# CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems

Robert P. Dick and Niraj K. Jha

Department of Electrical Engineering
Princeton University
Princeton, New Jersey 08544

dickrp/jha@ee.princeton.edu

## Abstract

Field programmable gate arrays (FPGAs) are commonly used in embedded systems. Although it is possible to reconfigure some FPGAs while an embedded system is operational, this feature is seldom exploited. Recent improvements in the flexibility and reconfiguration speed of FPGAs have made it practical to reconfigure them dynamically, reducing the amount of hardware required in an embedded system. We have developed a system, called CORDS, which synthesizes multi-rate, real-time, periodic distributed embedded systems containing dynamically reconfigurable FPGAs. Executing different tasks on the same FPGA requires that potentially time-consuming reconfiguration be carried out between tasks. CORDS uses a novel preemptive, dynamic priority, multi-rate scheduling algorithm to deal with this problem. To the best of our knowledge, dynamically reconfigured FPGAs have not previously been used in hardware-software co-synthesis of embedded systems. Experimental results indicate that using dynamically reconfigured FPGAs in distributed real-time embedded systems has the potential to reduce their price and allow the synthesis of architectures which meet system specifications that would otherwise be infeasible.

## 1 Introduction

Until recently, dynamic reconfiguration of FPGAs in hard real-time embedded systems was impractical. FPGA reconfiguration times have conventionally been on the order of 100 ms. However, recently a number of companies have released products which improve upon the reconfiguration times of existing FPGAs by an order of magnitude or more [1], [2]. In particular, the largest member of the Xilinx XC6200 family, XC6264, can be completely reconfigured in under 200 $\mu$s. However, a price is paid for this speed. Rapid reconfiguration FPGAs cost approximately ten times as much as FPGAs using conventional architectures. Rapid reconfiguration FPGAs are a new product and production has been limited. Therefore, their price is likely to decrease in the future. Nonetheless, if price is a concern, it is important to consider more conventional FPGAs, which have large reconfiguration times. If one derives a schedule which locates different instances of the same task type adjacent to each other, the number of reconfigurations an FPGA needs to undergo will be reduced, resulting in significant time savings.

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

Hardware-software co-synthesis is the process of automatically synthesizing the hardware and software portions of an embedded system. Given an embedded system specification, a hardware-software co-synthesis system must select general-purpose processors, application-specific processing elements, and communication resources to use in the embedded system (*allocation*), determine which resource will be used to carry out each portion of the specification's computation and communication (*assignment*), and produce a schedule for all of the specification's computations and communications (*scheduling*). Thus, given an embedded system specification, a co-synthesis system produces a detailed description of an architecture which will meet the specification.

FPGAs fit naturally into the hardware-software co-synthesis design flow. The holy grail of configurable computing research is a system which will accept a problem description in a general-purpose programming language, automatically partition it between hardware (FPGAs) and software (general-purpose processors), synthesize the required hardware, and manage communication between the two domains. This problem closely mirrors the co-synthesis problem. By using FPGAs in co-synthesis, designers can take advantage of research in the reconfigurable computing field. There are already systems which accept algorithm descriptions in general-purpose languages, like ANSI-C, and automatically produce FPGA configurations [3].

Other work has been carried out in hardware-software co-synthesis [4]–[8]. However, CORDS is the first co-synthesis system to deal with dynamically reconfigured FPGAs. It automatically selects an allocation from a set of FPGAs, general-purpose processors, and communication resources. It assigns tasks to FPGAs and general-purpose processors, and determines the connectivity of communication resources. Finally, it derives schedules for tasks and communication events. It allows preemption on general-purpose processors. It optimizes the sequence of tasks on FPGAs to reduce the total reconfiguration time required while considering the priorities of individual tasks.

The paper is organized as follows. In Section 2, we define terms which will be used in the discussion of CORDS. Section 3 describes the scheduling algorithm used by CORDS. Section 4 explains the evolutionary framework CORDS uses for optimization. In Section 5, we give experimental results. We conclude with Section 6.

## 2 Preliminary Definitions

In this section, we present concepts which will be used while describing the algorithms comprising CORDS.

**Task graph:** A *task* is a portion of the computation an embedded system is required to carry out. Correlation and convolution are examples of task types. Multiplication is also a task type, although past work typically as-

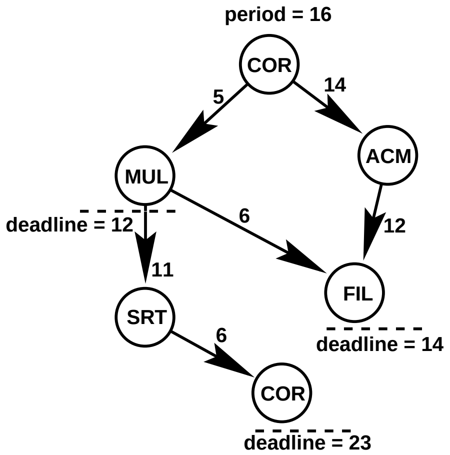


Figure 1: Task graph

sumes coarse-grained tasks, *i.e.*, each task is assumed to be something that would require multiple instructions on a general-purpose processor. An embedded system specification may contain more than one task of the same type.

