MOGAC: A Multiobjective Genetic Algorithm for Hardware-Software Co-Synthesis of Distributed Embedded Systems

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

Abstract—**In this paper, we present a hardware-software co-synthesis system, called MOGAC, that partitions and schedules embedded system specifications consisting of multiple periodic task graphs. MOGAC synthesizes real-time heterogeneous distributed architectures using an adaptive multiobjective genetic algorithm that can escape local minima. Price and power consumption are optimized while hard real-time constraints are met. MOGAC places no limit on the number of hardware or software processing elements in the architectures it synthesizes. Our general model for bus and point-to-point communication links allows a number of link types to be used in an architecture. Application-specific integrated circuits consisting of multiple processing elements are modeled. Heuristics are used to tackle multi-rate systems, as well as systems containing task graphs whose hyperperiods are large relative to their periods. The application of a multiobjective optimization strategy allows a single co-synthesis run to produce multiple designs which trade off different architectural features. Experimental results indicate that MOGAC has advantages over previous work in terms of solution quality and running time.**

Keywords— **Genetic algorithm, hardware-software co-synthesis, low-power synthesis, multiobjective optimization**

I. Introduction

Hardware-software co-design is the process of concurrently defining the hardware and software portions of an embedded system while considering dependencies between the two [1]–[4]. Designers rely on their experience with past systems when estimating the resource requirements of a new system. Since *ad hoc* design exploration is time-consuming, an engineer typically selects a conservative architecture after little experimentation, resulting in an unnecessarily expensive system. Most research in the area of hardware-software co-design has focused on easing the process of design exploration. Automating this process falls within the more specialized realm of co-synthesis. Given an embedded system specification, a co-synthesis system determines the hardware and software processing elements (PEs) needed as well as the communication links to be used. In addition, the system assigns each task to a PE and determines the PEs to which each link is connected. Finally, a schedule is provided for each PE and communication link such that all real-time constraints are met [5], [6]. Co-synthesis systems generate feasible, low-cost architecture descriptions without designer intervention.

Most real-life embedded systems are composed of multiple general-purpose processors and application-specific integrated circuits (ASICs), *i.e.*, they are distributed heterogeneous architectures [1], [7]. Related work in co-synthesis typically assumes a one CPU-one ASIC architecture [7]–[9]. However, many specifications can more efficiently be met by distributed architectures. A practical co-synthesis system cannot limit its design space to one CPU-one ASIC architectures.

Power consumption is often a concern during the design of embedded systems. The demand for portable battery-powered devices is high and likely to increase. It is important to reduce the average power consumption of such systems, thereby increasing battery lifespan [10]. Although early work in low-power electronics focused on changes to fabrication technology and logic design, it has been shown that larger gains can be obtained by considering power during the earlier phases of the design process [11].

In most related work, communication is assumed to have only one associated cost: time. However, communication links consume power as well as time. A co-synthesis system which targets low-power applications must take both PE and communication link power requirements into account. Many co-synthesis systems unrealistically simplify or omit communication link synthesis altogether. This stems from the one CPU-one ASIC assumption. Many real distributed embedded systems are composed of numerous PEs and there are several types of communication links available for connecting them. For a given system, a low-price and low-power feasible communication network may be composed of multiple busses and point-to-point links. A practical co-synthesis system must be capable of automatically generating a low-price, low-power, heterogeneous communication network.

There are four tasks which must be carried out by a co-synthesis system:

- **Allocation:** Determine the quantity of each type of PE and communication link to use.
- **Assignment:** Select a PE to execute each task upon. Choose a link to use for each communication event.
- **Scheduling:** Determine the time at which each task

The authors are with Princeton University, Princeton NJ 08544 USA. This work was supported in part by an NSF Graduate Fellowship and in part by NSF under Grant No. MIP-9423574.

and communication event occurs.

- **Performance evaluation:** Compute the price, speed, and power consumption of the solution.

Optimal co-synthesis is an intractable problem. Allocation/assignment and scheduling are each known to be NP-complete for distributed systems [12]. It is, therefore, not surprising that all co-synthesis systems which rely on optimal mixed integer linear programming [13], [14] and exhaustive exploration [15] can only be applied to small instances of the co-synthesis problem. Heuristics have seen some success with larger instances of the distributed system co-synthesis problem. These are discussed next.

Iterative improvement algorithms start with a complete, but sub-optimal, solution and make local changes to it while monitoring the solution's cost. Algorithms in this class are prone to becoming trapped in local minima. Although they have reasonable run-times, the results produced for systems of high complexity tend to be sub-optimal [5], [16], [17]. Tabu search is a form of iterative improvement in which past changes are remembered by the algorithm and used to guide future changes. This type of algorithm has been applied to the hardware-software partitioning problem [18] and architecture synthesis problem [19].

Constructive algorithms build a system by incrementally adding components. To be computationally tractable, a constructive algorithm must make changes with global impact while inspecting only the local effects of these changes. This often leads to an accumulation of sub-optimal decisions, especially when large systems are constructed. Despite their susceptibility to becoming trapped in local minima, constructive algorithms are capable of producing high quality results [20], [21]. However, in Srinivasan's work, power consumption is not taken into account, the communication model is simplistic, and multi-rate systems are not efficiently handled [20]. COSYN was the first co-synthesis system to take power consumption into account [21]. Communication links are modeled and heuristics are used to tackle multi-rate systems. Although fast, COSYN suffers from an inability to do true multiobjective optimization.

Two types of probabilistic optimization algorithms have been applied to the co-synthesis problem: simulated annealing algorithms and genetic algorithms. Simulated annealing algorithms are capable of escaping local minima of arbitrary depth [22]. These algorithms are a strict superset of greedy iterative improvement algorithms; randomized improvement is not necessarily direction-less improvement. Simulated annealing algorithms have been successfully used to partition hardware-software systems [18], [19], [23].

In general, genetic algorithms share simulated annealing algorithms' ability to escape local minima, but they offer other advantages as well. Genetic algorithms allow solutions to cooperatively share information with each other. They are capable of true multiobjective optimization, exploring the set of solutions which can only be improved in one way by being degraded in another (the *Pareto-optimal* set), instead of collapsing all costs into one with a weighted sum, as is the case for most other probabilistic optimization algorithms [24]–[26]. Saha's exploratory work demonstrates that genetic algorithms can be applied to the hardware-software partitioning problem [27]. A number of simplifying assumptions are made in Saha's system, however. Only one software processor is allowed, there are no provisions for synthesizing systems with multi-rate periodic task graphs, and a limited communication link model is used. Their genetic algorithm only optimizes one variable: price. Axelsson's system, similarly, optimizes only price and does not carry out communication link synthesis [19].

Teich *et al.* applied a multiobjective genetic algorithm to the heterogeneous distributed system co-synthesis problem [28]. Their approach does not target systems with hard real-time constraints. Power consumption is ignored. Multi-rate systems, and systems containing task graphs with periods less than their deadlines, are not handled. They use a method of crossover which randomly selects bits to swap and does not attempt to preserve sequences of bits describing related attributes. This approach does not preserve locality (see Section II-A for information on the importance of locality). In general, this results in an $\mathcal{O}\left(n^2\right)$ slowdown in the rate at which solutions are implicitly evaluated, when compared to an optimal locality-preserving crossover [29, pp. 40, 41]. In this work, solutions which are not valid and which cannot be made valid by the application of a repair operator are immediately terminated. Multiobjective optimization is not performed. Their experimental results consist of one small example and no comparisons are made with other co-synthesis systems.

MOGAC synthesizes distributed heterogeneous embedded systems. Price and power consumption are optimized under a number of hard constraints. MOGAC uses a communication model that is capable of synthesizing systems with multiple busses and point-to-point communication links. ASICs consisting of multiple PEs are modeled. MOGAC applies heuristics which allow multi-rate systems to be scheduled in reasonable time even when the least common multiple (LCM) scheduling method [30] would otherwise require a large number of task graph copies to be made. MOGAC's use of a multiobjective genetic algorithm allows it to provide a designer with multiple solutions which trade off different system costs.

The paper is organized as follows. In Section II, we present preliminary concepts and definitions. In Section III, we describe the algorithms employed by MOGAC. We give the experimental results in Sec-

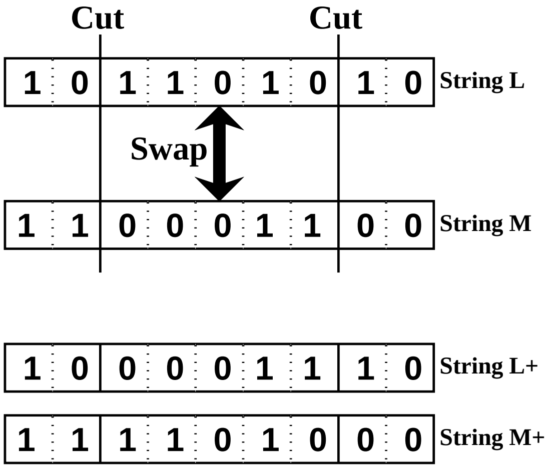

Fig. 1. Crossover.

