must be minimized to reduce heat production and increase battery life. Either clients or servers may initiate communication events.

The literature contains numerous case studies of embedded client–server system design and general descriptions of the client–server problem domain. Some researchers have discussed wireless and cellular systems [1], [2], some have focused on embedded systems in which the server is a satellite [3], [4], and others have studied telerobotics, systems in which a robot is partially or totally controlled via a limited-bandwidth communication channel [5], [6]. The majority of previous research on embedded client–server systems either surveys the problems typically faced by the designer of such systems or provides case studies detailing specific solutions to individual problems.

There is a significant body of work on hardware–software codesign, i.e., concurrent design of the hardware and software portions of an embedded system, and hardware–software cosynthesis, the automatic synthesis of hardware–software embedded systems [7], [8]. It should be noted that much of the work in hardware–software cosynthesis assumes the availability of a database describing the performance of different general-purpose and special-purpose processors when executing different types of tasks. This allows one to treat hardware and software implementations of the same task similarly during cosynthesis. Some work within the cosynthesis field concentrates on producing this database [9], [10].

The literature on solving the heterogeneous distributed embedded system hardware–software cosynthesis problem (the cosynthesis problem) can be placed within three main categories: 1) optimal solvers; 2) studies comparing different classes of optimization algorithms; and 3) potentially suboptimal heuristics. The hardware–software cosynthesis problem is composed of a number of subproblems, many of which are NP-complete, e.g., allocation-assignment and scheduling [11]. Presently, only potentially suboptimal algorithms are capable of synthesizing large distributed embedded system instances in a reasonable amount of time. Therefore, research on optimal solutions to hardware–software cosynthesis problem typically targets simplified versions of the problem or only tackles small problem instances. However, these approaches allow us to know the optimal solutions to some simple problems, making it possible to test the quality of potentially suboptimal heuristics when run on these problems.

Bender [12], Prakash and Parker [13], as well as Schwiegershausen and Pirsch [14] solved the cosynthesis problem with mixed integer linear programming (MILP). Kuchcinski used constraint logic programming to minimize the price of an embedded system under time constraints [15]. Lee *et al.* developed an $A^\star$ search algorithm in order to optimize embedded system resource allocation [16]. This algorithm uses earliest deadline

Manuscript received September 26, 2000; revised October 10, 2002. This work was supported in part by a National Science Foundation Graduate Fellowship, in part by Princeton University's George Van Ness Lothrop Fellowship in Engineering, and in part by DARPA under Contract DAAB07-00-C-L516. This paper was recommended by Associate Editor R. Gupta.

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

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

Digital Object Identifier 10.1109/TCAD.2003.819884

0278-0070/04$20.00 © 2004 IEEE

first scheduling integrated with a load balancing assignment algorithm borrowed from behavioral synthesis. However, it does not model intertask dependencies.

Some researchers compared different algorithms used to solve problems related to the cosynthesis problem. Axelsson compared the solutions produced by three different types of algorithms when run on a simplified version of the hardware–software cosynthesis problem [17]. Unfortunately, such comparative studies often suffer from time constraints, i.e., the problem definitions are simplified and easy to implement versions of meta-algorithms, e.g., simulated annealing, tabu search, and genetic algorithms, are compared.

In order to solve larger instances of the cosynthesis problem, many researchers have considered heuristics that take into account problem-specific information but are not guaranteed to arrive at optimal results. Researchers have developed iterative improvement algorithms [18], constructive algorithms [19], simulated annealing algorithms [20], evolutionary algorithms [21], [22], and a rapid, potentially suboptimal timing constraints solver [15]. Providing a complete survey of previous hardware–software cosynthesis research is beyond the scope of this article. However, a dissertation [23] and several research papers [8], [24]–[29] survey this research area,

COWLS synthesizes wireless embedded systems composed of servers and low-power clients that communicate with each other through a channel of limited bandwidth, e.g., a wireless link. Although a number of researchers have looked at specific manual architectural changes to allow power reduction in wireless systems, we know of no other algorithm that makes architectural power-aware wireless system design decisions automatically. In addition, COWLS uses a novel scheduling algorithm to pipeline the execution of tasks that serve multiple clients associated with a given server. This algorithm does not require client implementations to differ from each other. COWLS conducts Pareto-rank-based multiobjective optimization [30] to simultaneously optimize the price of the client–server system, the power consumption of the clients, and the response times of tasks that have only soft deadlines, while meeting all of the hard deadlines. It produces numerous solutions that trade off different architectural features, e.g., price, power consumption, and response time, of embedded client–server systems. We present experimental results for the recently released and publicly available embedded system synthesis benchmark suite (E3S) [10].

The paper is organized as follows. Section II describes the means by which embedded system behavior and timing constraints are specified to COWLS. In addition, it describes the models used for processing elements (PEs) and communication resources. In Section III we give a descriptive example to illustrate the sort of decisions COWLS must make during synthesis. Section IV formalizes the wireless client–server synthesis problem definition and describes the algorithms used within COWLS. In Section V, we introduce the E3S benchmark suite and present experimental results. We conclude in Section VI.

II. Embedded Client-Server System Problem Specification

In this section, we describe the inputs of the COWLS algorithm. The behavior of the synthesized embedded system and the timing constraints placed on this behavior are specified using client–server task sets, as described in the next subsection. The models for PEs, communication resources, and memory are described in Sections II-B–D.

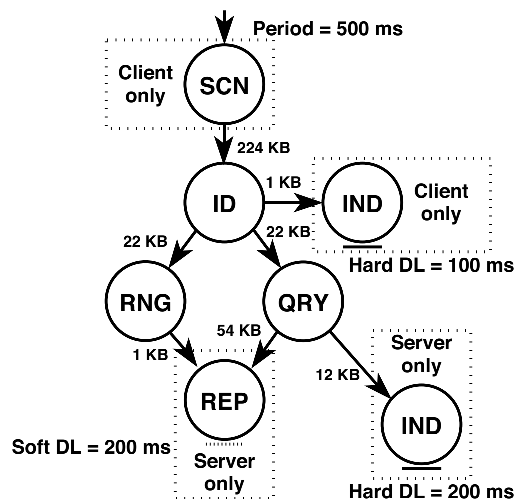

Fig. 1. Client–server task graph.

A. Client–Server Task Sets

COWLS accepts a description of the behavior and timing constraints of the embedded systems it synthesizes in the form of client–server task sets. A client–server task set is composed of one or more client–server task graphs. A task graph is a directed acyclic graph in which each node represents a task and each arc represents a data dependency and communication event. In Fig. 1, the top node represents the *SCN* task type and that node's outgoing arc represents a communication of 224 KB of data to the *ID* task. There may be more than one task instance of a given type, e.g., *IND* in Fig. 1. The *ID* task's incoming arc indicates that it may not begin execution until the *SCN* task has completed execution and transmitted 224 KB of data to it. Any task may have a hard deadline, signified by a solid bar, or a soft deadline, signified by a dotted bar. Thus, a designer may provide a specification containing only hard deadlines, i.e., real-time constraints, a specification containing only soft deadlines, or a specification containing any combination of the two types of deadlines. In the first case, COWLS will attempt to minimize price and power consumption while meeting all hard deadlines. In the second case, it will attempt to simultaneously minimize soft deadline violation, price, and power consumption. If all soft deadlines are set to zero, it will minimize system execution time, price, and power consumption.

In Fig. 1, the upper *IND* task must complete execution within 100 ms of the start of the task graph's execution. This hard deadline must be met for the synthesized architecture to be valid. The *REP* task should complete execution by 200 ms after the start of the task graph's execution. However, the system will still be valid if this soft deadline is not met. In the case of a task with a soft deadline, the aim is to minimize its finish time if the deadline cannot be met. The specification may require some tasks to be executed on the client, e.g., *SCN*. Others must be executed on the server, e.g., *REP*. A task graph's period is the amount of

time that elapses between its consecutive executions. The task graph's period is 500 ms in Fig. 1. Note that a system specification may contain multiple task graphs with different periods. A specification's hyperperiod is the least common multiple of the periods of its task graphs. A task graph may have a period that is less than, greater than, or equal to the maximum deadline within it. In addition, our model allows the representation of precomputation, postcomputation, streaming communication, and pipelined execution. Precomputation is the execution of a portion of a task before all of its input data have arrived. Postcomputation is the continued execution of a portion of a task after all of its output data have been transferred. Streaming communication is the transmission of data between two tasks during their execution.

B. PEs

A PE executes tasks. The PEs used within COWLS may be used to model general-purpose processors, digital signal processors (DSPs), application-specific integrated circuits (ASICs), and static-configuration field-programmable gate arrays (FPGAs). Note that the use of dynamically reconfigurable FPGAs in hardware–software cosynthesis has been addressed in other work [31], [32]. A solution may contain multiple instances of the same type of PE. COWLS models two classes of PEs: client PEs and server PEs. Client PEs may only exist within the client's PE allocation. Server PEs may only exist within the server's PE allocation. In general, server PEs have better performance than client PEs, but their power consumptions are higher.

