MOCSYN: Multiobjective Core-Based Single-Chip System Synthesis

Robert P. Dick and Niraj K. Jha

Department of Electrical Engineering
Princeton University
Princeton, New Jersey 08544

Abstract

In this paper, we present a system synthesis algorithm, called MOCSYN, which partitions and schedules embedded system specifications to intellectual property cores in an integrated circuit. Given a system specification consisting of multiple periodic task graphs as well as a database of core and integrated circuit characteristics, MOCSYN synthesizes real-time heterogeneous single-chip hardware-software architectures using an adaptive multiobjective genetic algorithm that is designed to escape local minima. The use of multiobjective optimization allows a single system synthesis run to produce multiple designs which trade off different architectural features. Integrated circuit price, power consumption, and area are optimized under hard real-time constraints. MOCSYN differs from previous work by considering problems unique to single-chip systems. It solves the problem of providing clock signals to cores composing a system-on-a-chip. It produces a bus structure which balances ease of layout with the reduction of bus contention. In addition, it carries out floorplan block placement within its inner loop allowing accurate estimation of global communication delays and power consumption.

1 Introduction

The process of concurrently defining the hardware and software portions of an embedded system while considering dependencies between the two is called hardware-software co-design [1]–[4]. Time pressure makes it difficult for an engineer to explore the numerous alternative designs which have the potential to meet a given set of specifications. As a result, an engineer frequently selects a conservative architecture after little experimentation, resulting in a needlessly expensive system. The majority of past research in the area of hardware-software co-design has attempted to ease the process of design exploration, allowing an engineer to examine a number of alternative implementations in the short time available. Co-synthesis is the automation of the generation of an embedded system architecture. Given an embedded system specification, a co-synthesis system selects hardware and software processing elements, devices upon which tasks execute. In addition, the system assigns each task to a processing element and ensures that tasks which need to communicate with each other are capable of doing so. Finally, schedules are produced for tasks and communication, such that real-time constraints are met [5], [6]. Co-synthesis systems generate feasible, low-cost architectures without designer intervention.

It is possible to implement some embedded systems using a single integrated circuit (IC), thereby reducing cost and improving performance [7]. Economic and time pressures frequently make it impractical to do an in-house design for each component in a single-chip system. Fortunately, the number of intellectual property (IP) cores available from the industry has dramatically increased in the past year. Companies like Alta Group, VAutomatic Inc., Virtual Chips, and Intronic offer a wide range of IP cores, *e.g.*, protocol processors, general-purpose processors, micro-controllers, digital signal processors, memory, and Data Encryption Standard engines.

Optimal co-synthesis is an intractable problem. Allocation/assignment and scheduling are each NP-complete for distributed systems [8]. As a result, it is not surprising that all co-synthesis systems which employ optimal mixed integer linear programming [9], [10] and exhaustive exploration [11] can only be applied to small instances of the co-synthesis problem. There are a number of co-synthesis algorithms which make the solution of large problem instances tractable. To achieve reasonable run-time, however, it is necessary to sacrifice the guarantee of solution optimality. Iterative improvement algorithms start with a complete solution and make local changes to it in an attempt to improve the solution's cost [5], [12], [13]. Constructive algorithms build a system by incrementally adding components to it [14], [15]. Simulated annealing algorithms have been successfully used to partition hardware-software systems [16]. Genetic algorithms have been applied to the hardware/software partitioning problem [17]. A multiobjective genetic algorithm was applied to the more general co-synthesis problem [18].

MOCSYN, which stands for **m**ulti**o**bjective **c**ore-based single-chip system **syn**thesis, differs from past work by considering a number of issues unique to core-based single-chip systems. MOCSYN determines the clock frequencies supplied to different cores. It generates priority-based bus structure of arbitrary topology, balancing ease of routing and bus contention minimization. In addition, it conducts floorplan block placement [19] within its inner loop, allowing estimates of global wiring delays and power consumption to be used during scheduling and cost calculation. Experimental results demonstrate that a global bus is, in general, inferior to the use of priority-based arbitrary bus topologies. Conducting block placement in the inner loop frequently results in an improvement in solution quality when compared with worst-case or best-case communication delay estimates.