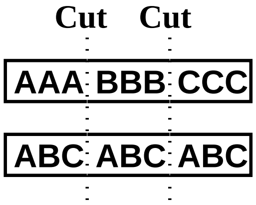

Fig. 2. String locality.

tion IV. We conclude with Section V.

II. Preliminaries

In this section, we present preliminary concepts used in genetic algorithms and co-synthesis algorithms.

A. Genetic Algorithms

Genetic algorithms maintain a pool of solutions which evolve in parallel over time. Genetic operators are applied to the solutions in the current pool to improve the solutions. The lowest quality solutions are then removed from the pool [29]. A cost is a variable that a genetic algorithm attempts to minimize, *e.g.*, price and power consumption. Genetic algorithms excel at simultaneously optimizing multiple conflicting costs. They have the ability to escape local minima and communicate information among solutions.

Next, we define some basic terms used to discuss genetic algorithms. In a conventional genetic algorithm, every solution is represented by an array, or *string*, of values. Although we discuss the genetic algorithm used by MOGAC in conventional terms, each solution is represented by a collection of strings and no primitive strings are ever computed. As discussed in Section II-C, the strings used in MOGAC are more intricate than those used in conventional genetic algorithms. Genetic operators are applied directly to the complex data structures which represent a solution. Such algorithms are sometimes called evolutionary algorithms. Although operating on unconventional strings increases the complexity of a genetic algorithm, sometimes this is the least complicated option available. MOGAC needs to maintain and modify a great deal of hierarchical information about its solutions. It is simpler, and faster, to operate on the information directly than to carry out conversion into a conventional string each time it is necessary to modify a string.

In conventional genetic algorithms, as well as in MOGAC, all changes to strings are brought about by three operators. *Reproduction* makes a copy of a solution. *Mutation* randomly changes part of a solution's description. *Crossover* swaps portions of different solutions. Fig. 1 shows an example of string crossover. In this illustration, crossover occurs between string **L** and string **M**. Two cuts are made and the portions of **L** and **M** between these cuts are swapped, producing the strings **L+** and **M+**. Crossover is the operator that gives genetic algorithms their strength; it allows different solutions to share information with each other.

Some genetic algorithms are capable of varying the probability of allowing a solution to be replaced by one of lower quality. Such an algorithm can be viewed as a generalized simulated annealing algorithm [31]. However, unlike a classical simulated annealing algorithm, this sort of genetic algorithm simultaneously operates on multiple solutions which share information with each other. The genetic algorithm employed by MOGAC shares the strengths of classical genetic algorithms and simulated annealing algorithms, and is capable of running as a simulated annealing algorithm or an iterative improvement algorithm, as well as a genetic algorithm. MOGAC has produced its highest quality results in the least amount of time when run in the genetic algorithm mode. The other modes were implemented for experimental purposes.

It is vital that the string encoding used to represent a solution maintain locality [29]. Features of a solution that depend closely on each other should be located near each other in a string and relatively independent features should be far apart. The reason for this requirement is most easily illustrated with an example.

In Fig. 2, **A**, **B**, and **C** represent variables associated with different features of a solution. As strings crossover with each other, they are cut into sections. The encoding of each feature is spread across the bottom string; information about a feature is likely to be split into separate solutions when crossover occurs. The feature encoding in the top string, however, is localized; information about a feature will probably remain in one string when crossover occurs. If a solution has discovered a good way of optimizing some feature of a problem, it is important for the encoding of that feature to remain intact. The practical effect of using a string encoding method and crossover method which maintain locality is that the genetic algorithm takes advantage of *implicit parallelism*, *i.e.*, n function evaluations im-

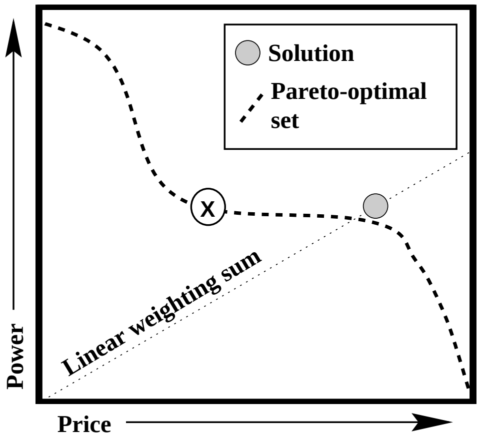

Fig. 3. Weighted sum cost function.

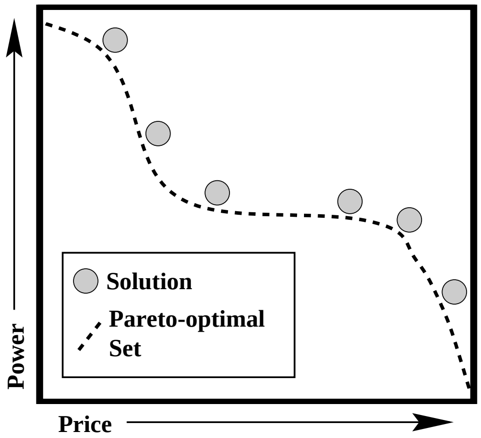

Fig. 5. True multiobjective optimization.

plicitly examine approximately n^3 string configurations [29].

B. Multiobjective Optimization

The co-synthesis problem is inherently one of multiobjective optimization. There are numerous costs and improving one cost of a system often results in the degradation of another. Most past co-synthesis systems have dealt with this optimization problem by using a linear weighted sum to collapse all the system costs into one variable and optimizing this variable. For this method to be successful, the weighting array used must be appropriate for the problem instance as well as the designer's desired solution. Unfortunately, the co-synthesis problem is too complicated for an instance's best weighting array to be known without first exploring that instance's Pareto-optimal set of solutions, *i.e.*, those solutions which can only be improved in one area by being degraded in another. It is impossible, however, to explore the Pareto-optimal set of solutions if an arbitrary weighting array has been used to collapse all costs into a single value.

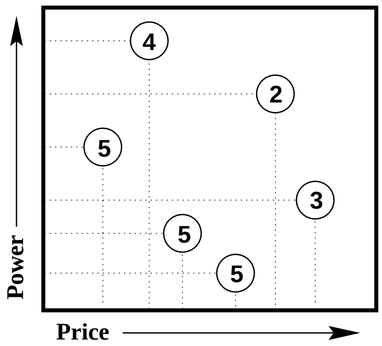

Fig. 4. Pareto-rank.

Assume a designer is trying to optimize two conflicting features of a system: price and power consumption. If the designer uses a conventional optimization algorithm which can only deal with one cost function, it is necessary to collapse the two costs into one value. Although an apparently reasonable weighting array can be selected, the designer has no way of knowing the shape of the Pareto-optimal curve ahead of time. In Fig. 3, **X** marks the designer's preferred solution. For a miniscule power consumption penalty, the price of the system can be significantly decreased. Unfortunately, the designer will never know that a valid solution exists at **X** because the weighting array prevents this portion of the Pareto-optimal curve from being explored. Although the limitations of single objective optimization can be seen from this simple example, the problem of selecting an appropriate weighting array becomes even more severe as the number of costs in a system increases.

A solution *dominates* another if all of its features are better. A solution's *Pareto-rank* is the number of other solutions, in the solution pool, which do not dominate it. Calculating Pareto-rank is an $\mathcal{O}\left(solution\_pool\_size^2\right)$ operation; each solution must be compared with every other solution. In Fig. 4, each circle represents a solution. Each solution's price and 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.

At the end of a multiobjective genetic algorithm's run, the designer is presented with a number of non-inferior solutions (see Fig. 5). These solutions are not dominated by any other solutions. This approach yields Fig. 3's solution **X**. Although the non-inferior solutions are not guaranteed to be the Pareto-optimal set of solutions for the problem instance (the heterogeneous distributed system co-synthesis problem contains multiple NP-complete problems, each of which would require multiple solutions), they do form an upper bound on the Pareto-optimal set, giving the designer insight into the shape of the problem's Pareto-optimal solution set. The trade-offs available between solution costs in these non-inferior solutions are made clear.

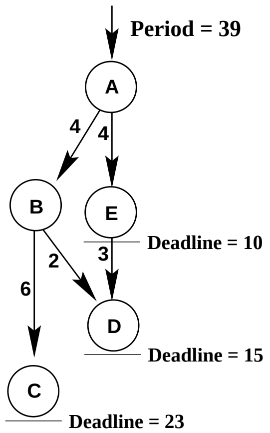

Fig. 6. Task graph.

C. Embedded System Model

MOGAC operates on an embedded system specification that defines a set of requirements which must be met, and a set of resources which can be used to fulfill those requirements. In this subsection, we provide high-level descriptions of the specifications MOGAC accepts.