COWLS requires a database that describes the relationships between tasks and PEs. Characterizing PEs in this manner requires that the designer know the input sequences that elicit the worst-case execution time for each task-PE pair. Alternatively, one may use worst-case performance analysis tools to determine an upper-bound on execution time, without requiring a specific input sequence [33], [34]. In addition, the average power consumption for each task-PE pair must be known or estimated. The power consumption of general-purpose and application-specific processors can be estimated by using models, simulation, and explicit analysis [35]–[40].

The following information establishes the relationships between tasks and PEs:

- a two-dimensional array indicating the worst-case execution time of each task on each PE;
- a two-dimensional array indicating the average power consumption of each task on each PE.

In addition to these arrays, each PE has a price, input/output energy per communicated bit, and idle power consumption. PEs may be buffered, in which case they can communicate and compute at the same time, or unbuffered, in which case communication and computation may not overlap in time. In the case of buffered communication, it is, of course, still necessary for a task's incoming data to arrive before it can begin execution.

C. Communication Resources

Each type of communication resource has a price per instance (to represent bus controller price), a maximum number of contacts, a price per contact (to represent bus bridge or interface circuit price), packet size (that can be very small to model communication that is not packet-based), energy consumption per packet, and transmission time per packet. A communication resource's number of contacts is the number of different PEs that it may connect together, i.e., a communication resource with two contacts is a point-to-point link and a communication resource with more than two contacts is a bus. Primary communication resources have four price values: the client and server have a price per instance and a price per contact. For primary communication resources, each contact is associated with a PE, on the client or server, that needs to be connected to the primary communication resource. The bus is assigned an appropriate controller price and a contact price equal to the price of a bus bridge.

The parameters of a communication bus can be determined from the bus specifications, as well as the controller datasheets. Each task graph edge must be assigned to a communication resource. The worst-case communication time and average power consumption of an edge are linearly dependent on the integer number of packets transferred via its communication resource. There may be more than one communication resource connected to a PE instance. In previous distributed computing work, it is commonly assumed that communication between tasks that are assigned to the same PE consumes an insignificant amount of time and power. We also make this assumption in COWLS. If an architecture contains two communicating tasks that execute on separate PEs, the architecture is invalid if there are no communication resources connecting the PEs.

D. Memory Model

COWLS uses a memory model in which each PE has a dedicated memory used by the tasks assigned to it. It might, at first, seem desirable to allow shared external memories in order to reduce the total quantity of memory, and number of packages, required in the embedded system. Unfortunately, using shared external memory requires that communication with memory be scheduled in a way that avoids contention between memory access requests by tasks assigned to different PEs. This would require detailed information about the exact times at which different tasks access memory. Gathering this information would be difficult; it would be processor-dependent and data-set dependent. In the absence of this information, in order to guarantee that hard real-time deadlines are met, it would be necessary to assume each task constantly accesses the shared memory during its execution. This would prevent multiple tasks from executing concurrently on different PEs, eliminating one of the major advantages of having multiple PEs. Therefore, COWLS associates dedicated memory with each PE.

COWLS compute the quantity of memory associated with each PE based upon code and data memory requirements. For each PE, COWLS requires an entry in the PE database giving the code size of each task type that may execute on that PE. The code memory for a PE in a solution's allocation is the sum of the code memory requirements of the tasks assigned to that PE. We do not currently have access to any benchmarks in which the data memory requirements of each task are given. Therefore, we make the assumption that each task requires an amount of data memory equal to the sum of the data quantities of its incoming

and outgoing communication events. Although this is a reasonable assumption for many dataflow tasks, it should be noted that this method of computing memory requirements could easily be changed in the presence of more detailed information about task data memory requirements. In order to compute the memory requirements for a PE, COWLS takes the maximum of the data memory requirements of all of the tasks assigned to it, and adds the sum of the code memory requirements of the tasks assigned to it. Note that task code could initially be stored in electrically programmable read-only memories (EPROMs), and transferred to PE local memories during system initialization. Memories commonly have sizes that are integer powers of two. In order to be conservative, COWLS ensures that each PE has a quantity of memory that is an integer power of two.

III. Descriptive Example

In this section, we show the types of design options COWLS explores during synthesis. Consider a system specification requiring a battery-powered camera to transmit digital images to a base station via a limited-bandwidth wireless link. If the designer has decided that the video information should be compressed, but has not yet decided what sort of processor should be used to carry out this operation, or even whether it should be done by the client or the server, COWLS will simultaneously explore the different options.

Fig. 2(a) shows one of the task graphs in the consumer benchmark from the E3S benchmark suite described in Section V-A. In this example, images must initially be generated on the client camera. They are then filtered, on either the client or server, converted to another image format, and compressed. Images must be transferred to the *sink* task within 2.5 s and, ideally, within 0.1 s. Image capturing (represented by the *src* node) must be carried out on the client. Data storage (represented by the *sink* node) must be carried out on the server. Storage has a hard deadline of 2.5 s and a soft deadline of 0.1 s. Printing has a hard deadline of 15 s and a soft deadline of 5 s. Display has a hard deadline of 15 s and a soft deadline of 1 s. For the sake of simplicity, we have ignored some tasks in the E3S Consumer benchmark while presenting this explanatory example.

The task graph shown in Fig. 2(a) carries out image acquisition (*src*), filtering (*filt-x*), conversion (*rgb-yiq*), data compression (*cjpeg*), and storage (*sink*). Using the client–server partitioning of this graph that is shown in Fig. 2(a), filtering, conversion, and data compression are executed on the client. The dotted rectangle containing the phrase *forced to client* indicates that the contained tasks must be assigned to the client. The dotted rectangle containing the phrase *forced to server* similarly indicates that the contained tasks must be assigned to the server. This partitioning reduces the load on the wireless communication link to 1 MB per task graph execution and allows an inexpensive primary communication resource to be used between the client and server. However, carrying out data compression on the client requires increased client price and power consumption.

In another possible partitioning, shown in Fig. 2(b), image acquisition (*src*) executes on the client and all other tasks execute on the server. In this partitioning, the client executes only

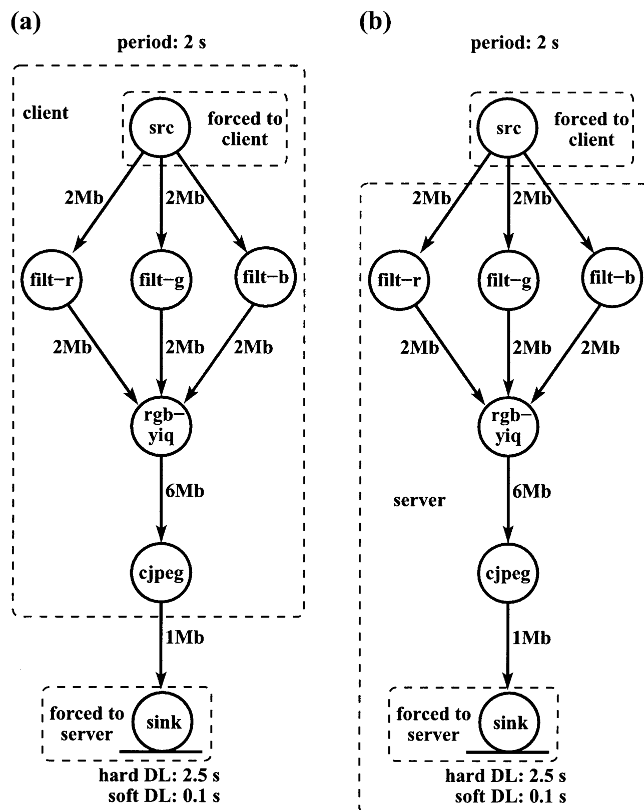

Fig. 2. Camera specification: (a) client–server assignment 1 and (b) client–server assignment 2.

essential functions, shifting all other computational burdens to the server. This decreases the client's price and power consumption. However, it increases the demands upon the communication link between the client and server, increasing its price and power consumption. Although some of the tradeoffs facing the designer of client–server systems are apparent even from this simple example, COWLS is capable of solving problems that are significantly larger and more complicated.

IV. Problem Formulation

In this section, we present the client–server synthesis problem formulation used for COWLS. In Sections IV-A and B, we define the client–server system problem and briefly describe the optimization infrastructure used by COWLS. In Section IV-C, we describe the manner in which problem-specific information was incorporated within this optimization infrastructure to allow good performance for client–server hardware–software cosynthesis. Section IV-D describes the initialization of solutions, i.e., candidate architectures. Section IV-E describes a client–server pipelining scheduling algorithm. We explain the manner in which a solution's costs are calculated in Section IV-F. Section IV-G presents our method of accelerating optimization by caching solutions.

A. Optimization Problem Introduction

This subsection describes the three main decisions that a wireless client–server system synthesis algorithm must make.

- **Allocation**: Determine the quantity of each type of resource, e.g., PEs or communication resources, to use. Determine which resources to use in the servers and which resources to use in the clients.
- **Assignment**: Select a resource to execute each task and communication event. In addition to assigning tasks to either the client or the server, they are assigned to specific PEs.
- **Scheduling**: Determine the time at which each task and communication event occurs. As we will discuss in Section IV-E, the complexity of scheduling is significantly increased by the interactions between clients and servers.

