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

Robert P. Dick and Niraj K. Jha

Department of Electrical Engineering
Princeton University
Princeton, New Jersey 08544

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.

1 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]. 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 [2]. Co-synthesis systems generate feasible, low-cost architecture descriptions without designer intervention.

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

Related work often limits the co-synthesis solution space to architectures consisting of one CPU and one application-specific integrated circuit (ASIC). Most real-life embedded systems are composed of multiple general-purpose processors and ASICs, *i.e.*, they are distributed heterogeneous architectures [1]. We will consider the general problem in which the numbers and types of PEs and links in an architecture are not bounded.

Optimal co-synthesis is an intractable problem. Allocation/assignment and scheduling are each known to be NP-complete for distributed systems [3]. It is, therefore, not surprising that all co-synthesis systems which rely on optimal mixed integer linear programming [4] and exhaustive exploration [5] 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. The constructive algorithm used in [6] was the first to target low power. However, like iterative improvement algorithms [2], [7], conventional constructive algorithms may become trapped in local minima. A genetic algorithm was previously applied to the hardware-software partitioning problem [8]. However, in this work only one general-purpose processor was allowed, there were no provisions for synthesizing systems with multi-rate periodic task graphs, and communication links were not modeled. In addition, run-times were not given and the genetic algorithm only optimized one variable: price.

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 [9] 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.

This work was supported in part by an NSF Graduate Fellowship and in part by NSF under Grant No. MIP-9423574.

2 Preliminaries

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

2.1 Embedded System Model

In this subsection, we provide high-level descriptions of the data types MOGAC operates on. Information about the computations carried out upon them can be found in Section 3.

Cost: A cost is a variable that a co-synthesis system attempts to minimize. Price, power consumption, and schedule length are examples of costs.

Task graph: A task graph 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. 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.

Processing element: A PE executes tasks. Two types of PEs are modeled: *grouped PEs* and *independent PEs*. Independent PEs represent general-purpose processors which can only execute one task at a time. However, multiple grouped PEs 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 following information establishes the relationships between tasks and independent PEs:

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

In addition to these arrays, independent PEs have price and idle power consumption values. The following information establishes the relationship between tasks and grouped PEs:

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

Grouped PEs do not have an inherent price. However, each grouped PE is assigned to an IC which does have a price. The following variables are associated with ICs: price, pin count, device count, idle power consumption, peak power dissipation, speed, and power efficiency. Each grouped PE places device count and pin count requirements on the IC to which it is assigned. For an architecture to be valid, each IC must meet the pin count and device count requirements of the grouped PEs assigned to it. In addition, each IC must meet the peak power dissipation requirements of the tasks assigned to the grouped PEs implemented on it.

The worst-case execution time for a task assigned to a grouped PE is equivalent to its relative worst-case execution time divided by the speed of the IC on which the grouped PE 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 grouped PE is implemented.

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, 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. 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.

Constraints: 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.

Strings: The *PE allocation string* is an array of integers. Each integer represents the number of instances of a PE type present in a solution. For a genetic algorithm to function properly, it is important for its strings to preserve locality, *i.e.*, similar entries must be located closer to each other in a string than dissimilar entries [10]. As mentioned earlier, 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 ordering 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. A locality-preserving heuristic is then used to impose an order on these vectors. The *link allocation string* and *IC allocation string* are similar to the PE allocation string and they are ordered using similar algorithms.

The *task assignment string* is an ordered string of PE instance references specifying the PE to which each task is assigned. This string is ordered by conducting a depth-first traversal of all the task graphs in the system specification. The *grouped PE assignment string* is an ordered string of IC instance references specifying the IC to which each grouped PE is assigned.

The *link connectivity string* is an ordered string of IC and independent PE instance references specifying the ICs and independent PEs to which each communication link is connected. The order of link types in this string is based on their order in the link allocation string.

2.2 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 obtain a new generation of solutions. The lowest quality solutions are then removed from the pool [10]. Genetic algorithms excel at simultaneously optimizing multiple conflicting costs. They have the ability to escape local minima and communicate information between solutions.