A *task graph* is a directed, acyclic, connected graph consisting of a collection of tasks, each of which is associated with a task type, and a collection of directed edges, each of which is associated with a scalar denoting the amount of data which must be transferred between the tasks it connects. Edges represent communication events. In Fig. 1, each circular node, denoting a task, is labeled with its task type and each directed edge is labeled with the amount of data which flows along it. Each edge points away from its *parent task* and toward its *child task*. A task's *parents* are the tasks to which it is connected by incoming edges. A task's *children* are the tasks to which it is connected by outgoing edges. A directed edge may begin executing only after its parent task has completed executing. A task may begin executing only after all its incoming edges have completed executing. All tasks without outgoing edges have *deadlines*. However, any other task may also have a deadline (indicated by dashed lines in Fig. 1). The task with no incoming edges is the start task. If a task does not complete its execution before its deadline is reached, hard real-time constraints are violated. A task graph's *period* is the interval at which it repeats execution. It is possible for a task graph's period to be less than some of the deadlines of tasks within it. In embedded system specifications which contain such task graphs, the execution of multiple instances of the same task graph overlap in time. An embedded system specification may contain multiple task graphs, each of which may contain different tasks and deadlines. In addition, different task graphs may have different periods.

**Processor:** A processing element (PE) is a device that executes tasks. CORDS models two types of PEs: processors and FPGAs. A *processor* is a general-purpose processor used to carry out tasks. Each processor has a price and a variable indicating whether or not it has a communication buffer. Processors without communication buffers may not concurrently execute a task and communicate data with another PE. For each pair of tasks and processors, there is an execution time, a preemption time, and a memory load. *Execution time* is the amount of time a processor requires to carry out a task. *Preemption time* is the amount of time required to save a task's state before interrupting it with another task. *Memory load* is the amount of memory required by a task when executed on a processor. This variable accounts for instruction and data space.

**FPGA:** An *FPGA* is a PE which must be reconfigured between the execution of different types of tasks. FPGAs are divided into configurable logic blocks (CLBs), each of which is capable of being configured to compute a number of combinational and sequential logic functions. Each FPGA is described by its price, the number of CLBs it provides, and the amount of memory required to configure an individual CLB. For each pair of tasks and FPGAs, there is an execution time and a CLB requirement. The *CLB requirement* is the number of CLBs filled by a configuration which is capable of carrying out a given task.

**Communication resource:** *Communication resources* connect different PEs to each other. Each communication resource is described by a price, a unit transmission time, and a contact count. *Unit transmission time* is the amount of time required to transmit a unit of data. *Contact count* is the number of PEs a communication resource is capable of connecting together.

**Optimization terms:** CORDS optimizes the following components of an architecture in attempts to meet an embedded system specification while minimizing price: PE allocation, communication resource allocation, task assignments, communication resource connectivities, communication event assignments, and event schedules. A *PE allocation* lists the number of each type of general-purpose processor and FPGA in an architecture. A *communication resource allocation* lists the number of each type of communication resource present in an architecture. *Task assignments* denote the PE upon each task is executed. *Communication resource connectivities* denote the PEs to which each communication resource is connected. *Communication event assignments* denote the communication resource through which each communication event is transmitted. In addition, CORDS generates a schedule for the tasks assigned to each PE and the communication events assigned to each communication resource.

When tasks are carried out on general-purpose processors, memory is required for instructions and data. Similarly, FPGAs require memory to store configurations and data. CORDS accounts for these requirements when computing an architecture's price.

An architecture's *cost set* characterizes the quality of the architecture. A cost set contains the number of tasks which could not be scheduled at any time, the number of communication events which could not be scheduled at any time, the degree to which the specification's task deadlines were violated, the degree to which CLBs were over-used in FPGAs , and the price of the architecture.

CORDS uses an evolutionary algorithm to optimize resource allocations, task assignments, and communication resource connectivities. This evolutionary framework is described in Section 4. The scheduling algorithm used by CORDS is described in Section 3.