In addition to making these three decisions, COWLS must evaluate embedded system performance. The costs of an architecture, e.g., price, speed, area, and power consumption, must be computed.

Each of the three decisions, listed above, influences others. Therefore, attempting to consider a decision in isolation, or without feedback from subsequent decisions, is likely to result in poor-quality solutions. COWLS takes advantage of incremental feedback during optimization.

The independent synthesis of a client or server is similar to the distributed, heterogeneous embedded system cosynthesis problem. COWLS uses a parallel recombinative simulated annealing (PRSA) optimization infrastructure [41]. PRSA algorithms draw on the strengths of genetic algorithms and simulated annealing algorithms. PRSA algorithms are well suited to solving low-power, heterogeneous distributed system synthesis problems with soft and hard deadlines because they are resistant to becoming trapped in local minima, they excel at solving multiobjective problems, they can incorporate problem-specific knowledge in a straightforward way, and their runtimes do not increase rapidly with increases in problem instance size. PRSA algorithms also have some disadvantages. They are difficult to implement and they do not guarantee optimal solutions. Despite the first disadvantage, we have already completely implemented a software prototype of the COWLS algorithm. The second disadvantage is not particularly important for the class of problems targeted by COWLS. Heterogeneous distributed embedded system synthesis contains, within it, numerous problems, each of which is NP-hard. For example, even the simplest heterogeneous scheduling problem solved by COWLS, i.e., scheduling without client–server pipelining, is NP-complete. The allocation-assignment problem is also NP-complete [11]. Therefore, unless $P = NP$, any algorithm guaranteeing optimal solutions to this class of algorithm will require an amount of time that is, in the worst case, exponential in the size of the problem instance. In practice, moderate-sized and large problem instances are intractable to optimal solvers for closely related problem domains [13]. Although COWLS does not guarantee optimal solutions, when run on problem instances for which other solvers have arrived at provably optimal solutions, the optimization infrastructure used by COWLS also produces these optimal solutions [23]. In addition, this infrastructure produces results with better or equivalent quality than competing algorithms when run on publicly available problem instances for which the optimal results are not known. A detailed explanation of optimization infrastructure used in COWLS is well beyond the scope of this paper. However, a description of this algorithm, and comparisons with the results produced by optimization algorithms developed by other researchers, have previously been published [23].

Although the synthesis of an isolated client or server is related to the heterogeneous distributed system synthesis problem, COWLS targets the servers and clients simultaneously, and examines the consequences of allowing tasks to migrate between clients and servers. A designer may specify the behavior and timing constraints of a client–server system using a modified version of the model presented in Section II-A. This version also allows some tasks to have their assignment constrained to PEs in the clients or PEs in the server, although many tasks will be free to migrate between client and server during synthesis. In addition, the scheduling problem is dramatically changed by the existence of multiple identical clients per server.

During PRSA optimization, numerous solutions simultaneously exist in the algorithm's *solution pool*. Randomized local changes are made to individual solutions (mutation), and information is traded between solutions, in order to improve the quality of solutions (crossover). The number of solutions in the solution pool remains constant during optimization. It is, therefore, necessary to eliminate some solutions when new solutions are created via mutation or crossover. In order to determine the quality of each solution, solutions are ranked relative to the other solutions in the solution pool. Note that each solution has multiple costs. In order to impose a weak order on solutions, we use Pareto-ranking, i.e., a solution dominates another if all of its costs are lower than or equal to the other solution and a solution's Pareto-rank is the number of other solutions that do not dominate it (we used this somewhat counter-intuitive definition to maintain a positive correlation between rank and solution quality). In PRSA algorithms, solutions are selected for existence in the next generation by Boltzmann trials. A Boltzmann trial is a probabilistic selection mechanism with temperature-dependent behavior. At the start of the optimization algorithm's run, the temperature is high, and the higher-rank candidate loses the trial as frequently as it wins. As time progresses, the temperature is decreased until, at temperature zero, the higher-rank candidate is always selected.

COWLS synthesizes embedded systems containing arbitrary-topology busses and point-to-point communication links, as well as the primary communication resources that are used to connect clients and servers. There may be multiple communication resources within the client, and within the server. Different primary communication resources may be available. However, only one primary communication resource may be present in a client–server pair, as multiple wireless transmitters and receivers will typically result in unreasonably expensive client–server systems.

An architecture's costs are derived from the manner in which resources are used in its construction. Therefore, by attempting to meet real-time constraints, one ensures that high-speed PEs, well-suited to tasks they execute, are used for tasks that lie along critical paths in the task graphs. By attempting to minimize price, one ensures the use of PEs that are capable of carrying out the required tasks with minimal price. By attempting to min-

TABLE I
VARIABLE DEFINITIONS

C	set of clusters	
S_c	set of solutions in cluster c	
PE_c	set of PEs in cluster c	
execs (pe, t)	true if and only if pe can execute tasks of type t	
$\bar{a}$	Boolean inversion of a	
T	set of tasks in the problem	
CM	proportion of clusters to mutate	$\approx 1/4$
CC	proportion of clusters to subject to crossover	$\approx 1/4$
SM	proportion of solutions to mutate	$\approx 1/4$
SC	proportion of solutions to subject to crossover	$\approx 1/4$
soln-eval (s)	evaluates the quality of solution s as described in Sections 4.5 and 4.6	
dominates (a, b)	true iff each of a's costs is lower than b's	
E	client host type	
F	server host type	
G	the set of all task graphs	
T_g	the set of tasks in graph g	
R_c	the set of communication resources in cluster c	
	Note that additional variables are introduced in Figs. 3 and 4.	

imize client power consumption, one minimizes the number of power-intensive tasks run on power-hungry PEs located on the client. Of course, some of these goals conflict with each other. For this reason, a single run of COWLS generates multiple solutions that explore the tradeoffs among different costs.

B. *Optimization Algorithm Overview*

In this section, we provide an overview of the optimization algorithm used in COWLS.

This optimization algorithm used by COWLS maintains a collection of solutions. These solutions are organized into clusters. The solutions within a cluster all have the same allocation of PEs and communication resources. Solutions in different clusters may have different PE and communication resource allocations. Mutation is the application of a randomized, although not necessarily random, change to a data structure. Crossover is the exchange of information between two data structures. In COWLS, task assignment and communication resource connectivity mutation are applied to solutions. Task assignment and communication resource connectivity crossover occur between different solutions in the same cluster. PE and communication resource mutation and crossover are applied to clusters. This constrained application of PRSA operators prevents invalid solutions and clusters from being generated and speeds optimization [22].

Table I contains the definitions of a number of variables that are used in Figs. 3 and 4. Fig. 3 contains pseudocode for the initialization and optimization algorithms used in COWLS. Fig. 4 contains pseudocode for subroutines used within Fig. 3. COWLS consists of 20 000 lines of dense C++ code. Therefore, the pseudocode presented in these figures necessarily omits a number of low-level details.

Our goal, in providing these figures, is to give the reader a basic idea of the flow of the algorithm. To this end, we have sometimes presented a simplified algorithm in the pseudocode instead of showing the functionally equivalent but more complicated algorithm used in COWLS. For example, in COWLS, the for loop at the bottom of Fig. 3 and marked with an asterisk (*)

```
Initialization
  For each c ∈ C
    For each t ∈ T
      If ∀pe_j ∈ PE_c, execs(pe_j, t) = false
        Add a random type PE, pe_k, constrained by execs(pe_k, t) = true
      Select a PE, pe_l, for t as described in Section 4.4
      Assign t to pe_l

Optimization
  Select a random subset C^M ⊆ C s.t. |C^M| = CM · |C|
  For each c ∈ C^M
    clust-mutate(c)
    clust-eval(c)
  Build a set of cluster pairs, C^C, from C
  While |C^C| < CC · |C|
    Randomly select c_j, c_k ∈ C s.t. c_j ≠ c_k
    Add {c_j, c_k} to C^C
  For each cluster pair {c_l, c_m} ∈ C^C
    clust-crossover(c_l, c_m)
  For each c ∈ C
    Select a subset S_c^M ⊆ S_c s.t. |S_c^M| = SM
    For each s ∈ S_c
      soln-mutate(s)
      soln-eval(s)
    Build a set of solution pairs, S_c^C, from c
    While |S_c^C| < SC · |S|
      Randomly select s_j, s_k ∈ c s.t. s_j ≠ s_k
      Add {s_j, s_k} to S_c^C
    For each solution pair {s_l, s_m} ∈ S_c^C
      soln-crossover(s_l, s_m)
      soln-eval(s_l, s_m)
  ∀c ∈ C, s_c ∈ c set Q_c = 0, Q_{s_c} = 0
  * For each c_j ∈ C, s_k ∈ c_j, c_l ∈ C, s_m ∈ c_l
    If s_k ≠ s_m, dominates(s_k, s_m) = false
      Increment Q_{s_m}
      Increment Q_{c_l}
  Conduct Boltzmann trials on Q until ∀c ∈ C, |S_c| and |C| equal their initial values
```

Fig. 3. Optimization algorithm.

only conducts the necessary subset of comparisons. It does not call **dominates** (s_i, s_j) for every s_i, s_j pair.