Task graph

Task graphs specify some of the requirements a designer places upon an embedded system. A task graph, as shown in Fig. 6, is a directed acyclic graph in which each node is associated with a task and each edge is associated with a scalar describing the amount of data that must be transferred between the two connected tasks. Each task may only begin executing after all of its data dependencies have been satisfied. Thus, in Fig. 6, task **D** may only begin execution after tasks **B** and **E** have each completed execution and transferred two and three units of data, respectively, to task **D**.

MOGAC places no restrictions on the granularity of task graphs. However, co-synthesis research generally assumes coarse-grained tasks, *i.e.*, each task is complicated enough to require numerous microprocessor instructions. The *period* of a task graph is the amount of time between the earliest start times of its consecutive executions. A node with no outgoing edges is called a *sink* node. A *deadline*, the time by which the task associated with the node must complete its execution, exists for every sink node. However, other nodes may also have deadlines associated with them. The deadline of a task graph is the maximum of all the deadlines specified in it. An embedded system specification may contain multiple task graphs, each of which may have a different period.

Processing element

A PE executes tasks. Two sorts of PEs are modeled: *cores* and *processors*. Processors represent general-purpose processors which can only execute one task at a time. Multiple cores may be located on the same integrated circuit (IC), upon which multiple tasks may execute simultaneously. This provides a model for ASICs which are capable of carrying out different tasks at the same time. The relationship between processors, cores, and ICs is shown in Fig. 7. PEs can be of various types, *e.g.*, an MC68000 is a PE type. A solution may contain more than one instance of a given type of resource, *e.g.*, a solution may contain more than one PE instance of the type MC68000.

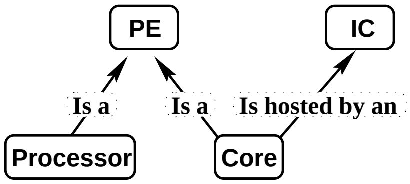

Fig. 7. PE-IC hierarchy.

MOGAC accepts a database which specifies the performance of each task on each available PE type, as well as providing other information about the PEs available, *e.g.*, a list of tasks which are incompatible with each type of PE, the price of each resource, and the number of devices provided by ICs and consumed by cores. Worst-case execution time and power consumption values for tasks on a given PE type can be obtained by direct measurement or simulation. Characterizing a PE database in this manner requires that the designer know the input vectors which elicit worst-case execution time and power consumption for each task-PE pair. Another option is to use worst-case performance analysis tools to determine an upper-bound on execution time or power consumption, without requiring a specific input vector [32]–[34].

The following information establishes the relationships between tasks and processors:

- A two-dimensional array indicating the worst-case execution time of each task on each processor.
- A two-dimensional array indicating the average power consumption of each task on each processor.

In addition to these arrays, processors have price and idle power consumption values.

The following information establishes the relationship between tasks and cores:

- A two-dimensional array indicating the relative worst-case execution time of each task on each core.
- A two-dimensional array indicating the relative average power consumption of each task on each core.
- A two-dimensional array indicating the peak power consumption of each task on each core.

Cores do not have an inherent price. However, each core is assigned to an IC which does have a price. The following variables are associated with ICs: price, device count, pins available, idle power consumption, peak power dissipation, speed, and power efficiency. Each core places a device count requirement, *e.g.*, number of transistors or configurable logic blocks (CLBs), on the IC to which it is assigned. For an architecture to be valid, each IC must meet device count requirements of the cores assigned to it and the pin count requirements of the communication links attached to it. In addition, each IC must meet the peak power dissipation requirements of the tasks assigned to the cores implemented on it. Tasks do not have pin count, device count, or peak power dissipation requirements. However, tasks may be carried out by cores, which place such requirements on their host ICs.

The worst-case execution time for a task assigned to a core is equivalent to its relative worst-case execution time divided by the speed of the IC on which the core is implemented. The task's average power consumption is its relative average power consumption divided by the power efficiency of the IC on which the task's core is implemented. Thus, in the current implementation of the algorithm, it is assumed that there is a linear relationship between core worst-case execution time and core relative worst-case execution time. Similarly, there is a linear relationship between core average power consumption and core relative average power consumption. This model could trivially be generalized to use a full lookup table approach (this is how the task execution time of a task on any given PE is determined).

Communication link

Communication links have the following attributes: packet size, average power consumption per packet, worst-case communication time per packet, price, number of contacts, pin requirement, and idle power consumption. Each task graph edge must be assigned to a communication link. The worst-case communication time and average power consumption of an edge are linearly dependent on the number of packets of data transferred through its link. The number of contacts a link supports is the number of ICs it can connect, *i.e.*, a link with two contacts is a point-to-point link. A link with more than two contacts is a bus. There may be more than one communication link connected to a PE instance. Pin requirement is the number of pins on an IC required to support the use of the communication link. In previous distributed computing work, it is commonly assumed that communication between tasks which are assigned to the same IC consumes an insignificant amount of time and power. We also make this assumption. If an architecture contains two communicating tasks which execute on separate ICs, the architecture is invalid if there are no communication links connecting the ICs.

III. Algorithm Description

In this section, we give a description of the algorithms used in MOGAC. We begin, in Section III-A, with an overview of the algorithm. This is followed,

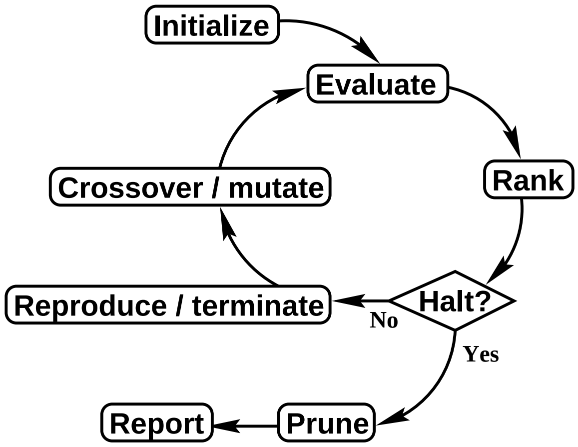

Fig. 8. MOGAC overview.

in Section III-B, by an explanation of solution clusters. Section III-C describes MOGAC's performance evaluation algorithms. Solution reproduction, mutation, and crossover are described in Section s III-D and III-E.

A. *Overview of MOGAC*

In this subsection, we give an overview of MOGAC's primary algorithm. MOGAC maintains a pool of solutions which evolve in parallel. Fig. 8 illustrates MOGAC's core algorithm. After initializing each solution with simple randomized algorithms, MOGAC enters a loop which repeats until the halting condition, the passage of a number of generations without improvement in the solution pool, is met. Each time the loop completes, a *generation* has passed.

After initialization, MOGAC evaluates each of its solutions. During evaluation, a solution's costs, *e.g.*, price and power consumption, are determined. The costs are then compared to the designer-supplied constraints to determine how severely the constraints are violated. At this point, the solutions are ranked using the multiobjective criterion described in Section II-B. If the halting conditions have not yet been reached, low-rank solutions are terminated and high-rank solutions reproduce to take their places. The newly born solutions are then modified via crossover and mutation. At this point, the generation has completed and another begins. Eventually, enough generations pass without improvement in the solution pool to trigger the halting condition. Before halting, MOGAC prunes any invalid and inferior solutions from its solution pool and presents the remaining solutions to the designer.

B. *Clusters*

In this subsection, we introduce the six strings which describe solutions in MOGAC and explain how clusters of solutions are used to prevent crossover from producing *structurally incorrect* solutions, *i.e.*, solutions which are physically impossible. Every solution in MOGAC is defined by a collection of six strings. The *PE allocation*

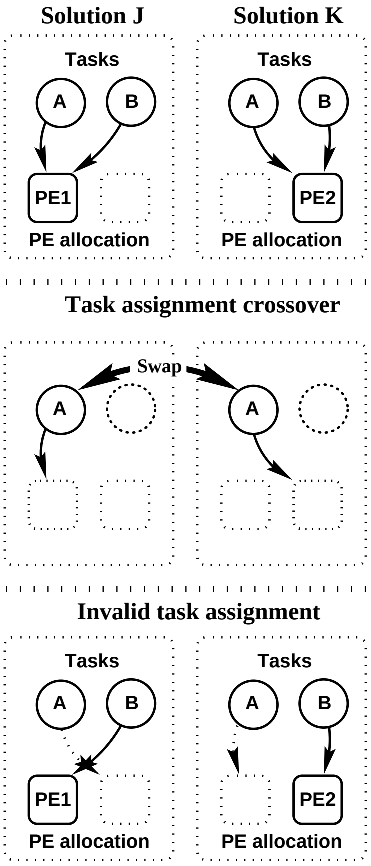

Fig. 9. Bad crossover.

string, *IC allocation string*, and *link allocation string* record the number and types of PEs, ICs and communication links present in a solution. The *task assignment string* records the PE instances used to carry out each task. The *core assignment string* records the IC used to host each core. The *link connectivity string* records the PE instances to which each link is connected. Formal definitions of these strings are given in Section III-E.