# 3 Scheduling

In this section, we describe the scheduling algorithm used in CORDS. When the scheduling algorithm is invoked, CORDS has already determined PE allocations, communication link allocations, task assignments, and communication link connectivities. Thus, it is only necessary to determine the time at which each task is executed, the communication resource to which each communication event is assigned, and the time at which each communication event occurs. This problem is NP-complete for distributed systems [9], and is further complicated by consideration of reconfiguration, *i.e.*, on FPGAs, the amount of time a task requires depends on the previous and next task in the FPGA's schedule. We, therefore, resort to a heuristic scheduling algorithm. CORDS uses a preemptive static critical path scheduling algorithm with dynamic task reordering based on FPGA reconfiguration time. Reordering is dynamic but the resulting schedule is static, *i.e.*, the time at which each event is carried out is computed by

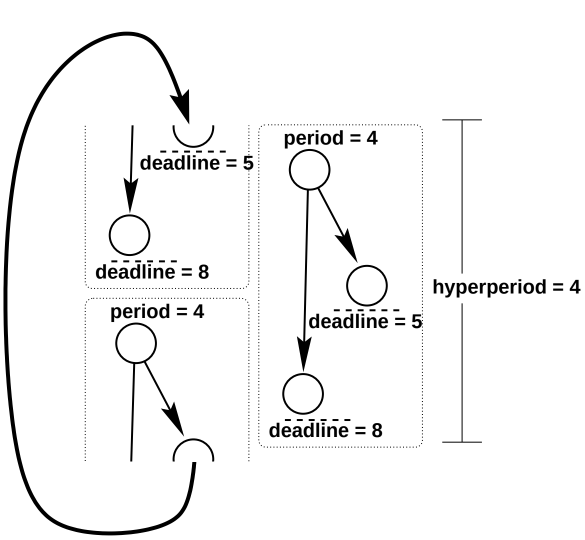


Figure 2: Period less than deadline

CORDS to determine whether or not hard deadlines are met by the schedule. Such guarantees are not possible, in general, when priorities are allowed to vary during the operation of the synthesized architecture.

CORDS targets multi-rate embedded systems. Such systems consist of multiple periodic task graphs which may have different periods. The *hyperperiod* of a system is the least common multiple of the periods of the task graphs in the system. A multi-rate schedule is valid if and only if all deadlines are met and each task graph is repeatedly executed until the hyperperiod has elapsed [10]. CORDS ensures validity of schedules by scheduling copies of task graphs until the hyperperiod has been reached. It is, therefore, possible to have multiple task copies in the schedule for a single task specified in the input task graphs.

Task graphs may have periods which are less than the maximum deadline in the task graph. This makes it possible for the execution of multiple instances of the same task graph to overlap in time. Handling this case complicates scheduling. Fig. 2 shows a system containing one task graph which has a period of four although the highest deadline within it is eight. As indicated by the bold arrow in Fig. 2, it is possible for tasks to be scheduled across the boundary between the system hyperperiod and time zero. In CORDS, all schedules are implicitly cyclic. Suppose $t_1$ and $t_2$ are times. From a resource's point of view, if $t_1 \bmod hyperperiod = t_2 \bmod hyperperiod$, then $t_1 = t_2$. Actual times are used when comparing task finish times with task deadlines, however.

CORDS uses a static, critical path based metric, called slack, to produce a preliminary order for tasks. Assuming worst-case communication and reconfiguration times, a task's *slack* is the amount of time its execution can be delayed, from its earliest possible execution time, without causing any other tasks to miss their deadlines. Slack is computed by finding the difference between the latest finish time and earliest finish time for each task. Earliest finish times are computed by conducting a topological search of the task graph, starting from the node with no incoming edges, and assuming worst-case reconfiguration times for all tasks which are assigned to FPGAs. Each communication event duration is assumed to be the duration required by the slowest communication resource connecting the PEs to which the communicating tasks are assigned. It is commonly assumed, in distributed computing research, that communication between tasks assigned to the same PE is effectively instantaneous, relative to inter-PE communication. We also make this assumption. Latest finish times are computed by conducting a backward topological search of the task graph, starting from the nodes which have deadlines, and assuming worst-case reconfiguration times for all tasks which are assigned to FPGAs. Slack is static, *i.e.*, it is computed before scheduling begins and is not adjusted during scheduling. Slack computation, therefore, takes $\mathcal{O}\left(edges + tasks\right)$ time.

When the scheduling algorithm begins, all start tasks, *i.e.*, those tasks with no incoming edges, are entered into a pending list which is sorted in order of decreasing slack. Ties are broken by ordering the equivalent-slack tasks by increasing task graph copy number. Tasks are sequentially removed from the end of the pending list and scheduled. After a task is scheduled, its children are checked to determine whether all of their parents have been scheduled, satisfying data dependencies. Children which satisfy this test are entered into the pending list, reconfiguration delays are recalculated, and the pending list is sorted again before scheduling the next task.

*Reconfiguration delay* is the amount of reconfiguration time an FPGA would require to change from the configuration capable of executing the task most recently scheduled on the FPGA, to a configuration capable of executing another task. Suppose two tasks, $f$ and $g$, are both assigned to the same FPGA. If $f$ was the task most recently scheduled to the FPGA, then the FPGA is configured to execute a task of $f$'s type. If $g$ is the same type of task as $f$, then the FPGA need not be reconfigured between their execution, otherwise the FPGA needs to be reconfigured. Some FPGAs are capable of partial reconfiguration. For such FPGAs, the reconfiguration time for a pair of configurations depends on the number of CLBs used by each configuration, in addition to the similarity between the configurations.