In Fig. 3, the subroutines **clust-mutate** and **clust-crossover** are analogous to the **soln-mutate** and **soln-crossover** subroutines shown in Fig. 4. Unlike **soln-crossover** and **soln-mutate**, PE and communication resource allocation mutation and crossover are conducted. The pseudocode for these subroutines has been omitted for the sake of brevity.

In Fig. 4, note the line in the **soln-crossover** subroutine that is marked with a dagger (†). This line describes the selection

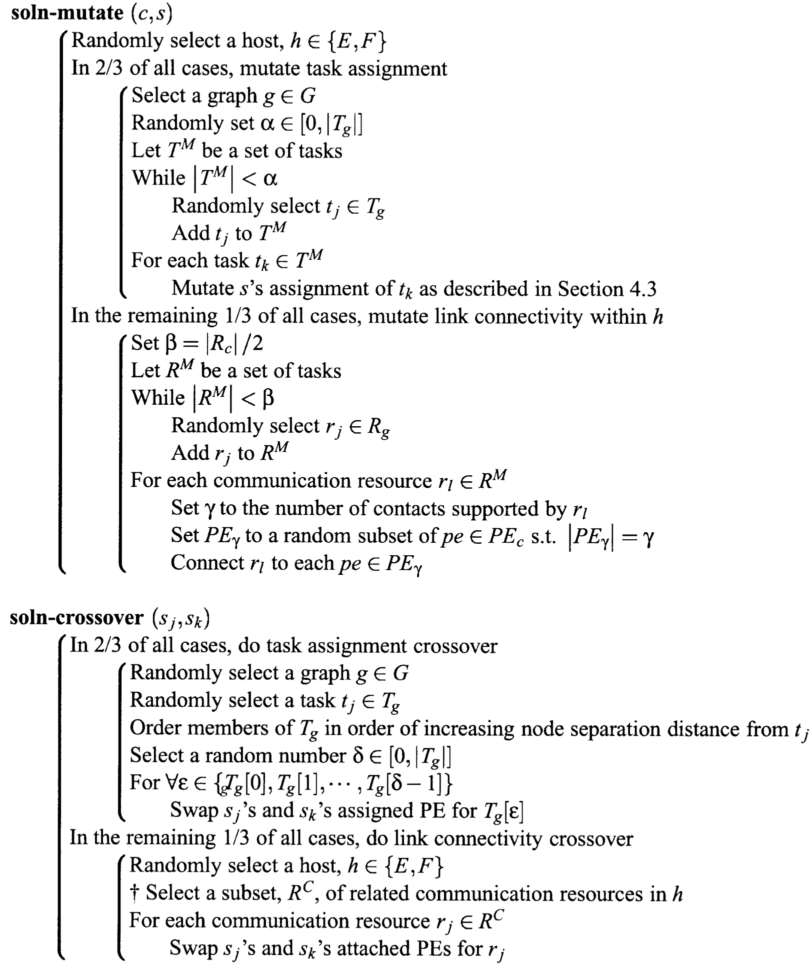

Fig. 4. Optimization algorithm subroutines.

of related communication resources. These communication resources are related in that the characteristics defining them, e.g., transmission rate, price, and power consumption, are similar. In PRSA and genetic algorithms, it is important, for performance reasons, to avoid breaking up related components of a solution during crossover, i.e., it is important to preserve locality [42]. We have previously described some methods for keeping related attributes together during crossover [23].

The **dominates** subroutine referred to in Fig. 3 takes solutions or clusters as parameters. It returns true if each cost of its first argument is lower than or equal to the corresponding cost of its second argument and all costs are not equal. Using this subroutine to define quality (Q) makes COWLS a Pareto-rank-based optimization algorithm, i.e., a solution is defined to have a higher quality than another if and only if all of its costs are lower.

The Boltzmann trials described at the bottom of Fig. 3 are temperature-dependant probabilistic events [41] to which two solutions or clusters are subjected. When COWLS starts optimization, the results of these trials are random. However, as optimization continues, it becomes increasingly likely that the higher quality candidate is selected and the lower quality candidate is eliminated. Using this type of trial makes it unlikely that a search will become trapped in local minima. Note that, although Boltzmann trials are conducted normally, the use of Pareto-ranking requires intercluster and intersolution comparisons that would make a parallel implementation more difficult than initially envisioned by Mahfaud and Goldberg [41]. However, we believe that the advantage of Pareto-ranking, for the inherently multiobjective problem tackled by COWLS, outweighs the potential advantage of easier parallel implementation resulting from tightly constrained solution and cluster comparisons.

C. Guided-Task Assignment Mutation

Each solution contains an allocation of hardware resources and an assignment of tasks and communication events to resources, as described in Section IV-A. Changes are made to solutions via mutation and crossover (see Section IV-A). Although it would be possible to use simple crossover and mutation operators, the quality of results for some classes of problems can be improved by incorporating problem-specific information into these operators. We describe a sophisticated crossover operator in another publication [23]. A description of the task assignment mutation operator used in COWLS follows.

A desire for improved performance when solving the wireless client–server embedded system synthesis problem motivated us to incorporate problem-specific knowledge within task assignment mutation heuristic used by COWLS. This change also resulted in improved performance for other problem domains. In

this subsection, we describe this guided-task assignment mutation algorithm.

As described in the previous subsection, mutation makes randomized changes to task assignments. However, these changes need not be entirely random; they may be guided by problem-specific heuristics. We have developed a guided-task assignment mutation algorithm that attempts to minimize PE overuse, task execution time, and communication time. After randomly selecting a task to be reassigned, this heuristic generates an array of PEs capable of executing it. Three costs are associated with each PE in the solution's allocation: communication time, execution time, and loading.

Communication time is a metric that takes into account the impact of a change to a task's assignment upon the amount of time required to transmit incoming and outgoing data. A task's neighbors are the tasks with which it communicates, i.e., the tasks connected to it by arcs as shown in Fig. 1. Let $Q_{a,b}$ be the quantity of data, in bits, transferred along the edge between a task, a, and one of its neighbors, b. Let function $\mathbf{ctime}(q, P_a, P_b)$ give an estimate of the amount of time required to transmit q bits of data between the PE, P_a, to which task a is assigned and the PE, P_b, to which task b is assigned. In a distributed system, we approximate the amount of time required to transmit information between a pair of PEs based on the average data transmission rate of the communication resources in that solution's allocation. We previously computed the set of communication resources between each pair of PEs to more accurately approximate communication time. However, the CPU time required for this operation was too costly to justify the potential for improved estimation. COWLS maintains separate average data transmission rates for the communication resources in the client, the communication resources in the server, and the wireless communication resource. The communication time $C_{P,T}$ for each PE, P, a task, T, might potentially be assigned to is the sum of the communication times for communication between that task and all of its neighbors, set N_T, i.e.,

$$C_{P,T} = \sum_{i \in N_T} \mathbf{ctime}\,(Q_{T,i}, P, P_i).$$

We attempted defining communication time as the maximum communication time for any neighbor of the task under consideration. However, using a sum instead of a maximum resulted in better solution quality.

In addition to communication time C_T, we use execution time to prioritize PEs to which a task might potentially be assigned. Execution time is the amount of time required to execute the task on the PE under consideration. Our final prioritization metric is loading, the proportion of a PE's time, in the system hyperperiod, that has already been occupied by the other tasks assigned to it, i.e., if h is the system hyperperiod, T_P is the set of all tasks assigned to PE P, and function $\mathbf{etime}(P, T)$ is the time required to execute task T on PE P, then the execution time $E_{P,T}$ for each PE, P, a task, T, might potentially be assigned to is defined as follows:

$$E_{P,T} = \sum_{i \in T_P} \frac{\mathbf{etime}\,(P, i)}{h}.$$

Unless all PEs are overloaded, i.e., have a loading greater than or equal to one, overloaded PEs are not considered legitimate targets for task assignment.

Note that we have three metrics for the quality of PEs to which a task's assignment might potentially mutate. We rank candidate PEs through Pareto-ranking. We considered using only two costs in this Pareto-ranking: loading and the sum of communication time and execution time. However, we found that leaving communication time and execution time separate until Pareto-ranking resulted in better solutions. After ranking, PEs are sorted by their ranks. We empirically determined that better results were produced when PEs of the same rank were randomly ordered, i.e., COWLS does not allow solution encoding to bias task assignment decisions. Once the PEs are ordered, we select one by indexing into the array of PEs using a random variable with a probability density function (PDF) that favors PEs with the highest rank. We tried using a number of different indexing functions but settled on a mathematically elegant approach that produces good results [23]. We generate a stream of pseudorandom values with a PDF that can be smoothly scaled between a flat PDF and a triangular PDF peaking at the PE with the highest rank.

In addition to guiding-task assignment mutation, we also probabilistically constrain differences in task assignment mutation between different copies of the same task in the hyperperiod. We allow tasks in different copies of a task graph to be assigned to different PEs. However, we have developed a more flexible way of integrating control of these task assignment probabilities into the PRSA algorithm. We allow the user to provide a parameter specifying the probability, per task assignment mutation, that the mutation will affect all of a task's copies instead of only one task copy. This allows arbitrary combinations of task assignments to be explored while making it possible to focus the search on promising areas of the solution space in which most copies of a task are assigned to the same PE. The designer may specify the proportion (a value greater than 0.9 works well in practice) of task assignment changes that are made to all copies of a task, and the proportion of the changes that are made to only a single copy. Note that this term also used in task assignment crossover.