Next, we define some basic terms used to discuss genetic algorithms. In a conventional genetic algorithm, every solution is represented by a *string* of values (usually Boolean). Although we discuss the genetic algorithm used by MOGAC in conventional terms, no primitive Boolean string is ever computed. As discussed in Section 2.1, strings 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.

In conventional genetic algorithms, 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. Conventionally, a bit in the solution's string is inverted. *Cross-over* combines parts of different solutions. This is the operator that gives genetic algorithms their strength; it allows different solutions to share information with each other. Conventionally, two strings are cut at the same offset from their starting points and the portions following the cut are swapped. The operators used by MOGAC are analogous to, but more complicated than, conventional genetic operators.

2.3 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 of 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.

Multiobjective genetic algorithms avoid the problems associated with collapsing multiple costs into one value. In such algorithms, solutions are ranked relative to each other, *i.e.*, a solution's rank is the number of other solutions to which it is not in some way inferior. The use of a multiobjective genetic algorithm allows MOGAC to explore the Pareto-optimal solution set instead of relying on an arbitrary weighting array to guide its search.

3 Algorithm Description

In this section, we give a description of the algorithms used in MOGAC.

3.1 Overview of the Algorithm

In this subsection, we present a high-level description of MOGAC's hierarchical genetic algorithm for co-synthesis. Initially, there are approximately 400 members in MOGAC's solution pool. As shown in Figure 1, the pool is broken into clusters of solutions. Every solution within a cluster has the same allocation strings. However, each solution's link connectivity and assignment strings may differ from those of other solutions within the cluster. Allocation string cross-over only occurs between clusters of solutions. Assignment string cross-over and link connectivity string cross-over only occur between solutions in the same cluster.

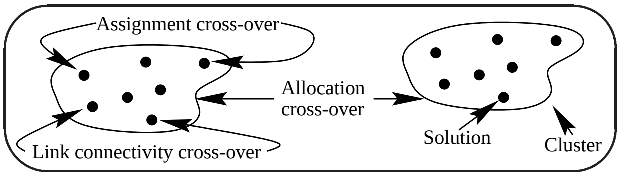

Figure 1: Solution clusters

Each solution's allocation string is initialized with a heuristic constructive algorithm; PEs are randomly selected until there exists at least one PE upon which each task can execute. Despite the simplicity of this heuristic, it frequently produces high-quality allocations. From this starting point, the genetic algorithm searches for higher-quality allocations. During the first generation or two, many of the solutions are invalid because there are not sufficient resources available to carry out tasks rapidly enough to meet the system's real-time deadlines. As the allocation strings mutate, more PEs and links become available and are incorporated into solutions through assignment string mutation and connectivity string mutation.

At this point, the evolve-evaluate cycle starts to supply feedback to the genetic algorithm. Solutions, which have adequate resources to meet their real-time constraints, consume little power, and make the best use of these resources, dominate other solutions. Non-inferior solutions, those which are not dominated by any other solutions, reproduce more often than dominated solutions. Occasionally, dominated solutions reproduce as well. Infrequently, allocation cross-over occurs, changing an entire cluster of solutions simultaneously.

MOGAC treats all non-inferior solutions equally even if they violate hard constraints. After each generation, MOGAC displays the costs of the members of its non-inferior solution set. At any time, the designer may halt the system and examine an individual solution in detail. If the designer chooses not to interfere with the co-synthesis run, MOGAC automatically adapts its parameters based upon its solution pool's rate of improvement. At the end of a run, the system presents its user with all of the valid non-inferior solutions it found.

3.2 Parameter Adaptation

In this subsection, we discuss the manner in which MOGAC adapts its own parameters. In general, each parameter has a starting value and an adaptation rate. At the end of each generation in which the quality of the solution pool did not improve, the relevant parameters are adjusted by their adaptation rates. When an improvement occurs, the relevant parameters are reset to their initial values. If a pre-specified number of generations pass without improvement, the co-synthesis run halts.

MOGAC has a number of static parameters. Although these parameters can be changed from run to run, they do not adapt during a run. There are a fixed number of clusters in the solution pool and each cluster contains a fixed number of solutions. In addition, the ratio of real to total task graph copies is constant throughout a run (see Section 3.3).