There is a reconfiguration delay associated with every task which is assigned to an FPGA. The reconfiguration delay for a task of type $h$, assigned to an FPGA whose most recently scheduled task was also of type $h$, is zero. Reconfiguration delay is dynamically adjusted during the execution of the scheduling algorithm. Every time a task is removed from the pending list, a dynamic check is first made to determine whether or not executing another task first would be likely to reduce total FPGA reconfiguration time without causing deadlines to be missed. *Dynamic priority* is defined to be the sum of a task's negative slack and its negative reconfiguration delay, *i.e.*, (dynamic priority) $=$ $-$(slack) $-$ (reconfiguration delay). It may seem counter-intuitive to increase the dynamic priority of tasks with low reconfiguration times. However, this encourages similar tasks to be scheduled on an FPGA consecutively, reducing the amount of reconfiguration necessary. If the task, $u$, which was just removed from the pending list is assigned to an FPGA, then the dynamic priorities of all the other tasks in the pending list which are assigned to the same FPGA as $u$ are compared with $u$'s dynamic priority. If another task has a higher dynamic priority than $u$, it is removed from the pending list and scheduled immediately, after which time $u$ is again considered for scheduling. When two tasks have equal dynamic priorities, the task belonging to the earlier copy of a task graph is scheduled first.

Suppose there are two tasks, $l$ and $m$, in the pending list and assigned to the same FPGA. Suppose $l$ has a slack of 4 ms and a reconfiguration delay of 5 ms, and task $m$ has a slack of 8 ms. Suppose the task most recently scheduled to $m$'s FPGA was of the same type as $m$. Therefore, $m$'s reconfiguration delay is 0 ms. Task $l$ has a dynamic priority of $-$(4 ms) $-$(5 ms) $=$ $-$(9 ms). Task $m$ has a dynamic priority of $-$(8 ms) $-$(0 ms) $=$ $-$(8 ms). Thus, although task $l$ has less slack than task $m$, *i.e.*, it lies along a more critical path, task $m$ will be scheduled first. Scheduling $m$ before another task is scheduled to its FPGA is likely to reduce the reconfiguration time required. Consider, next, a comparison between task $m$ and task $n$, which has a slack of 1 ms, a reconfiguration delay of 5 ms, and a resulting dynamic priority of $-(1\text{ ms}) - (5\text{ ms}) = -(6\text{ ms})$. Although scheduling $m$ first has the potential to reduce the reconfiguration time of $m$'s FPGA, $n$'s extremely low slack makes it inadvisable to take a chance on delaying $n$. Therefore, $n$ will be scheduled before $m$.

The first step of scheduling an individual task, $t$, is to schedule all of its incoming edges, *i.e.*, communication events. Each edge is scheduled on a communication resource connecting the PE to which $t$ is assigned and the PE to which $t$'s parent is assigned. When multiple communication resources are available, CORDS selects the communication resource upon which the communication event will complete at the earliest time. If either of the communicating PEs do not have communication buffers, CORDS schedules the communication event to the unbuffered PEs, as well. If there are no communication resources connecting the PEs involved, CORDS notes this in the architecture's cost set (see Section 2).

Every time a task is scheduled on a processor, CORDS determines whether or not preemption is likely to result in an improved schedule. CORDS first tentatively schedules a task, $t$, to the earliest time slot on its processor, which starts after its incoming edges have completed execution and has a long enough duration to accommodate the task. CORDS then checks to see whether preempting the task, $p$, which is scheduled to the same processor as $t$, previous and adjacent to $t$, would result in a *net improvement*, where net improvement is defined as the (increase in finish time for $p$) + (decrease in finish time for $t$) − ($t$ slack) + ($p$ slack). If preemption results in a net improvement, there is enough time available on the general-purpose processor before the next scheduled task, and preempting $p$ does not change the times at which it communicates with tasks on other PEs, then the preemption is carried out.

# 4 Evolutionary Algorithm

In this section, we describe the evolutionary algorithm used by CORDS to optimize PE allocations, communication resource allocations, task assignments, and communication resource connectivities. This algorithm shares some properties with parallel recombinative simulated annealing algorithms [11], and multiobjective genetic algorithms [12]. CORDS maintains a pool of architectures. A *generation* is a discrete unit of time. In every generation, architectures reproduce. The new architectures mutate and trade information with each other. The architectures are then ranked, relative to each other. Poor-quality architectures are eliminated until the number of remaining architectures is the same as the number of architectures at the beginning of the generation. When a user-specified number of generations has passed without improvement to its architectures, CORDS halts and displays the best architecture it has ever encountered.

CORDS maintains a global temperature which decreases every time a user-specified number of generations passes without improvement to its architectures. As described in remainder of this section, CORDS uses the global temperature to control the greediness of its optimization algorithm.