D. Initialization

At the start of the optimization algorithm's run, the initial solution pool must be populated. A user-defined number of solution *clusters* is created, each of which contains a user-defined number of solutions. Solutions within the same cluster have the same PE and communication resource allocations. This simplifies and accelerates optimization, as described in past work [22]. Constructive algorithms are used to initialize cluster allocations, communication resource allocations, task assignments, and communication resource connectivities.

In the first step of PE allocation initialization, it is ensured that, for each type of task in the task set, there is at least one PE capable of executing the task. This is accomplished by iteratively finding a task that cannot be executed by any of the PEs in the allocation, and adding a randomly selected PE of a type capable of executing the task. Note that, even after this step, it is still possible that there are too few resources to execute all of

the tasks in the system before their hard deadlines. In the next step, additional PEs are randomly added until there are sufficient hardware resources to execute all tasks within an amount of time equal to the hyperperiod multiplied by a scalar, g. The value g is proportional to twice the ratio of the index of the cluster to the total number of clusters, i.e., some clusters will have few PEs in their allocation and others will have many. This allocation diversity in the initial solution pool improves optimization.

After a PE allocation has been decided, task assignments are initialized by a two-stage algorithm. In the first stage, information is not yet available about communication times. Therefore, a modified version of the algorithm described in Section IV-C is used to assign each task to a PE. This algorithm considers all of the criteria of the guided-task mutation algorithm, with the exception of communication times. After the first stage of task assignment initialization is complete, the second stage reassigns each task using the full guided-task assignment mutation algorithm, i.e., it considers the communication times associated with different potential task assignments. Communication resource connectivity is initially random, i.e., each contact of a communication resource is attached to a randomly selected PE.

Task assignments are modified using the algorithm described in Section IV-C. This algorithm was originally designed to improve the performance of our optimization infrastructure when synthesizing client–server systems containing low-bandwidth communication resources. By considering the expected impact upon bandwidth caused by each potential change in task assignment, COWLS is able to avoid task assignments that result in increased communication time without compensating improvements in computation time or compensating reductions in PE overloading.

E. Scheduling and Client–Server Pipelining

In this section, we describe the scheduling algorithm used in COWLS.

In order to determine a solution's client power consumption, soft deadline violation, and hard deadline violation, it is necessary to generate its complete schedule. COWLS uses a rapid multirate list scheduler that is capable of handling task graphs with periods that are greater than, equal to, or less than the deadlines in the task graphs. The scheduler treats time as circular, i.e., an event that occurs at one point in time also occurs at every integer multiple of the hyperperiod from that point in time. This scheduler operates in two stages. During the first stage, the scheduler determines a priority for each task. During the second stage, communication events are assigned to communication resources, communication events are scheduled, and tasks are scheduled.

In order to prioritize tasks, the approximate earliest finish time (EFT) and latest finish time (LFT) of every task are determined by conducting a modified breadth-first search of each task graph. At this point, task assignments are fixed. Therefore, the execution time of each task is known. Communication event assignments are not fixed when EFT and LFT calculations are carried out. Therefore, it is not possible to know the exact amount of time required to carry out each communication event. A communication event's time is approximated by taking the maximum amount of time required by the event on any of the communication resources that connect the PEs to which the communication event's parent and child tasks are assigned. Raw times are used for EFT and LFT computation, i.e., these time values are not multiplied by the number of clients per server. A more detailed explanation of this decision requires knowledge of the method of pipelining used in COWLS. We explain this concept later in this section. Slack is the difference between a task's LFT and its EFT. The scheduler uses negative slack in order to prioritize task scheduling, i.e., low-slack paths in the task graphs have high scheduling priorities. If the schedule produced in this manner fails to meet all hard real-time deadlines, COWLS retries scheduling using negative LFT and negative earliest start time (EST) for prioritization.

Once tasks are prioritized, the second scheduling stage is entered. During this stage, the contents of a continuously updated prioritized list of tasks, whose data dependencies have been satisfied, are iteratively scheduled. Recall that some task graphs will be scheduled multiple times during the hyperperiod. Given that c is the offset of a task's copy in the hyperperiod, m is the maximum copy number for a given task, then a task's proportional copy number, p, is defined as follows:

$$p = \frac{c}{m}.$$

Tasks are sorted in the following manner. If the slacks of the tasks are unequal, the task with the lower slack is scheduled. If slacks are equal, the task with the lower proportional copy number is scheduled.

When a task is selected for scheduling, each of its incoming communication events is first scheduled on one of the communication resources connecting the PEs to which the task and its parent are assigned. The communication resource that allows the communication event to finish at the earliest time is used. If the tasks are assigned to the same PE, communication is treated as instantaneous. If they are assigned to PEs separated into the client and server, the communication event is scheduled on the primary communication resource, i.e., the wireless link.

While scheduling, bus contention is explicitly simulated. The scheduler is deterministic, i.e., given a particular resource allocation and task assignment, it always produces the same complete, static schedule. Therefore, after scheduling, the worst-case completion times of each task and communication event are known. This allows straightforward calculation of soft and hard deadline violations. In addition, the scheduler determines the communication resources upon which each communication event occurs. This information allows the calculation of power consumed by the client's communication resources. The energy consumption of each task that executes on the client, as well as the energy consumed by the client PEs while idle and communicating, are added to the energy consumption of the client communication resources and divided by the system hyperperiod to determine the total client power consumption.

Recall that there may be multiple clients per server. It is necessary to ensure that a server is capable of executing the tasks associated with each client. The most straightforward way of accomplishing this is to multiply the execution times of the tasks and communication events on the server, and the communication events between the server and clients, by the *client–server*

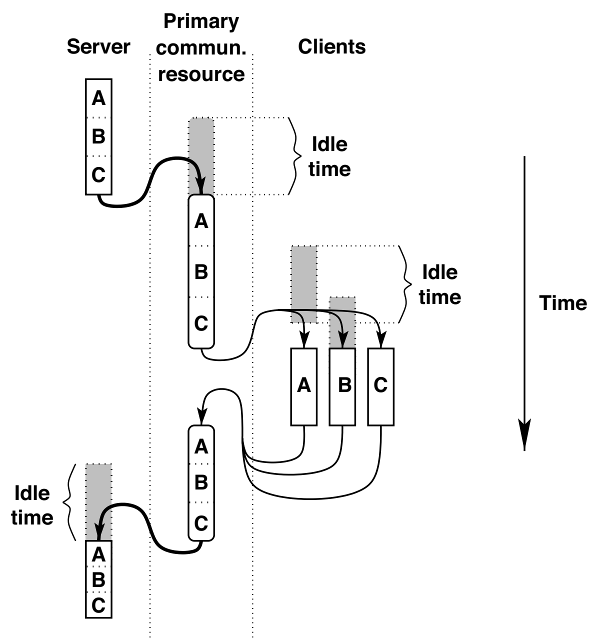

Fig. 5. Part of a nonpipelined schedule.

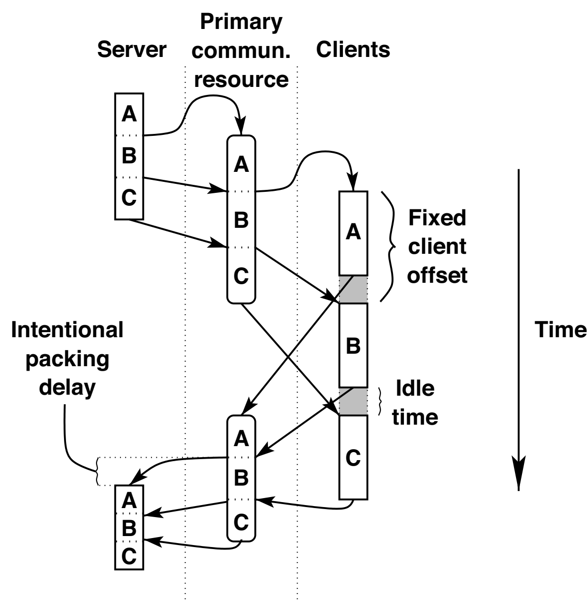

Fig. 6. Part of a pipelined schedule with a large client offset.

ratio, i.e., the number of clients per server. However, in order to ensure that this straightforward approach is correct, it is necessary to delay the execution of the corresponding tasks on each client until all of the tasks have received the data upon which their execution depends, and provide buffers for the transmitted data.

Consider the schedule portion shown in Fig. 5. Time increases from the top of the figure to the bottom. The left column depicts the schedule for the server. In the top rectangle of this column, each of the three portions (A, B, and C) corresponds to a task associated with one of three clients. In this figure, a straightforward, nonpipelined method of scheduling is used. The communication events that transmit data from the server to the client do not begin until the tasks associated with each client have completed execution. Similarly, none of the clients begins execution until data have been transmitted to each client. This results in the primary communication link and clients sitting idle when they might otherwise be carrying out work. There are a number of ways that this problem might be remedied.