If it were possible for solutions to indiscriminately crossover with each other, structurally incorrect solutions would sometimes be produced. Assume the existence of two solutions: **J** and **K**. As illustrated in Fig. 9, **J**'s PE allocation contains only one PE instance, of type **PE1**. **K**'s PE allocation contains only one PE instance, of type **PE2**. Therefore, all tasks in **J** are assigned to the PE of type **PE1** and all tasks in **K** are assigned to the PE of type **PE2**. If a crossover were to occur between the task assignment strings in the two solutions, the result would be the existence of some tasks in **J** which are assigned to a PE of type **PE2**. However, no PEs of type **PE2** exist in **J**. Similar problems are caused by the indiscriminate crossover of other types of strings.

It would be possible to detect structurally incorrect solutions and repair, or immediately terminate, them. However, examining every solution and modifying or terminating those which are structurally incorrect would be costly in terms of computation time. More

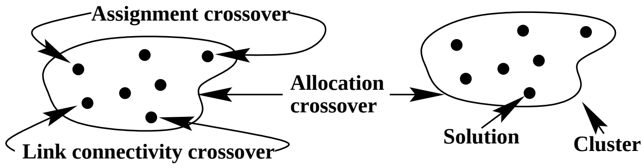

Fig. 10. Solution clusters.

importantly, the post-processing would destroy the locality of the crossover operator, *i.e.*, this step would disrupt the partial solutions which were swapped during crossover.

MOGAC uses the concept of solution clusters to prevent structurally incorrect solutions from being created in the first place. As shown in Fig. 10, solutions are grouped into clusters. Solutions within a cluster all share the same PE allocation, IC allocation, and link allocation strings. Thus, each solution in the single cluster has the same PE and communication link resources available to it. However, the task assignment, core assignment, and link connectivity strings of solutions in the same cluster may differ. Crossover of assignment and link connectivity strings occurs between solutions in the same cluster. Mutation of these strings can be applied to individual solutions. Solutions resulting from these operations are guaranteed to be structurally correct. Crossover of allocation strings occurs between entire clusters, destroying the solutions within the clusters. Similarly, when one of a cluster's allocation strings mutate, each of the solutions within the cluster are updated so that they share the cluster's new allocation string. Inter-cluster crossover and mutation of allocation strings occurs less frequently than intra-cluster crossover and mutation. Every time crossover or mutation are applied to clusters, instead of individual solutions, the information contained in the assignment and link connectivity strings of the involved solutions is no longer valid. These strings are, therefore, re-initialized.

There are three advantages to the use of solution clusters. The overall algorithm is simplified because it is not necessary to detect or repair structurally incorrect solutions. The algorithm's execution time is decreased because it is not necessary to deal with structurally incorrect solutions and because locality is not destroyed by repair operations, thus allowing more implicit parallelism in the genetic algorithm (see Section II-A). Finally, using clusters makes MOGAC a parallel algorithm. There is no need for solutions in different clusters to communicate with each other except during the infrequent application of inter-cluster crossover.

C. Solution Evaluation

Performance evaluation consists of calculating a solution's costs and determining how severely they violate the constraints imposed by the designer. If one of the system's costs is higher than its *hard constraint*, the system is invalid. For example, the schedule length of a task graph cannot exceed its hard real-time constraint. Valid systems may have costs which are higher than their *soft constraints*, although it is desirable to reduce a cost until it is lower than its soft constraint. In this subsection, we will explain how MOGAC does performance evaluation and then describe the process by which raw performance metrics are converted into hard and soft constraint violation values.

Scheduling

The PE allocations, IC allocations, link allocations, task assignment, core assignment, and link connectivities of MOGAC's solutions are derived from their strings. Scheduling, however, is carried out by a conventional algorithm before each solution evaluation. MOGAC uses a slack-based list scheduling algorithm to generate static PE and communication link schedules. Static scheduling makes it possible to guarantee that hard real-time constraints will be met [35]. In the current implementation, a non-preemptive schedule is generated. Although there are advantages to allowing preemption in coarse-grained scheduling problems, a non-preemptive scheduler was sufficient to allow MOGAC to meet or beat results from the literature (see Section IV). The advantages of preemptive scheduling are partially offset by a practical weakness. In general, preemption results in context-switching penalties which are costly in terms of power consumption [36]. MOGAC's scheduling algorithm assigns a priority to a task based upon the difference between its latest possible start time and its earliest possible start time. The relative priorities of tasks in different task graphs, as well as different copies of the same task graph, are based on the periods and deadlines of the different graphs. The scheduler is capable of dealing with embedded system specifications in which task graphs have periods less than their deadlines.

The *hyperperiod* is the LCM of all the task graph periods in a multi-rate system specification. Co-synthesis systems which use a straightforward application of the LCM scheduling method [30] are forced to repeatedly schedule each task graph until the hyperperiod of the system has elapsed. This can be computationally expensive for systems in which the hyperperiod is large, relative to the periods of individual task graphs. MOGAC uses heuristics to tackle system specifications with a large hyperperiod. One of these is an extension of a method used in real-time computing [37]. The problem caused by a large hyperperiod can be reduced by tightening the periods of some task graphs. Consider a system consisting of two periodic task graphs, where the first has a period of 12, and the second has a period of 13. The hyperperiod is, therefore, 156. If we tighten the period of the second task graph to 12, however, the system's hyperperiod reduces to 12. The designer has

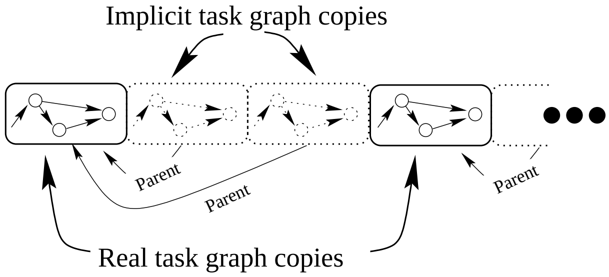

Fig. 11. Task graph copies.

full control over the aggressiveness with which the hyperperiod contraction heuristic is applied. MOGAC allows the designer to specify the maximum and minimum acceptable periods for each task graph in the system. Subject to these constraints, a period for each task graph is calculated such that the number of task graph copies needed for LCM scheduling is minimized.

We have developed a method in which some of the task graph copies in the hyperperiod are *implicit* and some are *real* (see Fig. 11). Each implicit copy has a real *parent*. Implicit copies are not entered in a solution's task assignment string; they share the assignment strings of their parents. Although it is necessary to schedule implicit task graph copies, there is no need to prioritize the nodes of these copies; the implicit task graph node priorities are equivalent to the parent task graph node priorities. Additionally, the absence of implicit copies from a solution's task assignment string reduces the size of the genetic algorithm's solution space, thus speeding optimization. Selecting a ratio of the number of real task graph copies to the total number of task graph copies involves making a trade-off between potential solution quality and MOGAC's run-time. This decision is left to the designer. For the examples in Section IV, a low ratio (< 0.2) rapidly produced high-quality results.

Cost calculation

System price, task graph completion time, and system power consumption are computed during cost calculation. System price is determined by taking the sum of the prices of all ICs, processors, and links in the allocation strings. The completion time of each node in a task graph is recorded during scheduling. Therefore, the completion times of all nodes with deadlines are available for inspection. All schedules span the system's hyperperiod. System power consumption is computed by stepping through each PE and link's hyperperiod schedule, obtaining the system energy required (this includes the idle PE/link energy), and dividing the energy by the hyperperiod [21].

Constraint violation

A system's constraint violations are derived from its costs and the constraints imposed by the designer. Solutions have a number of hard constraints. Although solutions in which one or more hard constraints have been violated are invalid, MOGAC treats them no differently than other solutions during its run. Solutions which violate their hard constraints are removed only at the end of a co-synthesis run. It may seem counter-intuitive to allow invalid solutions to survive. However, doing so is beneficial when solving constrained problems [38], for there are significant disadvantages associated with the alternatives. If one terminates invalid solutions immediately, one wastes a significant amount of computation time in identifying such solutions. The solutions most likely to eventually evolve into high-quality valid solutions are those that are near the boundary between valid and invalid. By immediately terminating all invalid solutions in each generation, one destroys many solutions which are likely to ultimately evolve into high-quality valid solutions. One could, instead, attempt to repair invalid solutions. However, it is, in general difficult to formulate a repair operation which is guaranteed to repair all solutions [28]. Thus, one will often be forced to terminate solutions even after expending computation time attempting to repair them. More importantly, a repair operation applied to a solution which was made invalid by crossover disrupts a portion of that solution, effectively changing the crossover operator such that it no longer preserves locality. These problems are analogous to the problem with terminating or repairing invalid solutions discussed in Section III-B.

Each system specification has price and average power consumption soft constraints. Typically, the desired price is set to 0. Thus,

$$\mathit{price\_violation} = \max(0,\ \mathit{price} - \mathit{desired\_price})$$

A system's average power violation is calculated in a similar manner.