**Architecture clusters:** If CORDS allowed architectures with different PE allocations to trade task assignments or communication resource connectivity information, invalid architectures might result. Consider what would happen if an architecture which contains FPGAs traded part of its task assignment with an architecture which has no FPGAs. After the trade, some of the tasks in the architecture with no FPGAs might be assigned to FPGAs. CORDS prevents this problem from occurring by clustering together architectures with equivalent PE and communication resource allocations. When task assignment or communication resource connectivity information is traded, the trade occurs between architectures in the same cluster. When PE allocation and communication resource allocation information are traded, the trade occurs between entire clusters, modifying every architecture in the involved clusters.

In every generation, architectures are randomly selected and copied until the number of architectures has been doubled. Cluster reproduction is similar. However, it occurs less frequently than architecture reproduction. The newly produced architectures are all modified by, alternately, mutation and information trading.

**Mutation:** Mutation makes randomized changes to an architecture or cluster. When an architecture mutates, CORDS first determines whether the task assignment or communication resource connectivity will mutate. A random variable, $w$, between zero and the average number of contacts on a communication resource, is selected. If $w$ is greater than one, the task assignment mutates, otherwise the communication resource connectivity mutates. Letting *t_count* be the number of tasks in the embedded system specification multiplied by the global temperature, task assignment mutation causes a randomly selected set of *t_count* tasks to be reassigned to randomly selected PEs. Letting *c_count* be the number of communication resources in the architecture multiplied by the global temperature, communication resource connectivity mutation causes *c_count* communication resources to disconnect all of their contacts and randomly reconnect them to PEs.

When a cluster mutates, CORDS first determines whether the PE allocation or communication resource allocation will mutate. A random variable, $x$, between zero and the average number of contacts on a communication resource, is selected. If $x$ is greater than one, the PE allocation mutates, otherwise the communication resource allocation mutates. Cluster mutation may cause an instance of a randomly selected PE type to be added to all the architectures in the cluster or it may cause a randomly selected PE to be removed from all the architectures in the cluster. The probability of an additive mutation is equivalent to the global temperature maintained by CORDS, which ranges from one to zero. Thus, early in an optimization run, when the global temperature is high, the number of PEs allocated is more likely to increase than to decrease. Later in the run, the number of PEs is more likely to decrease. We have empirically determined that associating PE addition probability with global temperature results in the production of better solutions than using a fixed PE addition probability. All of the architectures in the cluster randomly change the parts of their task assignments and communication resource connectivities which depend on the lost PE such that none of the tasks or communication resources depend on the lost PE. Communication resource allocation mutation is analogous to PE allocation mutation.

**Information trading:** CORDS uses an evolutionary algorithm which is based on two types of genetic algorithm [11], [12]. However, each of these algorithms has problems dealing with the optimization of multi-dimensional information. Below, we describe these problems and explain how CORDS avoids them.

In a genetic algorithm, each architecture is represented by a string, *i.e.*, a linear array, of values. Genetic algorithms trade information by conducting *cross-over* between strings, *i.e.*, two strings are cut at the same random offset from their first values and the portions following the cut are swapped. Unfortunately, many real problems cannot be cleanly represented by a string of values. Conventionally, researchers who use genetic algorithms impose a linear order on the information they optimize. However, there are problems with this approach. For evolutionary algorithms to operate efficiently, it is necessary for

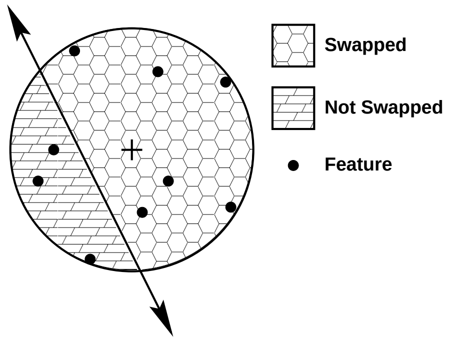


Figure 3: Feature trade selection

their information trading operation to preserve *locality*, *i.e.*, information trading should separate information describing closely related features of an architecture less frequently than it separates information describing disparate features [13].

Imposing a linear order on multi-dimensional information is guaranteed to disrupt locality. Consider the problem of representing $n$-dimensional vectors in a system where locality is inversely proportional to Euclidean distance. Imposing a linear optimal locality order on this information is equivalent to the $n$-dimensional Euclidean traveling salesman problem, which is NP-complete. Thus, one is generally forced to resort to approximation algorithms. Even if it were possible to get an optimal solution to this problem, in general, reducing the dimensionality of the information from $n$ to one results is a distortion of space and the disruption of locality. Although CORDS preserves locality when trading information between architectures and clusters, it does not disrupt the locality of $n$-dimensional features. CORDS never imposes a linear order on its information.