Most parameters are dynamic; they adapt based upon the rate at which the quality of the solution pool is improving. The frequency with which each genetic operator is applied is parametric and adaptable. There are a number of parameters which control the aggressiveness of MOGAC's genetic operators. Other parameters control the probability of a superior solution being replaced by an inferior one. If desired, these values can be set to monotonically decrease during a co-synthesis run, allowing MOGAC to approximate a simulated annealing algorithm.

3.3 Multi-Rate Systems

As mentioned in Section 1, co-synthesis systems which use the results in [9] to guarantee valid schedules for multi-rate systems have difficulty synthesizing architectures in which the hyperperiod, *i.e.*, the LCM of the periods of all task graphs, is much larger than the periods of individual task graphs. For such systems, it is necessary to assign and schedule the tasks and communication events in numerous copies of some task graphs. The number of copies of a task graph is the system's hyperperiod divided by the graph's period. Past work has dealt with this problem by forcing corresponding tasks in different copies of the same task graph to be assigned to the same PE instance [2]. Although this does reduce the complexity of assignment and scheduling, it decreases the flexibility of a co-synthesis system. To derive the most efficient architecture for a given system specification, it may be necessary to assign corresponding tasks in different copies of a task graph to different PEs. MOGAC uses two heuristics to tackle system specifications with large hyperperiods. The first of these is an extension of a method used in real-time computing [11].

Hyperperiod contraction: The problems caused by large hyperperiods 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 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.

Scalable implicit task graph copies: We have developed a method in which some of the task graph copies in the hyperperiod are *implicit* and some are *real* (see Figure 2). 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 still 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 (see Section 3.6). 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.

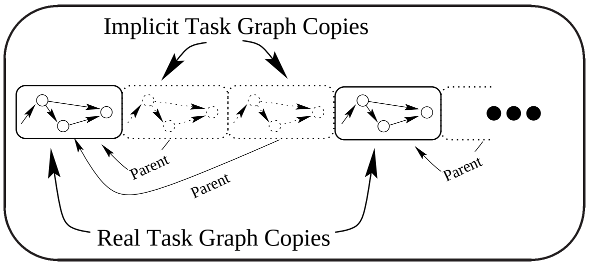

Figure 2: Task graph copies

3.4 Ranking and Reproduction

In this subsection, we explain the manner in which solutions and clusters are selected for reproduction.

Solution ranking and reproduction: Solutions within a cluster are ranked using the method presented in Section 2.3. In each generation, a pre-specified number of solutions within each cluster are eliminated to make space for the reproduction of other solutions. Solutions are selected for reproduction by indexing inward from the highest-ranking solution with a Gaussian random variable. The inverse of the variance of this variable is defined as *elitism*. MOGAC's elitism is specified by the user. A designer can choose to protect the highest-ranked solutions in the solution pool from mutation and cross-over or to allow the modification of any solution.

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 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,

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

and,

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

Once cluster ranks have been determined, cluster reproduction is analogous to solution reproduction.

3.5 Evolution

In this subsection, we give an overview of mechanisms through which the solution pool evolves. Evolution in MOGAC is hierarchical. The cluster-level genetic operators simultaneously affect every solution in a cluster, while solution-level operators only affect individual solutions within a cluster. Solution-level operators are typically applied more frequently than cluster-level operators.

Cluster-level operators: PE allocation mutation causes a PE to be added or removed from a cluster's allocation. PE allocation cross-over selects two PE type cut-points at random and two clusters at random. The PE counts of the PE types between the cut-points are swapped between the selected clusters. The link allocation and IC allocation operators are analogous to the PE allocation operators.

Solution-level operators: Task assignment mutation causes a randomly selected task to assign itself to a different PE instance. Task assignment cross-over selects two task cut-points at random and two solutions at random. The PE assignments of the tasks between the cut-points are swapped between the selected solutions.

Grouped PE assignment mutation causes a randomly selected grouped PE to assign itself to a different IC instance. Grouped PE assignment cross-over selects two grouped PE assignment cut-points at random and two solutions at random. The IC assignments of the grouped PEs between the cut-points are swapped between the selected solutions.