Every task graph has one or more nodes with specified deadlines. A system's hard real-time constraint violation is the sum of the time constraint violations of all such nodes in all the real and implicit task graph copies in the system. For every IC, the peak power dissipation and device count requirements of all the cores assigned to that IC are summed. Similarly, the pin count requirement placed on an IC by all of the communication links attached to it are summed. When an IC is not capable of meeting the requirements of the cores assigned to it or communication links connected to it, the appropriate hard constraint violations in the solution are increased.

D. Ranking and Reproduction

In this subsection, we explain the manner in which solutions and clusters are selected for reproduction. The number of clusters and solutions maintained by MOGAC is conserved during one run of the algorithm. For each cluster or solution created via reproduction, another is terminated. 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.

$gauss$ is a Gaussian random variable.
$gauss$'s mean = 0 and variance = 1.

$new\_solutions$ is the number of solutions to be replaced via reproduction.

Sort solutions in the order of increasing Pareto-rank.

For $index := 0$ to $new\_solutions - 1$:
  Select a random instance, g, from $gauss$.

  Set $offset := maximum\_index - g / solution\_selection\_elitism$.

  Set $solution[index] := solution[offset]$.

Fig. 12. Solution reproduction algorithm.

Solution ranking and reproduction

Solutions within a cluster are ranked using the method presented in Section II-B. In each generation, a pre-specified number of solutions within each cluster are eliminated to make space for the reproduction of other solutions. MOGAC maintains a variable called $solution\_selection\_elitism$ which controls the probability of high-rank solutions being selected for reproduction. This variable increases during the run of the algorithm. The practical effect of this feature is to allow MOGAC to easily escape local minima during the start of a run. Near the end of a run, however, MOGAC becomes greedier to allow its solutions to converge on local minima. Solutions are selected for reproduction by indexing inward from the highest-ranking solution with a Gaussian random variable whose variance is the inverse of the $solution\_selection\_elitism$. The pseudocode for MOGAC's reproduction algorithm is shown in Fig. 12.

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

Cluster ranking and reproduction

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, in a straightforward way, to take partial domination into account. Cluster domination is represented by a scalar instead of a Boolean value. The definition of rank must also be adjusted when it is applied to clusters. Let x and y be clusters. $nis\,(x)$ is the set of non-inferior solutions in x. $dom\,(a,b)$ is 1 if a is not dominated by b and 0 otherwise. Then,

$$clust\_domination\,(x,y) = \max_{a\in nis(x)} \sum_{b\in nis(y)} dom\,(a,b)$$

and,

$$rank[x] = \sum_{y\in {\scriptstyle set\ of \atop clusters} \wedge\, y\neq x} clust\_domination\,(x,y)$$

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.

E. Evolution

In this subsection, we formally define the six strings which describe each solution. These strings were introduced in Section III-B. In addition, we explain how these strings are modified to allow solutions to evolve.

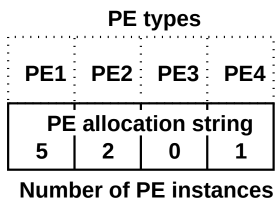

Fig. 13. Example PE allocation string.

Allocation Strings

The PE allocation string, IC allocation string, and link allocation string are arrays of integers. Each integer represents the number of instances of a single type of PE, IC, or link present in a solution. An example PE allocation string is shown in Fig. 13. This example string indicates that there are five instances of type **PE1**, two instances of type **PE2**, zero instances of type **PE3**, and one instance of type **PE4**, in the solution. As mentioned before, for a genetic algorithm to function properly, it is important for its strings to preserve locality, *i.e.*, related entries must be located closer to each other in a string than disparate entries [29]. The allocation string ordering algorithm places PEs such that those with similar characteristics, *e.g.*, price, have a higher probability of being located close together in the string than those with disparate characteristics. The order of PE types in the PE allocation string is determined in the following way.

As mentioned in Section II-C, the relationship between tasks and PEs is defined by a collection of two-dimensional arrays. For the purpose of characterizing a PE type, the one-dimensional arrays corresponding to that PE type are selected from these two-dimensional arrays. Thus, each PE can be characterized by a collection of one-dimensional arrays and some scalars. The first step in determining the order of PE types in the PE allocation string is to collapse each PE type's arrays into scalars. This conversion is done by taking a sum of each array's entries and weighting each entry with the number of tasks, of the type corresponding to that entry's position, which exist in the embedded system specification. After this step, each PE is described by a collection of scalars, *i.e.*, a vector. Imposing a linear locality-preserving cycle on a set of n-dimensional vectors is equivalent to the Traveling Salesman problem. This problem is NP-complete. An approximation algorithm is used to impose an order on these vectors which, in general, places vectors, which are close to each other in the n-dimensional space, close together in the PE allocation string. The *link allocation string* and *IC allocation string* are similar to the PE allocation string and they are ordered using similar algorithms.

```
Start from an empty PE allocation.
For each task t:
    If there exist no PEs capable of executing t:
        Randomly select a PE type, pe, which
          is capable of executing t.

        Add an instance of pe to the solution's
          PE allocation string.
```

Fig. 14. PE allocation string initialization.

PE allocation strings are initialized with the simple constructive algorithm shown in Fig. 14. If the solution contains any cores, its IC allocation string is initialized to contain a single, randomly chosen IC. Otherwise, the IC allocation string is initially empty. Initially, a solution's link allocation string is empty, *i.e.*, the solution contains no links. Links are introduced by subsequent mutations. The intention of these initialization algorithms is to set up minimal valid solutions which will be improved via mutation and crossover.

An allocation string's mutation operator selects a PE, IC, or link type at random; each PE, IC, or link type has the same probability of being selected. The number of instances of the selected PE, IC, or link type is either incremented or decremented, with equal probability. When the crossover operator is applied to two allocation strings, the strings are cut at the same two random offsets and the portions between the cuts are swapped. Af-

Randomly select a task instance, t, in the task assignment string.
pe is the position, in the allocation string, of the PE to which t is assigned.
g is a Gaussian random variable with mean = 0 and variance = 1.
Set $pe^+ := \lceil g \cdot PE\_aggressiveness + pe \rceil$.

If there are no PEs of type pe^+ allocated or t may not execute on pe^+:
    Select the nearest neighbor of pe^+ for which PEs exist and upon which t may execute.

Fig. 16. Task assignment string mutation.

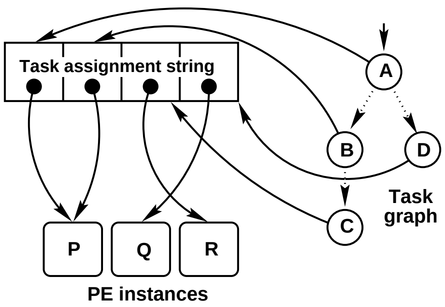

Fig. 15. Example task assignment string.

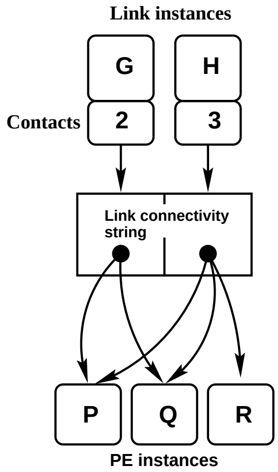

Fig. 17. Example link connectivity string.

ter the crossover or mutation of a PE allocation string, the constructive algorithm shown in Fig. 14 is applied to the participating string. This enforces the condition that, for each task, there exists at least one PE capable of executing it. Usually, it is not necessary for this post-processing step to make any changes to the PE allocation string. Similarly, if a crossover or mutation causes a solution which contains one or more cores to be without ICs, a single, randomly selected, IC is introduced.

Assignment Strings

The task assignment string is an array of PE instance references specifying the PE to which each task is assigned. An example task assignment string is shown in Fig. 15 (see Section II-C for more information about task graphs). In this example, the task assignment string indicates that task **A** is assigned to PE instance **P**, **B** to **P**, **C** to **R**, and **D** to **Q**. Task assignment strings are ordered by conducting a depth-first traversal of all the task graphs in the system specification and concatenating the results. This ordering algorithm makes it more probable that tasks which are located close together, along paths through task graphs, will be located close together in the task assignment string. The core assignment string is an ordered string of IC instance references specifying the IC to which each core is assigned. It is ordered using an algorithm similar to that applied to the PE allocation string.

Initially, each task is randomly assigned to a PE instance in the PE allocation string which is capable of executing it. The constructive algorithm used to initialize a solution's PE allocation string guarantees that there is at least one PE capable of executing each task (see Fig. 14). Similarly, each core in the core assignment string is randomly assigned to an IC.