CORDS represents each feature as a vector within a hyper-sphere. Fig. 3 illustrates the selection of features to trade. Each feature is represented by a dot and lies within a hyper-sphere (a circle in this two-dimensional example). When architectures trade information, CORDS determines which portions of the information to trade by dividing the hyper-space with a randomly oriented and randomly located hyper-plane (a line in this example). The features associated with vectors on one side of the plane are traded (those to the upper-right in Fig. 3). Features on the other side of the hyper-plane remain unchanged. Although no linear order is ever imposed, the probability of a pair of features being separated by an information trade is inversely proportional to the Euclidean distance between them.

A PE's hyper-space vector is determined by its price, execution time, and configuration time (zero for general-purpose processors). A communication resource's hyper-space vector is determined by its price, unit transmission time, and number of contacts. A task graph's hyper-space vector is determined by its period and the maximum deadline in the task graph. Each set of vectors is pre-processed such that the set is incident on and centered in a unit $n$-dimensional sphere. Thus, determining the features to be traded between architectures during information sharing is reduced to selecting a randomly oriented hyper-plane with a random offset between the most distant features along the normal to the hyper-plane, and determining which side of the hyper-plane each feature lies on.

When architectures trade information, CORDS first determines whether task assignment or communication resource connectivity information will be traded. A random variable, $y$, between zero and the average number of contacts on a communication resource, is selected. If $y$ is greater than one, task assignments trade information, otherwise communication resource connectivities trade information. Task assignment information trading causes a randomized but locality-preserving set of task assignments (selected as described above) to be traded between two architectures. Communication resource connectivity information trading is analogous to task assignment information trading.

When clusters trade information, CORDS first determines whether task allocation or communication resource allocation information will be traded. A random variable, $z$, between zero and the average number of contacts on a communication resource, is selected. If $z$ is greater than one, task allocations trade information, otherwise communication resource allocations trade information. Task allocation information trading and communication resource allocation information trading are analogous to task assignment information trading.

**Architecture cache:** Every time an architecture is changed, it is necessary to determine its new cost. Rescheduling an architecture each time it changes would be the most straightforward approach. Scheduling, however, is the most time-consuming operation undertaken by CORDS. CORDS maintains a least-recently used cache of architecture cost sets to prevent the re-evaluation of architectures after every modification. In CORDS, scheduling is deterministic. Therefore, for any PE allocation, communication resource allocation, task assignment, and communication resource connectivity, there exists exactly one set of system cost set. Thus, any architecture is characterized by a small amount of information, relative to the amount of information contained in a full schedule.

Sometimes, architecture mutation and inter-architecture communication produce an architecture which was previously scheduled. In these cases, the architecture's cost set is retrieved from CORDS's cache, making it unnecessary to carry out scheduling. We use a cache containing seven times as many entries as the number of architectures CORDS operates on. Our experimental results indicate that the cache is hit approximately 60 % of the time. Its use consistently cuts run-time in half.

**Ranking:** After architecture reproduction, mutation, and information trading, CORDS ranks the architectures in each cluster relative to each other. Architecture $p$ dominates architecture $q$ if all the members of $p$'s cost set (defined in Section 2) are less than or equal to the corresponding members in $q$'s cost set and the two cost sets are not equal. An architecture's rank is the number of other architectures that do not dominate it. Fig. 4 illustrates ranking of a set of architectures whose cost sets contain only two costs, price and deadline violation. Each oval contains a letter associated with an architecture, and a number indicating the architecture's rank. Thus, architecture **B** has a rank of two because there are two other architectures which do not dominate it, *i.e.*, there are two architectures which are not to the lower-left of architecture **B**. This method of ranking is called Pareto-ranking and it has a number of interesting properties elaborated on in the literature [12], [14].

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

**Boltzmann trials:** Given two ranks, $J$ and $K$, and the global temperature, $T$, a Boltzmann trial preserves the architecture associated with $J$ and eliminates the architecture associated with $K$ with probability