Link connectivity mutation causes a link instance to randomly reconnect itself to IC and independent PE instances. Link connectivity cross-over selects two link cut-points at random and two solutions at random. For each link between the cut-points, the list of IC and independent PE instances to which the link is connected is swapped between the selected solutions.

3.6 Scheduling

Scheduling occurs in MOGAC's inner loop. Since the use of a sophisticated, but slow, algorithm would reduce the design space that can be explored in a given amount of time, we use a heuristic list scheduling algorithm that, in the theoretical worst case, runs in time $\mathcal{O}\left(edges^2 + nodes^2\right)$. However, the typical run-time is $\mathcal{O}\left(edges + nodes\right)$. The scheduling algorithm is tailored to periodic systems and can even handle the case when task graphs have periods less than their deadlines.

Prioritization: The priority of every task in each of the system's real task graph copies is calculated. Each task graph copy has an earliest start time (EST) which is determined by multiplying its position in the hyperperiod by its period. The EST of every task is determined by conducting a topological sort beginning at the start node of each real task graph copy and propagating the execution times of the tasks along each path forward. The latest start time (LST) of every node is determined similarly, by propagating the execution times of tasks backward from the nodes which have deadlines.

A node's *slack* is the difference between its LST and EST. A node's *cumulative slack* is the sum of the slacks of the nodes along the highest-slack path from that node to the graph's start node. A topological sort of the graph, beginning at the start node, allows each node's cumulative slack to be calculated. At re-converging paths, the maximum of the cumulative slacks of the parents is propagated forward. Nodes with low cumulative slacks have higher priorities than nodes with high cumulative slack. This prioritization method was selected because of its speed ($\mathcal{O}\left(nodes + edges\right)$) and the ease with which it can be applied to multi-rate systems. Cumulative slack provides reasonable relative priorities for nodes in different task graphs. Additionally, this prioritization method is compatible with the scalable implicit task graph copy heuristic described in Section 3.3.

Once node priorities have been determined, the nodes from the first copy of each task graph are introduced into a priority-ordered list. When a graph copy's start time is reached, the next real or implicit copy of that graph is introduced into the priority list. An offset is added to the cumulative slack of the nodes of all of the subsequent copies of each graph. This offset is a weighted sum of the task graph's highest cumulative slack and its period, multiplied by the index of the task graph copy, *i.e.*,

$$offset = (k_1 \cdot max\_cum\_slack + k_2 \cdot period) \cdot graph\_index$$

where k_1 and k_2 are constants.

Edge scheduling: Before a node is scheduled, all of its incoming edges are scheduled. For each incoming edge, the earliest possible finish time for the associated communication event is noted. For each link that may be used to carry out communication along a given edge, communication time is computed in the following manner,

$$commun\_time = time\_per\_packet \cdot \left\lceil \frac{data\_quantity}{packet\_size} \right\rceil$$

The communication event finish time is found by locating the earliest unused time slot in the link's schedule with a size equal to or larger than the $commun\_time$ which starts after the parent task finishes executing. As a result of this step, the total time required to schedule edges is actually

$$\mathcal{O}\left(\begin{matrix} max\_links \\ on\_edge \end{matrix} \times \begin{matrix} number \\ of\_edges \end{matrix} \times \begin{matrix} max\_events \\ on\_link \end{matrix}\right)$$

Fortunately, $max\_links\_on\_edge$ is rarely greater than 3 and $max\_events\_on\_link$ is usually significantly less than $number\_of\_edges$.

Once the earliest finish time for each edge has been computed, the communication events are scheduled in order of decreasing earliest finish time. Scheduling an edge may affect the earliest finish times of the remaining edges. Therefore, the earliest finish time of each edge is recomputed immediately before it is scheduled. However, the edges are not re-prioritized after each scheduling event because this would significantly slow down the scheduling algorithm.

Node scheduling: The PE instance to which each node is assigned is determined by the genetic algorithm. After all of a node's incoming edges have been scheduled, the node's task is scheduled to its PE in the earliest time slot which is large enough to allow execution and starts after the latest incoming communication event has completed.

3.7 Performance Evaluation

Performance evaluation consists of calculating a solution's costs and determining how severely they violate the soft and hard constraints imposed by the designer. In this subsection, we will explain how MOGAC does performance evaluation and then describe the process by which raw performance metrics are converted into system costs.