The task assignment string mutation operator selects a task at random and changes the PE type used to carry out that task (see Fig. 16). An analogous algorithm is used for the mutation of core assignment strings. MOGAC maintains a variable called $PE\_aggressiveness$, which decreases during the run of the algorithm. If the value of this variable is small, a nearby PE type will probably be used to carry out the task. If $PE\_aggressiveness$ is large, it is likely that pe^+ will be far from pe in the PE allocation string. The PE allocation string is ordered in a locality preserving way. Hence, there is an inverse correlation between distance on the PE allocation string and PE type similarity. Decreasing $PE\_aggressiveness$ during a run allows MOGAC to initially mutate task assignment strings in a way that is likely to cause large jumps across the solution space. As a run nears its end, task assignment mutation makes only small changes to the task assignment string, fine-tuning it.

When the crossover operator is applied to two task assignment strings, the strings are cut at the same random offset and the portions following the cut are swapped. The two participating strings always come from solutions which have the same PE allocations because task assignment string crossover is an intra-cluster genetic operator. The mutation operation for core assignment strings is analogous.

The *link connectivity string* is an array of IC and processor instance references specifying the ICs and processors to which each communication link is connected.

Generate an array, pe, of PE locations.
Set each PE location in pe to a unique location in the PE allocation string.

For each array, $link$, of PE references in the link connectivity string:
 Randomize the order of the entries in pe.

 $connect$ is the number of PEs to which $link$ may connect.

 For $index := 0$ to $\min(connect - 1, length[pe])$:
  Set $link[index] := pe[index]$.

Fig. 18. Link connectivity string initialization.

An example link connectivity string is shown in Fig. 17. In this illustration, link **G**'s two contacts are connected to PE instances **P** and **Q**. Link **H** connects **P**, **Q**, and **R**. More than one link may be connected to the same PE instance. In Fig. 17, PE instance **Q** is an example of a PE connected to two communication links. The order of link types in the link connectivity string is equivalent to their order in the link allocation string.

Link Connectivity Strings

Initially, each link is randomly connected to PEs in the PE allocation string (see Fig. 18). The link connectivity mutation operator selects a location in the string at random and applies the inner loop of the initialization algorithm shown in Fig. 18 to it. In other words, it connects a link to PEs randomly. The link connectivity string's crossover operator cuts the participating strings at the same random offset and swaps the portions following the cut. The two participating strings always come from solutions which have the same link allocations because link connectivity string crossover is an intra-cluster genetic operator.

IV. Experimental Results

MOGAC is a prototype consisting of approximately 18,000 lines of C++ and Bison code. Our results were obtained on a 200 MHz Pentium Pro system with 96 MB of main memory running the Linux operating system. We compare our results with those of Yen [5], Hou [17], and COSYN [21], which were obtained on a SPARCstation 20, as well as those of SOS [13], which were obtained on a Solbourne Series5e/900 (similar to a SPARC 4/490). The CPU times are given in seconds.

MOGAC's input consists of two ASCII files. The first file specifies the attributes of each PE, IC, and link type which may be used to implement an architecture. In addition, this file specifies the relationships between PEs and tasks, *i.e.*, for each PE it contains arrays specifying the worst-case execution times, average power consumptions, and peak power consumptions of each task on that PE. The second file specifies the topologies, periods, deadlines, tasks, and communication flows associated with all the task graphs comprising the system specification. MOGAC runs without designer intervention and, upon halting, outputs one or more solutions. Each solution is a system architecture consisting of a price, power consumption, PE allocation, IC allocation, link allocation, core assignments, task assignments, link connectivities, task schedules for each PE, and communication event schedules for each link.

A. Price Optimization

MOGAC has a slew of parameters which can be modified to tune its performance. Although every problem has its own optimal parameter settings, it would be inappropriate to only report the CPU time necessary to achieve a given solution if significantly more time was spent finding a good set of parameters. We, therefore, use the same set of parameters for all the examples presented in this subsection. In addition, the same value is used to seed MOGAC's random number generator for every result presented in this paper, with the exception of Table IV.

It was necessary to trade off run-time against solution quality when selecting a general parameter set for the examples in this subsection. Using a smaller solution pool and cluster pool would allow MOGAC to produce low-cost solutions for simple examples more rapidly. However, the solution quality for more complicated examples would suffer. For illustrative purposes, run-times achieved by tuning MOGAC's parameters to an individual problem's complexity, as well as the run-times which resulted from using the general parameter set, are shown in the price optimization tables.

Table I compares MOGAC's performance with that of COSYN [21] and Yen's system [5] when each is run on the clustered and unclustered versions of Hou's task graphs [17]. Task clustering is the process of using a pre-pass 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 co-synthesis problem by decreasing the number of tasks which must be assigned. Hou ran Yen's system on the clustered and unclustered versions of his graphs. We use the same clusters as Hou when comparing our results with his, and those of COSYN. For the example upon which it was possible to make a comparison

TABLE I
HOU'S EXAMPLES

Example	No. of Tasks	Yen's system		COSYN		MOGAC		
		Price	CPU time (s)	Price	CPU time (s)	Price	CPU time (s)	Tuned time (s)
Hou 1 & 2 (unclustered)	20	170	10,205	N. A.	N. A.	170	5.7	2.8
Hou 3 & 4 (unclustered)	20	210	11,550	N. A.	N. A.	170	8.0	1.6
Hou 1 & 2 (clustered)	8	170	16.0	170	5.1	170	5.1	0.7
Hou 3 & 4 (clustered)	6	170	3.3	N. A.	N. A.	170	2.2	0.6

TABLE II
PRAKASH & PARKER'S EXAMPLES

Example ⟨Performance⟩	No. of Tasks	SOS		COSYN		MOGAC		
		Price	CPU time (s)	Price	CPU time (s)	Price	CPU time (s)	Tuned time (s)
Prakash & Parker 1 ⟨4⟩	4	7	28	N. A.	N. A.	7	3.3	0.2
Prakash & Parker 1 ⟨7⟩	4	5	37	5	0.2	5	2.1	0.1
Prakash & Parker 2 ⟨8⟩	9	7	4,511	N. A.	N. A.	7	2.1	0.2
Prakash & Parker 2 ⟨15⟩	9	5	385,012	5	0.4	5	2.3	0.1

between MOGAC and COSYN, COSYN's performance was similar to that of MOGAC. Unfortunately, this was a small example, containing only eight tasks. The only existing implementation of COSYN is solely owned by Lucent. We relied on results reported in the literature to compare with COSYN.

It is interesting to observe the impact of increased problem complexity upon MOGAC and Yen's system. MOGAC's CPU time increases slightly when it solves the unclustered versions of Hou's examples instead of the clustered versions. In contrast, Yen's system takes approximately 1,000 times as long to produce solutions. Despite consuming significantly less CPU time, in one case MOGAC produces a lower-price architecture than Yen's system. The difference in solution quality between Yen's system and MOGAC is likely to be a result of the general class of algorithm used by each system. The run-time of Yen's system is significantly influenced by the method used to guarantee schedule validity. In addition, Yen uses an optimization algorithm in which a single solution is iteratively improved. Although the search is not blind, only a single stage of look-ahead is used. For each real evaluation, only a single solution is implicitly evaluated. Invalid solutions are terminated instantly instead of being improved upon. The use of a locality-preserving crossover operator allows MOGAC's genetic algorithm to implicitly evaluate more than one solution for each explicit evaluation (see Section II-A). Instead of maintaining a single solution which crawls across the solution space, MOGAC maintains multiple solutions which spread out across the solution space. These solutions share information with each other. MOGAC attempts to improve invalid solutions, which are otherwise of high quality, instead of terminating them immediately. We believe that these features allow MOGAC to tackle large problem instances without a prohibitive increase in execution time.

The hyperperiod contraction heuristic described in Section III-C was applied to the clustered and unclustered versions of the task graphs called Hou 3 & 4. The period of one of the task graphs in these examples was contracted by 5%. We were able to decrease MOGAC's CPU time, without decreasing solution quality, by tuning the size of MOGAC's solution pool and making its halting conditions less tolerant.

Table II compares MOGAC's performance with that of SOS [13] and COSYN when they are applied to Prakash and Parker's task graphs. The performance number shown by each task graph is the worst-case finish time for the task graph. For instance, "Prakash & Parker 1 ⟨4⟩," refers to Prakash and Parker's first task graph with a worst-case finish time of 4 time units. In these graphs, an unconventional model for communication is used [13]. A task may begin executing before all of its input data have arrived. We converted their specifications into graphs which conform to the conventional communication model, *i.e.*, a task can only begin execution when all of its input data have arrived. Their model implies that part of each task is independent of the task's input data. This is expressed by splitting each task into a portion which depends on input data and a portion which is independent of its input data. We assure that each task's subtasks are assigned to the same PE. It is not surprising that SOS requires significantly more CPU time than MOGAC. The mixed integer linear programming algorithm used in SOS is exhaustive, evaluating all solutions which have the potential to be optimal, while MOGAC makes no guarantee of optimality. However, for each of these examples, we can see that MOGAC also obtains optimal results.