$$\left(1 + e^{(J-K)/T}\right)^{-1}$$

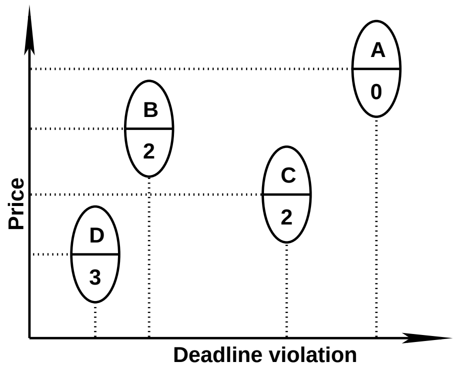


Figure 4: Architecture ranking

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

# 5 Experimental Results

We use a set of task graphs, processors, and communication resources produced by TGFF [15] based on information found in trade journals [16], data sheets [2], and discussions with a representative of Xilinx Corporation. The same optimization parameters, *e.g.*, solution pool size, are used by CORDS for all of the examples within each of the following tables. Each of our 35 examples contains five task graphs. Each task graph contains an average of 20 tasks. There are 15 types of tasks, five types of processors, ten types of FPGAs, and five types of communication resources. The tightness of the deadlines differs from example to example. The depth of a task is the number of tasks on the longest path between it and the start task. The tasks in Example A1 which have deadlines, have an average deadline of 70 ms times the depth of the task. In each subsequent example, the average task deadline increases by 450 ms, multiplied by the depth of the task. Thus, the average task deadline in Example A5 is 1.87 s times the depth of the task. The seed given to TGFF's random number generator for each example is equivalent to that example's number, *e.g.*, TGFF is seeded with three for Example A3 and Example C3. The processors have an average price of $20, with a variability of $10, *i.e.*, processor prices range from $10 to $30. Tasks have an average execution time of 300 ms, with a variability of 285 ms, on the processors. Preemption time has an average of 150 $\mu$s with a variability of 140 $\mu$s. Execution time and preemption time are both inversely correlated to processor cost. Tasks executed on processors require an average of 40 kilobytes of memory, with a variability of 28 kilobytes. 9.7 % of processors lack communication buffers. Communication resources have an average price of $20 with a variability of $10. Communication time is 50 $\mu$s per kilobyte, with a variability of 40 $\mu$s per kilobyte. Communication events have an average size of 42 kilobytes with a variability of 40 kilobytes. Memory has a price $3.17 per megabyte, with a minimum unit size of 256 kilobytes.

For FPGAs, average task execution time is 20 ms with a variability of 19 ms. The average task execution time on FPGAs, relative to the average task execution time on processors, is approximately $\frac{1}{12}$ as high, a conservative estimate based upon the literature, in which speedups of 20-100 times are frequently reported. The average memory load of a task executed on an FPGA is 42 kilobytes with a variability of 28 kilobytes, in addition to the memory required to hold the CLB contents for the task. XC6200 family parts have price ranging from $200 to $400 [1]. The average number of CLBs required by a task implemented on a 6200 family FPGA is 2000, with a variability of 1970. Task reconfiguration time for the 6200 family is 5 ns per CLB. The XC6216 provides 4096 CLBs. The XC6264 provides 16386 CLBs.

Table 1: Resource modification experiments

| Example | Price or (deadline viol.) w. processors only | Price or (deadline viol.) w. processors and XC4000s | Price or (deadline viol.) w. processors and XC6200s |
|---|---|---|---|
| A1 | (unsched.) | $ 162 | $ 360 |
| A2 | (65.32 %) | $ 32 | $ 175 |
| A3 | (1.47 %) | $ 45 | $ 226 |
| A4 | (3.48 %) | $ 66 | $ 346 |
| A5 | (0.15 %) | $ 61 | $ 503 |
| A6 | $ 89 | $ 39 | $ 65 |
| A7 | $ 108 | $ 43 | $ 91 |
| A8 | $ 60 | $ 23 | $ 32 |
| A9 | $ 116 | $ 20 | $ 117 |
| A10 | $ 38 | $ 29 | $ 38 |
| A11 | $ 54 | $ 54 | $ 62 |
| A12 | $ 16 | $ 16 | $ 16 |
| A13 | $ 63 | $ 54 | $ 70 |
| A14 | $ 34 | $ 36 | $ 34 |
| A15 | $ 52 | $ 31 | $ 52 |

Eight XC4000 series parts are used in the examples. Their price ranges from approximately $30 to $400. Their CLB counts range from 100 to 1024. XC4000 series members do not support partial reconfiguration, *i.e.*, each reconfiguration requires the entire FPGA to be programmed. Therefore, task CLB counts only affect the total memory requirements of the tasks, not their XC4000 series FPGA's reconfiguration time.

The examples are available via anonymous FTP at ftp://ftp.ee.princeton.edu/pub/dickrp/CORDS. For each example, CORDS required less than 15 CPU minutes on a 200 MHz Pentium Pro processor. *Deadline violation* is the amount by which an architecture overran its deadlines, as a percentage of the sum of the maximum deadlines in each copy of the task graph. When forced to use processors only, CORDS was unable to produce a solution for Example A1 in which all tasks were scheduled within the hyperperiod, even when deadline violations were allowed. The second column in Table 1 shows the best architectures produced by CORDS when it uses only processors. For high example numbers, in which deadlines are lax, processors alone are sufficient to produce valid architectures. For the examples with tighter deadlines, CORDS is able to synthesize valid architectures by using a combination of processors and FPGAs. The third column shows the best architectures produced by CORDS when using processors and XC4000 series FPGAs. The fourth column shows the best architectures produced when using processors and XC6200 family FPGAs.

In general, by using XC4000 series and XC6200 fam-

[1]The XC6200 family is a low-volume and high-cost part used primarily for research. Xilinx Corporation is, however, integrating many of the features present in the XC6200 family into a high-volume part. The prices given here are rounded to the nearest $100 at the request of a representative of Xilinx Corporation.