One possible approach is to explicitly schedule each client separately, thereby allowing every task to execute as soon as its incoming data are ready. This approach has two disadvantages, one tolerable and one intolerable. Scheduling each client separately would increase the average runtime of the scheduler by a factor of the client–server ratio. However, this synthesis-time cost might be tolerable if the increased scheduling flexibility resulted in improved schedules. More importantly, this approach would result in each client having a different schedule. We considered the resulting increased complexity of manufacturing, debugging, and maintaining such a system sufficient to disqualify this approach. To give some idea of the problems associated with such a scheme, note that it would require the maintenance of a number of client designs equal to the client–server ratio.

The approach we selected gains a significant amount of scheduling flexibility without sacrificing synthesis-time efficiency or dramatically increasing the complexity of producing, debugging, and maintaining the embedded system. We pipeline the execution of tasks and communication events associated with different client copies. However, we constrain each client to the same schedule. Each client's schedule is offset, in time, by a fixed duration from every other client's schedule. The approach may be most directly illustrated with the aid of a diagram. Fig. 6 is analogous to Fig. 5. However, it shows a portion of a schedule produced using our pipelining approach. Note that the first series of communication events (shown at the top of the center column) may begin as soon as their parent tasks have completed. Similarly, the client task may begin execution as soon as their data have arrived, under the constraint that each client task must be separated from its corresponding task in other clients by a fixed amount of time, the client offset. As a result of adhering to a fixed client offset, it is only necessary to produce one client schedule explicitly. Each of the other client schedules is equivalent to the explicit schedule offset in time by an integer multiple of the client offset.

As described earlier in this section, the raw execution time of tasks is used during EFT and LFT calculation. Pipelining frequently allows tasks to be scheduled as soon as the corresponding incoming communication event has completed. Therefore, using raw task and communication event durations allows more accurate EFT and LFT estimates than using task and communication event durations multiplied by the client–server ratio.

Consider the second set of server tasks in Fig. 6. Note that the task associated with client copy A begins execution after its incoming data are ready. This intentional packing delay is introduced to ensure that the tasks are scheduled as one contiguous event. We considered the alternative of allowing the tasks to be separated by an arbitrary amount of time. However, this leads to a dramatic increase in the time complexity of the scheduling al-

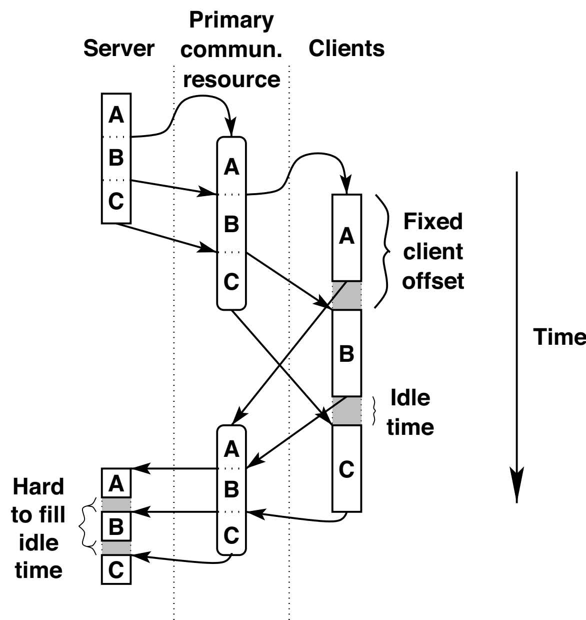

Fig. 7. Part of a pipelined schedule without packing.

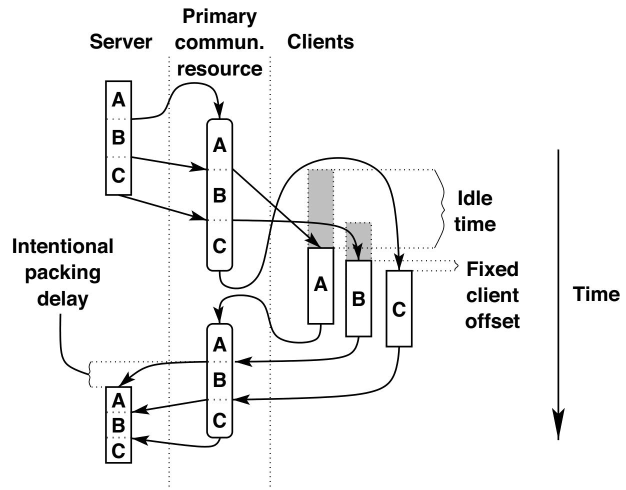

Fig. 8. Part of a pipelined schedule with a small client offset.

gorithm with little gain in scheduling flexibility. By allowing arbitrary gaps between the scheduling events of the tasks and communication events associated with different clients, one introduces numerous (a number equal to the client–server ratio minus one, in general) gaps into the schedule every time an event is scheduled. Scheduling complexity is increased, not only by the necessity of checking each of these gaps when every new event is scheduled but, more importantly, by the necessity of finding a location for new events, each of which consists of a pattern of gaps and active periods. We avoid these problems by making a set of tasks, associated with different clients, contiguous.

Even if allowing noncontiguous scheduling of the events associated with different clients did not grossly increase computational complexity, it would be of dubious benefit. Fig. 7 shows a portion of a pipelined schedule without packing. Consider the second server task set, to the lower left. By allowing arbitrary delays between the tasks associated with different clients, we have traded a moderate idle slot in a position where it can easily be filled or masked by other tasks in practice, for numerous (equal to the client–server ratio minus one) small idle slots that increase the computational complexity of scheduling and are difficult to fill or mask. These observations led us to use the packing approach.

We initially considered the selection of the client offset to be an important problem. Compare the client schedules of Figs. 6 and 8. In the first case, idle time is introduced between client tasks by using a client offset that is larger than the ideal offset. In the second case, the execution of the first client's task is delayed in order to enforce the constraints imposed by a client offset that is smaller than the ideal offset. Unfortunately, it is necessary to use a single client offset for all tasks in order to ensure that all client schedules are identical. Therefore, the client offset is likely to be too large for some tasks and too small for others in any problem of moderate complexity. One must select a client offset that provides a good tradeoff between these two alternatives. We set the client offset to be equal to the average time required by the communication events assigned to the primary communication resource. We experimentally determined that the qualities of the results produced by a synthesis run are not strongly dependent upon the client offset, as long as a few conditions hold. An explanation of this phenomenon and a comparison between nonpipelined and pipelined scheduling are presented in Section V.

F. Cost Calculation

In this section, we describe the process by which a solution's costs are calculated.

After making changes to solutions, it is necessary to determine whether or not those changes resulted in improved costs. Thus, after modifying a solution, COWLS carries out cost calculation to determine its aggregate price, the client's power consumption, and the degree to which soft deadlines are violated. In addition to these visible costs, there are a number of hidden costs that need never be displayed to the designer. Hard deadline violation is an example of such a cost. All solutions in which the hard deadline violation is nonzero are eliminated before results are presented to the designer. However, during optimization, solutions with hard real-time deadline violations are allowed to exist, for they have the capacity to evolve into high-quality, valid solutions during optimization. Soft deadline violation proportion is the sum of the soft deadline violation times in every copy of each task graph, divided by the hyperperiod.

Once a schedule is computed for a solution, that solution's client power consumption and soft deadline violation information is stored in a cache (see the next section) and used for any equivalent solutions that subsequently arise during optimization. Aggregate price is computed by taking the sum of the prices of the PEs, task execution memory, communication buffer memory, communication resources, and the primary communication resource associated with the client, multiplying this by the expected number of clients, and adding to this the sum of the prices of the resources used in the server multiplied by the expected number of servers. This gives a total client–server system price.

G. Solution Cache

Every time a solution is changed, it is necessary to determine its new cost. Carrying out cost evaluation every time a solution changes would be the most straightforward approach. However, solution evaluation, which requires scheduling as well as other time-consuming components of cost calculation, is the most time-consuming operation undertaken by our algorithms. In order to avoid needless solution evaluations, COWLS maintains a cache of solution cost sets to prevent the reevaluation of solutions after every modification. In our algorithms, scheduling, floorplanning, and bus topology generation are deterministic. Therefore, for any PE allocation, task assignment, link allocation, and link connectivity, there exists exactly one system cost set. Thus, any solution is characterized by a small amount of information, relative to the amount of information computed during cost evaluation.

Sometimes, solution mutation and crossover produces a solution identical to one for which cost calculation was previously done. In these cases, the solution's cost set is retrieved from a cache, making it unnecessary to carry out cost evaluation. We use a least-recently used (LRU) replacement policy. The cache size is dynamically controlled based on total memory usage, i.e., we allow more entries to exist if the entries consume little memory. Our experimental results indicate that the cache is usually hit 50% of the time. Its use generally cuts synthesis time in half.

V. Experimental Results

In this section, we present experimental results and discuss their implications. These results provide a reference point for other researchers, suggest the superiority of certain synthesis tool design decisions, and allow a deeper understanding of the client–server synthesis problem. COWLS targets a new problem domain. Note that, in Section IV-A, we discussed the suitability of the optimization infrastructure used by COWLS, and referred readers to a comparison of its performance with that of past work.