Table III compares MOGAC's performance with that of Yen's system when each system is applied to Yen's large random task graphs [5]. Random 1 consists of six task graphs, each of which contains approximately

TABLE III
YEN'S LARGE RANDOM EXAMPLES

Example	No. of Tasks	Yen's system		MOGAC		
		Price	CPU Time (s)	Price	CPU Time (s)	Tuned (s)
Yen's Random 1	50	281	10,252	75	6.4	0.2
Yen's Random 2	60	637	21,979	81	7.8	0.2

TABLE IV
EFFECT OF VARYING RANDOM SEED

Problem	Reported Price	Effort	Average Price	Average CPU Time (s)
Hou 1 & 2 (unclustered)	170	1	183.3	23.1
		2	175.0	56.2
		3	176.7	89.4
		4	171.7	156.8
Hou 3 & 4 (unclustered)	170	1	176.0	41.8
		2	177.7	80.9
		3	171.0	125.5
		4	171.7	226.0
Hou 1 & 2 (clustered)	170	1	176.3	11.9
		2	176.7	26.3
		3	170.7	39.7
		4	170.7	73.3
Hou 3 & 4 (clustered)	170	1	176.6 [a]	12.6
		2	175.7	30.5
		3	174.0	41.6
		4	178.7	72.4
Prakash & Parker 1 ⟨4⟩	7	1	7.0	11.4
		2	7.0	31.7
		3	7.0	49.9
		4	7.0	89.6
Prakash & Parker 1 ⟨7⟩	5	1	5.0	8.0
		2	5.0	24.8
		3	5.0	40.1
		4	5.0	73.3
Prakash & Parker 2 ⟨8⟩	7	1	7.3	10.1
		2	7.1	27.2
		3	7.0	42.3
		4	7.0	72.1
Prakash & Parker 2 ⟨15⟩	5	1	5.0	6.0
		2	5.0	18.0
		3	5.0	29.5
		4	5.0	54.1
Yen's Random 1	75	1	75.0	18.7
		2	73.7	80.1
		3	74.4	125.2
		4	74.4	225.6
Yen's Random 2	81	1	81.0	32.1
		2	81.0	91.1
		3	81.0	148.0
		4	81.0	266.4

[a] In the case of the effort 1 solution, MOGAC found valid solutions for 29 of the 30 random seed values. For all other effort levels, MOGAC found valid solutions for all of the random seed values.

eight tasks. There are eight PE types available in this example. Random 2 consists of eight task graphs, each of which contains approximately eight tasks. There are 12 PE types available in this example. Neither of these examples contains communication links; all communication costs are 0. The observations comparing MOGAC to Yen's system, in the discussion of Table I, apply to these examples as well.

The task graph periods in these systems are co-prime. Therefore, the hyperperiod contraction heuristic presented in Section III-C significantly reduces the number of task graph copies that MOGAC is required to schedule. The heuristic was prevented from specifying task graph periods to be less than the corresponding deadlines, or greater than the periods specified in [5]. MOGAC's performance depends on the seed given to its pseudo-random number generator. Each problem instance has a different random seed for which MOGAC produces the best results most rapidly. However, MOGAC is able to arrive at a high-quality solution

TABLE V
EFFORT DEFINITIONS

Effort	Solutions	New Solutions	Clusters	New Clusters	Generations Before Halting
1	26	10	33	17	5
2	34	14	40	20	10
3	36	14	44	22	14
4	44	18	45	23	20

TABLE VI
POWER CONSUMPTION EXAMPLES

Example	No. of Tasks	MOGAC Ignoring Power			MOGAC Optimizing Power		
		Price	Power	CPU time (s)	Price	Power	CPU time (s)
Hou 1 & 2 (unclustered)	20	170	60.6	16.9	170	51.8	89.6
Hou 3 & 4 (unclustered)	20	170	62.4	30.7	170	48.6	26.3
Hou 1 & 2 (clustered)	8	170	75.3	7.8	170	62.5	9.5
Hou 3 & 4 (clustered)	6	170	47.1	3.9	170	43.3	5.1
Prakash & Parker 1 ⟨4⟩	4	7	75.4	10.1	7 15	75.4 64.2	20.1
Prakash & Parker 1 ⟨7⟩	4	5	44.4	8.5	5 7 10	44.4 35.1 21.5	16.9
Prakash & Parker 2 ⟨8⟩	9	7	49.8	8.4	7 12	49.8 40.0	17.2
Prakash & Parker 2 ⟨15⟩	9	5	48.0	6.32	5 7 12	48.0 26.8 21.8	22.4
Yen's Random 1	50	75	25.6	43.1	75 151 225 301	17.7 8.2 6.8 3.3	453.1
Yen's Random 2	60	81	39.8	59.2	81 153 158 214 338	34.4 25.4 15.7 9.9 7.0	268.8

given suboptimal seeds, if its solution pool size or cluster pool size are increased, or its halting conditions are made more lenient.

Table IV shows the average results of optimizing each of the price optimization examples thirty times, given random seeds ranging from one to thirty. In this table, *reported price* is the price reported for a single run of MOGAC with a fixed seed (see Tables I, II, and III). *Effort* corresponds to the computing resources MOGAC is allowed to dedicate to the problem. The meaning of each effort value is given in Table V. The *average price* column shows the average price of the solutions. MOGAC was run in single-objective optimization mode for these experiments. Therefore, each run produces only one non-dominated solution. When MOGAC is given the same parameters as were used in the previous tables in this section, there are a small number of example-random seed combinations for which it does not arrive at valid solutions. Slightly more liberal parameters were used for Table IV than for the preceding tables. This ensures that *average price* is meaningful. Note that, when allowed a modest increase in run-time, MOGAC robustly deals with varying random seeds.

Table V shows the parameter settings corresponding to each effort setting in Table IV. *Solutions* is the total number of solutions per cluster and *new solutions* is the number of solution reproductions which occur per generation, per cluster. Similarly, *clusters* and *new clusters* are the total number of clusters and the number of cluster reproductions per generation. *Generations before halting* is the number of generations which must pass without improvement in MOGAC's solution pool before MOGAC halts.

B. Multi-Objective Power and Price Optimization

Table VI displays the results of simultaneously optimizing the price and power consumption of system architectures based on examples presented in past work. The database for the example called Yen's Random 2 contains two IC types and two core types in addition to the processor types specified by Yen, for a total of 14 PE types. The values shown in the "Ignoring Power" column indicate the results of running MOGAC, in single objective price optimization mode, on the same embedded system specifications. MOGAC was given the same parameters for all of the examples in this subsection, although the parameter set used for price optimization in Section IV-A differs from the parameter

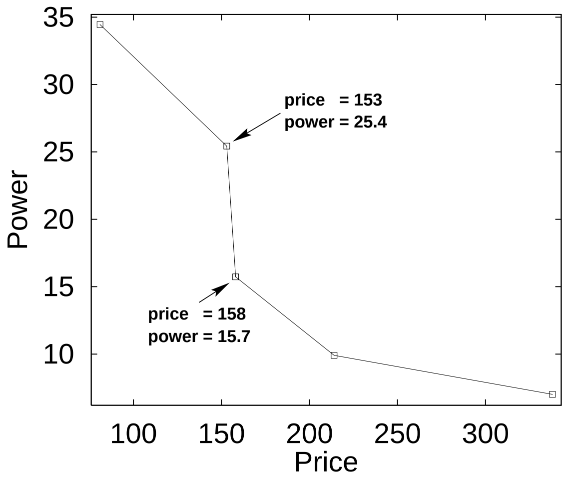

Fig. 19. Yen's Random 2 example.

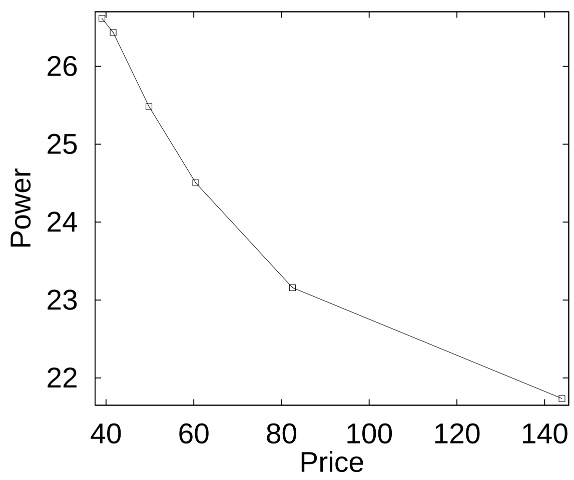

Fig. 20. Very Large Random 1 example.