Table 2: Conventional vs. rapid reconfiguration FPGAs

| Example | Price or (deadline viol.) w. processors and XC4000s | Price or (deadline viol.) w. processors and XC6200s |
|---|---|---|
| B1 | $ 72 | $ 589 |
| B2 | (1.05 %) | $ 178 |
| B3 | $ 27 | $ 228 |
| B4 | (6.80 %) | $ 647 |
| B5 | $ 62 | $ 504 |

Table 3: Dynamic priority experiments for XC4000 series

| Example | Price or (deadline viol.) w.o. dynamic priority | Price or (deadline viol.) w. dynamic priority | Price decrease |
|---|---|---|---|
| C1 | $ 48 | $ 49 | -2.08 % |
| C2 | $ 78 | $ 64 | 17.95 % |
| C3 | $ 56 | $ 25 | 55.36 % |
| C4 | (0.02 %) | $ 133 | n.a. |
| C5 | $ 90 | $ 56 | 37.78 % |
| C6 | $ 32 | $ 33 | -3.12 % |
| C7 | $ 81 | $ 77 | 4.94 % |
| C8 | $ 27 | $ 10 | 62.96 % |
| C9 | $ 90 | $ 51 | 43.33 % |
| C10 | $ 61 | $ 55 | 9.84 % |
| C11 | $ 62 | $ 67 | -8.06 % |
| C12 | $ 25 | $ 10 | 60.00 % |
| C13 | $ 70 | $ 47 | 32.86 % |
| C14 | $ 72 | $ 34 | 52.78 % |
| C15 | $ 69 | $ 24 | 65.22 % |

ily parts, CORDS was able to produce valid architectures for a number of examples that could not be solved using only processors. Using XC4000 series FPGAs typically resulted in a reduction of price, when compared to architectures using only processors. As a result of the high price of 6200 family parts, architectures containing processors and 6200 family parts are generally more expensive than architectures containing processors and 4000 series parts. However, in some cases the more rapid reconfiguration of 6200 family parts allows the satisfaction of specifications which are not met using only processors and 4000 series parts. This is especially true for examples in which reconfiguration time is similar to computation time. The examples shown in Table 2 differ from those in Table 1 in three ways: the amount of time spent executing tasks and communicating data are reduced such that reconfiguration time and execution time for tasks associated with a 4000 series part are similar, there are five task types instead of fifteen, and tasks with deadlines have an average deadline of 32 ms times the depth of the task. In general, when CORDS produces a valid architecture using either processors and 4000 series parts, or processors and 6200 family parts, the architecture composed of processors and 4000 series parts is less expensive. However, a design using processors and 6200 family parts are sometimes capable of meeting specifications which are not met using processors and 4000 series parts.

The examples shown in Table 3 are different from those shown in Table 1 in one way: the tasks in examples in Table 3, which have deadlines, have an average deadline of 310 ms times the depth of the task. Table 3 compares the quality of the architectures produced by CORDS running in two different modes. The second column shows architectures produced when CORDS only considers static task slack during scheduling. The third column shows the architectures produced when CORDS reorders tasks based on their dynamic priorities. In one out of the fifteen examples, reordering based on dynamic task priorities allowed CORDS to produce a valid architecture when scheduling based on static priorities alone produced no architectures which met their deadlines. Reordering based on dynamic priority improved architecture price in 11 of the examples. For three examples, reordering resulted in a slight increase in price. However, for the 14 examples for which reordering resulted in a change in price, the average price reduction was approximately 30 %.

## 6 Conclusions

CORDS is the first co-synthesis system to consider the effects of dynamically reconfiguring FPGAs during the operation of an embedded system, and reduce the amount of FPGA reconfiguration time. Experimental results indicate that time multiplexing tasks on dynamically reconfigurable FPGAs has the potential to decrease system price and allow otherwise infeasible specifications to be met.

## References


[1] "Altera ARC-PCI reconfigurable computing platform." http://www.altera.com/html/new/pressrel/pr_arc-pci.html.

[2] "Xilinx part information." http://www.xilinx.com/partinfo/.

[3] D. Galloway, "The transmogrifier C hardware description language and compiler for FPGAs," in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pp. 136–144, Apr. 1995.

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

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

[7] 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.

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

[9] 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.

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

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

[12] 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.

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

[14] R. P. Dick and N. K. Jha, "MOGAC: A multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems," in *Proc. Int. Conf. Computer-Aided Design*, pp. 522–529, Nov. 1997.

[15] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: Task graphs for free," in *Proc. Int. Workshop Hardware/Software Codesign*, Mar. 1998.

[16] "Computer design." Product trends sections of vol. 35: n. 2, 6, 8, 9, vol. 36: n. 1, 9, and vol. 37: n. 1–3.