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, independent PEs, and links in the allocation strings. The completion time of each node in a task graph is recorded during scheduling (see Section 3.6). Therefore, the completion times of all nodes with deadlines are available for inspection. System power consumption is computed by stepping through

Table 1: Hou's examples

Example	Hou		COSYN		MOGAC		
	Price	CPU Time (s)	Price	CPU Time (s)	Price	CPU Time (s)	Tuned CPU Time (s)
Hou 1 & 2 (unclustered)	170	10,205	N. A.	N. A.	170	5.7	2.8
Hou 3 & 4 (unclustered)	210	11,550	N. A.	N. A.	170	8.0	1.6
Hou 1 & 2 (clustered)	170	16.0	170	6.9	170	5.1	0.7
Hou 3 & 4 (clustered)	170	3.3	N. A.	N. A.	170	2.2	0.6

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 [6].

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.

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

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

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 of the real and implicit task graph copies in the system. For every IC, the peak power dissipation, pin count, and device count requirements of all the grouped PEs assigned to that IC are summed. When an IC is not capable of meeting the requirements of the grouped PEs assigned to it, the appropriate hard constraint violations in the solution are increased.

4 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 [2], Hou [7], and COSYN [6], which were obtained on a SPARCstation 20, as well as those of Prakash and Parker [4], 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 of the task graphs comprising the system specification. MOGAC outputs one or more solutions. Each solution is a system architecture consisting of a price, power consumption, PE allocation, IC allocation, link allocation, grouped PE assignments, task assignments, link connectivities, task schedules for each PE, and communication event schedules for each link.

4.1 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 of 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.

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 1 compares MOGAC's performance with that of COSYN [6] and Hou's system when each is run on the clustered and unclustered versions of Hou's task graphs [7]. Task clustering is the process of using a prepass to collapse multiple tasks into a cluster of tasks. This cluster is treated like a single task during assignment and scheduling, *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. It is important, however, that tasks which communicate a large amount of data with each other be placed in the same cluster. This is due to the fact that communication between tasks which are assigned to the same PE is less expensive than communication between tasks which are assigned to different PEs. Hou used a task clustering algorithm which takes communication quantity and task execution efficiency into account. We use the same clusters as Hou when comparing our results with those of his system's, and those of COSYN.

It is interesting to observe the impact of increased problem complexity upon MOGAC and Hou'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, Hou'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 Hou's system.

The hyperperiod contraction heuristic described in Section 3.3 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%.

Table 2: Prakash & Parker's examples

Example $\langle$Performance$\rangle$	Prakash & Parker		COSYN		MOGAC		
	Price	CPU Time (s)	Price	CPU Time (s)	Price	CPU Time (s)	Tuned CPU Time (s)
Prakash & Parker 1 $\langle 4 \rangle$	7	28	N. A.	N. A.	7	3.3	0.2
Prakash & Parker 1 $\langle 7 \rangle$	5	37	5	0.2	5	2.1	0.1
Prakash & Parker 2 $\langle 8 \rangle$	7	4,511	N. A.	N. A.	7	2.1	0.2
Prakash & Parker 2 $\langle 15 \rangle$	5	385,012	5	1.5	5	2.3	0.1

Table 3: Yen's large random examples

Example	Yen		MOGAC		
	Price	CPU Time (s)	Price	CPU Time (s)	Tuned CPU Time (s)
Yen's Random 1	281	10,252	75	6.4	0.2
Yen's Random 2	637	21,979	81	7.8	0.2