2 Data structures and definitions

In this section, we describe the data structures used in MOCSYN and define basic co-synthesis terms.

Cost: A cost is a variable that a synthesis system attempts to minimize. Price, power dissipation, and IC area are examples of costs.

Task graph: As shown in Fig. 1, 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. A task with an incoming edge, *i.e.*, a task toward which an edge's arrow points, may execute only after receiving data from the other task to which the edge is connected. Thus, in Fig. 1, **DCT** is data-dependent on **NEG**. The *period* of a task graph is the amount of

This work was supported in part by an NSF Graduate Fellowship and in part by NSF under Grant No. MIP-9729441.

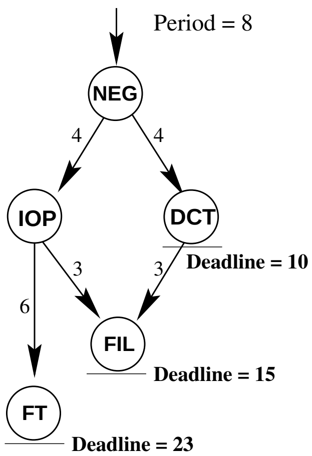

Figure 1: Task graph

time between the earliest start times of its consecutive executions. A node without any outgoing edges is a *sink node*. A *deadline*, the time by which the task associated with the node must complete its execution, exists for every sink node. Other nodes may also have deadlines associated with them.

Multi-rate: A multi-rate system contains task graphs with different periods. It was shown in [20] that, in order to guarantee a valid schedule for a multi-rate system, each task graph must repeatedly execute until the least common multiple (LCM) of the periods, termed as the hyperperiod, of all the task graphs in the system has elapsed.

Core: A core executes one or more tasks. Multiple cores may be located on the same IC, upon which multiple tasks may execute simultaneously. The following information establishes the relationship between tasks and cores:

- A two-dimensional array indicating the relative worst-case execution time of each task on each core.
- A two-dimensional array indicating the relative average power dissipation of each task on each core.
- A two-dimensional array indicating the core types upon which each task type may be executed.

In addition, each core has a price which corresponds to royalties paid to the IP producer on a per-use basis. This price is zero for royalty-free IP cores. If IP has a one-time fee instead of, or in addition to, a per-use royalty, the price is equivalent to the one-time fee divided by the expected production volume. Each core has a width, a height, a maximum clock frequency, a variable indicating whether or not its communication is buffered, and an energy consumption per cycle dedicated to communication.

Core allocation: The information denoting the number of cores of each type present in an IC.

Link: A link is a potential point-to-point contact between a pair of cores. Any given link may be merged with other links during bus formation (see Section 3.7), thereby ceasing to exist.

Task assignment: The information denoting the core upon which a given task is executed.

Architecture: A set of allocation, assignment, and scheduling information which defines an embedded system.

3 The MOCSYN algorithm

In this section, we give an overview of MOCSYN (see Section 3.1) and describe the algorithms of which it is composed. Section 3.2 describes the algorithm used to determine the clock frequency used for each type of core. Section 3.3 describes the initialization of MOCSYN's data structures. Section 3.4 describes how core allocation and task assignment are determined. Section 3.5 explains how links are prioritized. The block placement algorithm used by MOCSYN is described in Section 3.6. Bus topology generation is explained in Section 3.7. Scheduling is explained in Section 3.8. Finally, cost calculation is described in Section 3.9.

3.1 Algorithm overview

In this subsection, we give a high-level description of the MOCSYN algorithm. MOCSYN carries out the following tasks:

1. Determine a *clock frequency* for each core type, subject to trade-off between execution time and power consumption.
2. Determine the *allocation* of cores to use.
3. Determine the tasks to *assign* to each core, subject to trade-off between ease of routing and minimization of bus contention.
4. Determine a *bus structure* to use on the IC.
5. Derive a *block placement* for the cores, allowing an estimation of wire delay, wire power consumption, and silicon area.
6. *Assign* each communication event to a bus.
7. *Schedule* the tasks on the cores and the communication events on the communication links.