A. E3S

We have developed an embedded system synthesis benchmark suite, called E3S, based on data from EEMBC [43]. The first release of E3S contains 17 processors, e.g., the AMD ElanSC520, Analog Devices 21 065L, Motorola MPC555, and Texas Instruments TMS320C6203. These processors are characterized based on the measured execution times of 47 tasks, power numbers derived from processor datasheets, and additional information, e.g., die sizes, some of which were necessarily estimated, and prices gathered by e-mailing and calling numerous processor vendors. In addition, E3S contains communication resources modeling a number of different busses, e.g., CAN, IEEE1394, PCI, USB 2.0, and VME. As well as containing models for a number of conventional busses, these benchmarks contain models for IEEE 802.11, Bluetooth, and GSM wireless links. These task sets follow the organization of the EEMBC benchmarks. There is one task set for each of the five application suites: automotive/industrial, consumer, networking, office automation, and telecommunications. This benchmark suite has been publicly released and is available via HTTP [10].

TABLE II
Multiobjective Optimization

Example	Price ($)	Average power (mW)	Soft DL viol. prop.
Automotive-Industrial	518	186	0.00
	563	98	0.00
Networking	372	136	0.86
	408	136	0.74
	423	134	0.86
	507	134	0.80
	543	134	0.75
	583	135	0.74
Telecom	659	143	1.62
	659	147	0.98
	659	152	0.64
	660	146	0.92
	673	143	1.52
	996	366	0.58
	1168	145	0.88
	1559	327	0.62
	1684	343	0.52
	2902	344	0.51
Consumer	610	126	0.98
	1038	167	0.61
	2890	165	0.65
Office automation	344	159	1.14
	385	158	0.71
	418	157	0.75
	436	157	0.72
	476	172	0.69
	492	164	0.69
	651	162	0.68
	664	162	0.67
	893	158	0.68

B. Multiobjective Optimization for the E3S Benchmarks

This section presents the result of using COWLS to conduct multiobjective optimization on the E3S benchmarks described in Section V-A. We used a publicly available version of these benchmarks in which at least one task in each task graph has its assignment locked to the client and at least one has its assignment locked to the server, e.g., the Consumer benchmark discussed in Section III. A fully functional prototype of COWLS has been implemented in approximately 20 000 lines of C++ code with heavy use of the standard template library (STL). Three Linux machines were used to synthesize architectures for these benchmarks: a Pentium III running at 900 MHz, an Athlon Thunderbird running at 1.4 GHz, and an Athlon running at 650 MHz.

Table II shows the sets of solutions produced for the five task sets in the E3S benchmark suite. There are five clients for each server. For these benchmarks, COWLS was used to explore the tradeoffs among different system costs, instead of attempting to minimize a single cost. Given a similar amount of CPU runtime, it would be possible to better optimize a single

cost by ignoring all other costs. However, this approach would ignore the fundamentally multiobjective nature of embedded system design. COWLS took between 13 and 80 CPU min when run on each of these benchmarks. We rounded the prices and power consumptions of the solutions up to the nearest dollar and milliwatt. For most of the benchmarks, COWLS found numerous solutions that trade off price, average power consumption, and soft deadline violation proportion. Note that COWLS produced multiple solutions for each benchmark. In particular, let us revisit the camera (E3S Consumer) example we described in Section III. COWLS produced three wireless client–server architectures for this example. These architectures had prices ranging from $610 to $2,890, power consumptions ranging from 126 to 167 mW, and soft deadline violation proportions ranging from 0.61 to 0.98. The first solution to the camera example contains an IBM PowerPC 405GP running at 266 MHz on the client, an ST Microelectronics ST20C2 running at 50 MHz on the server, and an IEEE 802.11b Lucent WaveLAN card. COWLS found that the requirements placed on the wireless communication resource could be significantly reduced by assigning all tasks, prior to compression, to the client, as shown in Fig. 2(a). The other two solutions have relatively more PEs and communication resources and use these resources to reduce soft deadline violation. Note that the E3S Consumer benchmark contains another, printing and display, graph in addition to the data acquisition graph shown in Fig. 2(a) and (b).

TABLE III
PRICE-ONLY PIPELINING COMPARISON EXPERIMENTS

Example	Price with pipelining	Price without pipelining	Example	Price with pipelining	Price without pipelining
1	525	526	2	885	448
3	547	653	4	671	671
5	686	849	6	845	861
7	542	1092	8	617	618
9	719	910	10	561	1035
11	513	590	12	583	408
13	3277	n.a.	14	954	954
15	740	622	16	695	1003
17	491	500	18	1455	n.a.
19	1149	754	20	829	773
21	726	809	22	n.a.	1017
23	663	663	24	874	n.a.
25	431	586	26	465	716
27	444	570	28	919	n.a.
29	1564	1564	30	1442	n.a.
31	430	430	32	557	515
33	1020	690	34	952	558
35	440	603	36	1016	1127
37	657	657	38	420	441
39	464	927	40	820	787
41	618	988	42	914	1017
43	843	1369	44	714	714
45	615	868	46	832	758
47	744	n.a.	48	897	825
49	754	n.a.	50	2085	n.a.
Improved: 31					
Degraded: 12					

C. Evaluation of Client–Server Pipelining

In the interest of evaluating the performance of the client–server pipelining algorithm described in the previous section, we did a number of experiments in which we compared different versions of pipelining scheduler with each other, and with a straightforward nonpipelining scheduler. Multiobjective optimization significantly complicates presentation of, and comparison between, the results of different optimization runs because each run produces numerous examples. For these comparative examples, we had COWLS ignore soft deadline violation and power, concentrating only on price optimization. As a result, each run produces only one result. We rounded the prices of the solutions up to the nearest dollar.

Table III shows the result of running COWLS on 50 examples in which the processors come from the E3S benchmark suite and the task sets are randomly generated using parametric pseudorandom task graph software [44]. Clients were offset from each other by the average amount of time taken per communication event assigned to the primary (wireless) communication resource. Quality improved 2.6 times as frequently as it degraded. For each processor, we generated a server version and a client version. The server version is identical to the E3S processor. The client version has one-fifth the power consumption of the E3S processor and five times the execution time for each task, but is otherwise identical. Our task sets each contains 12 tasks. Each task type is randomly selected from the networking and telecom E3S benchmarks. Each communication event has a quantity of $1 \cdot 10^3$ bits (1 kb). Approximately one third of the tasks must be assigned to a client, one third must be assigned to the server, and one third may be assigned to either client or server. There are five clients for each server. There is no guarantee that every example generated in this manner will have a valid solution. In this table, an entry of n.a. indicates that no solution was found for the problem and parameters associated with the entry. For cases in which no solutions were found by either the client communication pipelining or client communication nonpipelining version of COWLS, we omitted the example from the table.

We found that pipelining schedules usually results in an improvement to solution quality. As shown in Table III, solution quality improved approximately two and a half times as frequently as it degraded. Although there were some cases when using a nonpipelining scheduler allowed the production of a superior solution, one should not draw the conclusion that it would be wise to run the scheduler in pipelining and nonpipelining mode for every cost evaluation and take the best cost. By doubling the amount of time required for each solution evaluation, one would halve the number of solutions that may be evaluated. One could, instead, use the scheduling method that is generally superior, i.e., the pipelining scheduler, and allow a more thorough exploration of the solution space, guided by the evolutionary algorithm, in the same amount of time.

As discussed in Section IV-E, we had initially considered the selection of a client offset value to be an important problem. However, in practice, solution quality is highly resistant to degradation. Let us define the client offset factor as a scalar by which the average primary communication event duration is multiplied to calculate the client offset value. Varying this factor

from zero to two results in only small changes to the number of cases for which pipelining resulted in an improvement to solution quality. Solution quality remains mostly independent of the client offset factor until it approaches the ratio of primary link communication time to computation time. Examining the schedules with a simple graphing tool revealed that, up until this point, the task delays required due to dependency on data transmitted via the primary communication link mask the idle slots that result from having a large client offset value.

VI. Conclusion

Despite the previous work dealing with embedded client–server systems and hardware–software cosynthesis, we know of no previous work that automatically synthesizes such systems. COWLS automatically synthesizes embedded client–server systems. It uses a multiobjective PRSA algorithm to simultaneously produce multiple solutions that trade off different costs. It optimizes price, client power consumption, and soft deadline violations under hard real-time constraints and constrained client–server communication bandwidth. COWLS incorporates a novel and tractable scheduling algorithm that pipelines the execution of tasks associated with different clients while maintaining identical client schedules. This form of pipelining has been found to improve solution quality in the majority of cases.

References

[1] D. Halchin and M. Golio, "Trends for portable wireless applications," *Microwave J.*, vol. 40, pp. 62–78, 1997.

[2] S. Komaki and E. Ogawa, "Trends of fiber-optic microcellular radio communication networks," *IEICE Trans. Electron.*, vol. E79-C, pp. 98–103, 1996.