Table 4: Power consumption examples

Example	MOGAC Ignoring Power			MOGAC Optimizing Power		
	Price	Power	CPU Time (s)	Price	Power	CPU Time (s)
Hou 1 & 2 (unclustered)	170	66.4	8.3	170	53.3	39.4
Hou 3 & 4 (unclustered)	170	69.0	8.3	170	64.4	47.2
Hou 1 & 2 (clustered)	170	71.2	6.5	170	56.1	18.4
				190	55.1	
				290	51.4	
Hou 3 & 4 (clustered)	170	47.1	2.2	170	43.3	23.9
				200	36.6	
				270	34.9	
Prakash & Parker 1 $\langle 4 \rangle$	7	75.4	3.4	7	75.4	10.3
Prakash & Parker 1 $\langle 7 \rangle$	5	44.4	2.2	5	44.4	5.9
Prakash & Parker 2 $\langle 8 \rangle$	7	70.6	2.2	7	49.8	7.2
Prakash & Parker 2 $\langle 15 \rangle$	5	48.0	1.5	5	48.0	16.6
				7	26.8	
				12	22.0	
Yen's Random 1	75	26.3	6.8	75	17.4	228.4
				173	6.4	
				293	5.8	
				299	5.4	
				323	3.4	
				339	2.9	
Yen's Random 2	68	48.5	14.4	68	38.5	571.7
				81	34.1	
				119	24.9	
				158	15.8	
				200	13.9	
				214	9.9	
				338	7.0	
				530	5.7	

Table 2 compares MOGAC's performance with that of Prakash and Parker's optimal mixed integer linear programming approach 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 $\langle 4 \rangle$," 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 [4]. 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. For Prakash & Parker 2, a point-to-point communication model is used. For each of these examples, we can see that MOGAC also obtains optimal results.

Table 3 compares MOGAC's performance with that of Yen's system when each system is applied to Yen's large random task graphs [2]. Random 1 consists of 6 task graphs, each of which contains approximately 20 tasks. There are 8 PE types available in this example. Random 2 consists of 8 task graphs, each of which contains approximately 20 tasks. There are 12 PE types available in this example. Neither of these examples contains communication links; all communication costs are 0.

The task graph periods in these systems are co-prime. Therefore, the hyperperiod contraction heuristic presented in Section 3.3 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 [2].

4.2 Multi-Objective Power and Price Optimization

Table 4 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 grouped PE types in addition to the independent PE 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 when doing multiobjective optimization for all of the examples in this subsection, although the general parameter set used for multiobjective price and power optimization differs from that used for price optimization. The database files used for these examples are available via anonymous FTP at ftp://ftp.ee.princeton.edu/pub/dickrp/

The advantage of multiobjective optimization, over the use of a linear weighted sum, can clearly be seen in Table 4. 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 of the corresponding architectures whose costs are listed in Table 4.

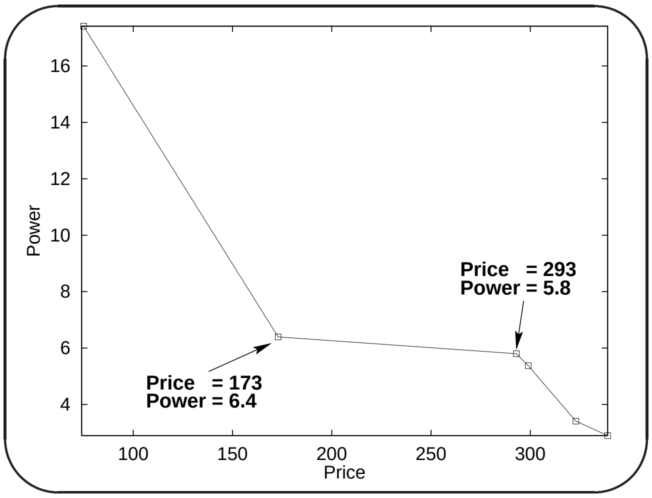

Figure 3: Non-inferior solutions for Yen's Random 1 Example

MOGAC provides information about the shape of 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. Figure 3 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 1 example are non-inferior, a designer would rarely select the solution with a price of 293 and a power consumption of 5.8 when, for a power penalty of only 0.6, a solution with a price of 173 can be obtained. Exploration of the problem instance's Pareto-optimal curve gives the designer information about the trade-offs available between different implementations of an embedded system.

5 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. When applied to large system specifications, MOGAC produces significantly lower-cost solutions than previous co-synthesis systems, 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] T.-Y. Yen, *Hardware-Software Co-Synthesis of Distributed Embedded Systems*. PhD thesis, Dept. of Electrical Engg., Princeton University, June 1996.

[3] 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.

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

[5] 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.

[6] 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.

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

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

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

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

[11] 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.