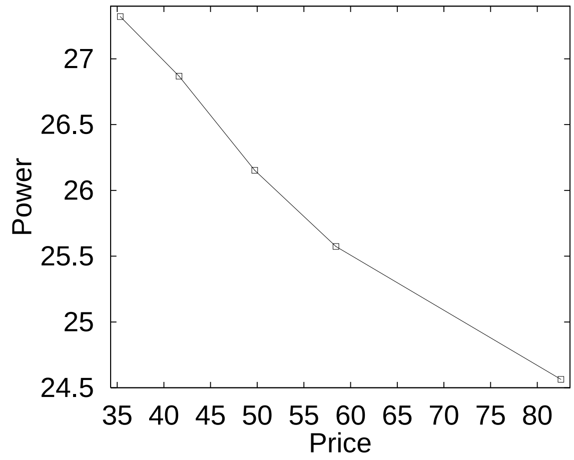

Fig. 21. Very Large Random 2 example.

set used in this subsection. The database files used for these examples are available via anonymous FTP at ftp://ftp.ee.princeton.edu/pub/dickrp/Trans/Mogac

The advantage of multiobjective optimization, over the use of a linear weighted sum, can clearly be seen in Table VI. When MOGAC simultaneously optimizes power and price, it provides a designer with its entire set of non-inferior solutions. For each system specification, only a single co-synthesis run was necessary to produce all the corresponding architectures whose costs are listed in Table VI.

MOGAC provides an upper bound on a problem's Pareto-optimal solution set instead of merely producing a single solution. This approach allows a designer to see the relationship between the costs of different architectures which satisfy the same system specification. Fig. 19 illustrates the danger of selecting a solution without knowing the shape of a system's non-inferior solution curve. Although all of MOGAC's solutions for Yen's Random 2 example are non-inferior, a designer would rarely select the solution with a price of 153 and a power consumption of 25.4 when, for a price penalty of only 5, a solution with a power consumption of 15.7 can be obtained. Presenting a non-inferior solution set shows the designer the cost trade-offs available between different solutions.

Figs. 20 and 21 show the results of optimizing very large multi-rate examples which require communication link synthesis. These pseudo-random examples were generated with the TGFF system [39]. They are available via anonymous FTP. The first very large example contains 8 task graphs, each of which has approximately 63 tasks. There are 8 PE types and 5 link types available. MOGAC took 40.9 CPU minutes to arrive at the non-inferior solution curve shown in Fig. 20. The second very large example contains 10 task graphs, each of which has approximately 100 tasks. There are 20 PE types and 10 link types available. MOGAC took 203.5 CPU minutes to arrive at the non-inferior solution curve shown in Fig. 21. The primary purpose of these examples is to demonstrate that MOGAC can rapidly solve extremely large problem instances. We hope that others will use these examples for comparative purposes.

V. Conclusions

In this paper, we have presented a method for the co-synthesis of low-power real-time multi-rate heterogeneous hardware-software distributed embedded systems. A novel multiobjective genetic algorithm, which allows exploration of the Pareto-optimal set of architectures instead of providing a designer with a single solution, has been practically applied to a number of examples found in the literature. MOGAC has been shown to rapidly synthesize architectures with costs that are lower than or equal to those presented in previous work. For large examples upon which comparisons with other systems are possible, MOGAC produces significantly lower-cost solutions, despite requiring orders of magnitude less run-time. It has been demonstrated that adaptive multiobjective genetic algorithms are well suited to solving the co-synthesis problem.

REFERENCES

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

[2] M. Chiodo *et al.*, "Hardware/software co-design of embedded systems," *IEEE Micro*, vol. 14, pp. 26–36, Oct. 1993.

[3] G. De Micheli, "Computer-aided hardware-software codesign," *IEEE Micro*, pp. 10–16, Aug. 1994.

[4] P. H. Chou, R. B. Ortega, and G. Borriello, "The chinook hardware/software co-synthesis system," in *Proc. Int. Symp. System Synthesis*, pp. 22–27, Sept. 1995.

[5] T.-Y. Yen, *Hardware-Software Co-Synthesis of Distributed Embedded Systems*. PhD thesis, Dept. of Electrical Engg., Princeton University, June 1996.

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

[7] R. K. Gupta and G. De Micheli, "Hardware-software cosynthesis for digital systems," *IEEE Design & Test of Computers*, pp. 29–41, Sept. 1994.

[8] R. Ernst, J. Henkel, and T. Benner, "Hardware/software cosynthesis for microcontrollers," *IEEE Design & Test of Computers*, vol. 12, pp. 64–75, Dec. 1993.

[9] E. Barros, W. Rosenstiel, and X. Xiong, "A method for partitioning UNITY language to hardware and software," in *Proc. European Design Automation Conf.*, pp. 220–225, Sept. 1994.

[10] S. Devadas and S. Malik, "A survey of optimization techniques targeting low power circuits," in *Proc. Design Automation Conf.*, pp. 242–247, June 1995.

[11] A. P. Chandrakasan, "Design of portable systems," in *Proc. Custom Integrated Circuits Conference*, pp. 259–266, May 1994.

[12] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, NY, 1979.

[13] S. Prakash and A. Parker, "SOS: Synthesis of application-specific heterogeneous multiprocessor systems," *J. Parallel & Distributed Computers*, 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. European Design Automation Conf.*, pp. 8–13, Sept. 1995.

[15] J. D'Ambrosio and X. Hu, "Configuration-level hardware/software partitioning for real-time systems," in *Proc. Int. Workshop Hardware/Software Codesign*, vol. 14, pp. 34–41, Aug. 1994.

[16] T.-Y. Yen and W. H. Wolf, "Communication synthesis for distributed embedded systems," in *Proc. Int. Conf. Computer-Aided Design*, pp. 288–294, Nov. 1995.

[17] J. Hou and W. Wolf, "Process partitioning for distributed embedded systems," in *Proc. Int. Workshop Hardware/Software Codesign*, pp. 70–76, Mar. 1996.

[18] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, "System level hardware/software partitioning based on simulated annealing and tabu search," *Design Automation for Embedded Systems*, vol. 2, pp. 5–32, Jan. 1997.

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

[20] S. Srinivasan and N. K. Jha, "Hardware-software co-synthesis of fault-tolerant real-time distributed embedded systems," in *Proc. European Design Automation Conf.*, pp. 334–339, Sept. 1995.

[21] B. Dave, G. Lakshminarayana, and N. K. Jha, "COSYN: Hardware-software co-synthesis of embedded systems," in *Proc. Design Automation Conf.*, pp. 703–708, June 1997.

[22] F. Romeo, *Simulated Annealing: Theory and Applications to Layout Problems*. PhD thesis, Dept. of Electrical Engg. & Computer Science, University of California, Berkeley, Mar. 1989.

[23] D. Herrmann, J. Henkel, and R. Ernst, "An approach to the adaptation of estimated cost parameters in the COSYMA system," in *Proc. Int. Workshop Hardware/Software Codesign*, pp. 100–107, Mar. 1994.

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

[25] C. M. Fonseca and P. J. Fleming, "Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization," in *Proc. Int. Conf. Genetic Algorithms*, pp. 416–423, July 1993.

[26] J. L. Breeden, "Optimizing stochastic and multiple fitness functions," in *Proc. Evolutionary Programming*, vol. 4, pp. 127–134, Mar. 1995.

[27] D. Saha, R. Mitra, and A. Basu, "Hardware software partitioning using genetic algorithm approach," in *Proc. Int. Conf. VLSI Design*, pp. 155–160, Jan. 1997.

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

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

[30] E. L. Lawler and C. U. Martel, "Scheduling periodically occurring tasks on multiple processors," *Information Processing Letters*, vol. 7, pp. 9–12, Feb. 1981.

[31] A. E. Eiben, E. H. L. Aarts, and K. M. V. Hee, "Global convergence of genetic algorithms: A Markov chain analysis," in *Lecture Notes in Computer Science: Parallel Problem Solving from Nature*, vol. 496, pp. 4–12, Springer-Verlag, Berlin, Germany, 1991.

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

[33] V. Tiwari, S. Malik, and A. Wolfe, "Compilation techniques for low energy: An overview," in *Proc. Int. Symp. Low-Power Electronics*, pp. 38–39, Oct. 1994.

[34] B.-D. Rhee *et al.*, "Issues of advanced architectural features in the design of a timing tool," in *Proc. Workshop Real-Time Operating Systems and Software*, pp. 59–62, May 1994.

[35] K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," *Proc. IEEE*, vol. 82, pp. 55–67, Jan. 1994.

[36] B. P. Dave, *Hardware-Software Co-Design of Heterogeneous Real-Time Distributed Embedded Systems*. PhD thesis, Dept. of Electrical Engg., Princeton University, 1998.

[37] S. Kim and J. Browne, "A general approach to mapping of parallel computations upon multiprocessor architectures," in *Proc. Int. Conf. Parallel Processing*, vol. 2, pp. 1–8, Aug. 1988.

[38] A. E. Smith and D. M. Tate, "Genetic optimization using a penalty function," in *Proc. Int. Conf. Genetic Algorithms*, pp. 499–503, July 1993.

[39] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: task graphs for free," in *Proc. Int. Workshop Hardware/Software Codesign*, pp. 97–101, Mar. 1998.