[3] G. Comparetto and R. Ramirez, "Trends in mobile satellite technology," *IEEE Comput.*, vol. 30, pp. 44–52, Feb. 1997.

[4] F. Ananasso and F. D. Priscoli, "Issues on the evolution toward satellite personal communication networks," in *Proc. Global Telecommun. Conf.*, Nov. 1995, pp. 541–545.

[5] R. E. Barry and J. P. Jones, "Rapid world modeling from a mobile platform," in *Proc. Int. Conf. Robotics & Automation*, Apr. 1997, pp. 72–78.

[6] D. W. Gage, "Telerobotic requirements for sensing, navigation, and communications," in *Proc. Nat. Telesyst. Conf.*, May 1994, pp. 145–148.

[7] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1994.

[8] G. De Micheli and R. K. Gupta, "Hardware/software co-design," *Proc. IEEE*, vol. 85, pp. 349–365, Mar. 1997.

[9] S. Malik, M. Martonosi, and Y.-T. S. Li, "Static timing analysis of embedded software," in *Proc. Design Automation Conf.*, June 1997, pp. 147–152.

[10] E3S: The Embedded System Synthesis Benchmarks Suite. [Online] Available: http://www.ee.princeton.edu/cad/projects.html.

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

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

[13] S. Prakash and A. Parker, "SOS: Synthesis of application-specific heterogeneous multiprocessor systems," *J. Parallel Distributed Comput.*, vol. 16, pp. 338–351, Dec. 1992.

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

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

[16] C. Lee, M. Potkonjak, and W. Wolf, "Synthesis of hard real-time application specific systems," *Design Automation Embedded Syst.*, vol. 4, no. 4, pp. 215–242, 1999.

[17] J. Axelsson, "Architecture synthesis and partitioning of real-time systems: A comparison of three heuristic search strategies," in *Proc. Int. Workshop Hardware/Software Co-Design*, Mar. 1997, pp. 161–165.

[18] W. H. Wolf, "An architectural co-synthesis algorithm for distributed, embedded computing systems," *IEEE Trans. VLSI Syst.*, vol. 5, pp. 218–229, June 1997.

[19] B. P. Dave, G. Lakshminarayana, and N. K. Jha, "COSYN: Hardware–software co-synthesis of heterogeneous distributed embedded systems," *IEEE Trans. VLSI Syst.*, vol. 7, pp. 92–104, Mar. 1999.

[20] 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.

[21] 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.

[22] R. P. Dick and N. K. Jha, "MOGAC: A multiobjective genetic algorithm for hardware–software co-synthesis of distributed embedded systems," *IEEE Trans. Computer-Aided Design*, vol. 17, pp. 920–935, Oct. 1998.

[23] R. P. Dick, "Multiobjective synthesis of low-power real-time distributed embedded systems," Ph.D. dissertation, Dept. Elect. Eng., Princeton Univ., Princeton, NJ, July 2002.

[24] J. K. Adams and D. E. Thomas, "The design of mixed hardware/software systems," in *Proc. Design Automation Conf.*, June 1996, pp. 515–520.

[25] R. Ernst, "Codesign of embedded systems: Status and trends," *IEEE Design Test Comput.*, vol. 12, pp. 45–54, Apr. 1998.

[26] L. Garber and D. Sims, "In pursuit of hardware–software codesign," *IEEE Comput.*, vol. 31, pp. 12–14, June 1998.

[27] G. Goossens, J. V. Praet, D. Lanneer, W. Geurts, A. Kifli, C. Liem, and P. G. Paulin, "Embedded software in real-time signal processing systems: Design technologies," *Proc. IEEE*, vol. 85, pp. 436–454, Mar. 1997.

[28] K. G. Shin and P. Ramanathan, "Real-time computing: A new discipline of computer science and engineering," *Proc. IEEE*, vol. 82, pp. 6–23, Jan. 1994.

[29] W. H. Wolf, "Hardware–software co-design of embedded systems," *Proc. IEEE*, vol. 82, pp. 967–989, July 1994.

[30] C. M. Fonseca and P. J. Fleming, "Multiobjective genetic algorithms made easy: Selection, sharing, and mating restrictions," in *Proc. Genetic Algorithms Eng. Syst.: Innovations Applicat.*, Sept. 1995, pp. 45–52.

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

[32] L. Shang and N. K. Jha, "Hardware–software co-synthesis of low power real-time distributed embedded systems with dynamically reconfigurable FPGAs," in *Proc. Int. Conf. VLSI Design*, Jan. 2002, pp. 345–352.

[33] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proc. Design Automation Conf.*, June 1995, pp. 456–461.

[34] B.-D. Rhee, S. L. Min, S.-S. Lim, H. Shin, C. S. Kim, and C. Y. Park, "Issues of advanced architectural features in the design of a timing tool," in *Proc. Workshop Real-Time Oper. Syst. Software*, May 1994, pp. 59–62.

[35] Z. Chen and K. Roy, "A power macromodeling technique based on power sensitivity," in *Proc. Design Automation Conf.*, June 1998, pp. 678–683.

[36] M. Lee, V. Tiwari, S. Malik, and M. Fujita, "Power analysis and minimization techniques for embedded DSP software," *IEEE Trans. VLSI Syst.*, vol. 5, pp. 123–135, Mar. 1997.

[37] V. Tiwari, S. Malik, A. Wolfe, and M. T.-C. Lee, "Instruction level power analysis and optimization of software," *J. VLSI Signal Process.*, vol. 13, no. 2–3, pp. 223–238, 1996.

[38] Xilinx, Inc. (1997, June). *A Simple Method of Estimating Power in XC4000XL/EX/E FPGAs* [Online] Available: http://www.xilinx.com.

[39] A. Raghunathan, N. K. Jha, and S. Dey, *High-level Power Analysis and Optimization*. Boston, MA: Kluwer, 1997.

[40] R. Y. Chen, R. M. Owens, M. J. Irwin, and R. S. Bajwa, "Validation of an architectural level power analysis technique," in *Proc. Design Automation Conf.*, June 1998, pp. 242–245.

[41] S. W. Mahfoud and D. E. Goldberg, "Parallel recombinative simulated annealing: A genetic algorithm," *Parallel Comput.*, vol. 21, pp. 1–28, Jan. 1995.

[42] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[43] Embedded Microprocessor Benchmark Consortium [Online] Available: http://www.eembc.org.

[44] 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.

Robert P. Dick (S'95–M'02) received the B.S. degree in computer engineering from Clarkson University, completing this four-year degree in three years with the highest possible grade in every class, Potsdam, NY, and the Ph.D. degree in electrical engineering from Princeton University, Princeton, NJ, in 2002.

He is currently an Assistant Professor of Electrical and Computer Engineering, Northwestern University, Evanston, IL. He was a Visiting Professor in the Department of Electronic Engineering, Tsinghua University, Beijing, China from 2002 to 2003. He was a Visiting Researcher at NEC Computers and Communication Research Laboratories, Princeton, NJ, in 1999. His publications have focused on the multiobjective optimization and synthesis of embedded systems. In addition, he has published in the areas of real-time operating system power consumption analysis and wireless ad-hoc network protocols. His interests are in the automatic design and dynamic adaptation of computers.

Prof. Dick received the George Van Ness Lothrop Honorific Fellowship from Princeton University. Fellowships in this category are granted to the top 17 fourth-year graduate students at Princeton University. He received the Best Paper Award at PDCS'02 for a paper he coauthored.

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, Stony Brook, in 1982, and the Ph.D. degree in electrical engineering from the University of Illinois, Urbana, IL, in 1985.

He is a Professor of Electrical Engineering at Princeton University, Princeton, NJ. He is the Director of the Center for Embedded System-on-a-Chip Design funded by the New Jersey Commission on Science and Technology. He has coauthored three books, entitled *Testing and Reliable Design of CMOS Circuits* (Norwell, MA: Kluwer, 1990), *High-Level Power Analysis and Optimization* (Norwell, MA: Kluwer, 1998), and Testing of Digital Systems (Cambridge, U.K.: Cambridge Univ. Press, 2003). He has also authored three book chapters. He has authored or coauthored more than 230 technical papers. A paper of his was chosen for inclusion in "The Best of ICCAD: A Collection of the Best IEEE International Conference on Computer-Aided Design Papers of the Past 20 Years." He holds 11 U.S. patents. His research interests include low-power hardware and software design, computer-aided design of integrated circuits and systems, digital system testing, and distributed computing.

Dr. Jha is a Fellow of the ACM. He has served as an Associate Editor of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS II: ANALOG AND DIGITAL SIGNAL PROCEESSING. He is currently serving as an Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, the *Journal of Electronic Testing: Theory and Applications* (JETTA), and the *Journal of Embedded Computing*. He has served as the Guest Editor for the JETTA special issue on high-level test synthesis. He has also served as the Program Chairman of the 1992 Workshop on Fault-Tolerant Parallel and Distributed Systems. He is the recipient of the AT&T Foundation Award, the NEC Preceptorship Award for research excellence, and the NCR Award for teaching excellence. He has coauthored six papers which have won the Best Paper Award at ICCD'93, FTCS'97, ICVLSID'98, DAC'99, PDCS'02, and ICVLSID'03.