MOCSYN uses a genetic algorithm to optimize embedded system architectures. In general, genetic algorithms have the ability to escape local minima. They allow solutions to cooperatively share information with each other. Genetic algorithms are especially useful for simultaneously optimizing more than one cost. Conventional iterative improvement and simulated annealing algorithms maintain only one solution at a time. Most single-solution optimization algorithms collapse all costs into a single value with a weighted sum [21], [22]. Genetic algorithms maintain a pool of solutions which evolve in parallel. This allows solutions to be ranked relative to each other. Genetic algorithms are capable of true multiobjective optimization, exploring the *Pareto-optimal set* of solutions, *i.e.*, those solutions which are better than any other solution in at least one way. In a genetic algorithm, solutions are iteratively improved by *mutation*, randomized local changes to a solution, and *crossover*, during which information is shared between different solutions. In this paper, we concentrate on the unique problems encountered when designing single-chip core-based systems.

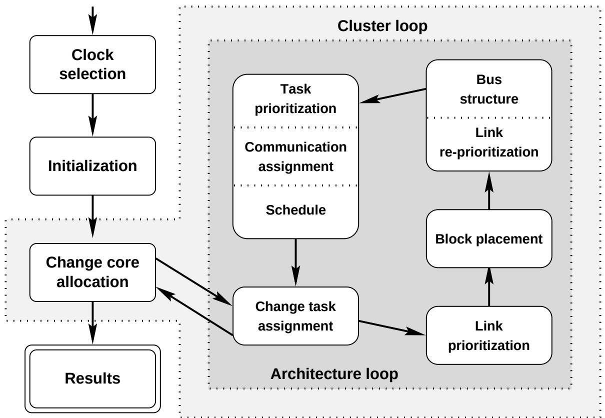

Figure 2: MOCSYN overview

An overview of the MOCSYN algorithm is shown in Fig. 2. Initially an optimal, but potentially slow, algorithm determines the clock frequency to provide to each core. Basic data structures are then initialized. MOCSYN is a hierarchical algorithm. It uses a genetic algorithm to improve the task assignments of individual architectures. More detailed information about this genetic optimization framework can be found in [23]. After this phase has been repeated an arbitrary (user-selectable) number of times, an attempt is made to improve the core allocation of a *cluster* of architectures, *i.e.*, a collection of architectures which share the same core allocation but may have different task assignments. Within the architecture optimization loop, a number of deterministic algorithms are used to concurrently evaluate the core allocation and task assignment of each architecture. First, a priority is assigned to each link, *i.e.*, the communication carried out between each pair of cores. These priorities are used to generate a block placement for the cores, ensuring that core pairs for which communication priority is high are located near each other. Links are re-prioritized based on global wiring delay information which is extracted from the block placement. A bus structure which trades off potential bus contention for ease of routing is produced. At this point, tasks are prioritized and a schedule is generated for the tasks assigned to each core. Communication events are concurrently assigned to, and scheduled on, busses. At the completion of each architecture optimization loop, changes are made to the task assignments in an attempt to improve them. At the completion of each cluster optimization loop, changes are made to the core allocations in an attempt to improve them. Each of the sub-algorithms noted in Fig. 2 is described in the following sections.

3.2 Clock selection

In this subsection, we discuss the problems associated with selecting a clock frequency for each core in an IC and describe the algorithm used in MOCSYN to solve these problems.

Communication between cores in an IC can be single-frequency synchronous, multi-frequency synchronous, or asynchronous [24], [25]. Single-frequency synchronous communication has the potential to keep communication overhead at a minimum. However, its use requires that all the cores which communicate with each other be clocked at the same frequency. When different cores have different maximum frequencies, all cores must be clocked at a frequency less than or equal to the maximum frequency of the slowest core. Thus, using a single-frequency synchronous communication protocol will generally force sacrifices in core speed. Multi-frequency synchronous communication allows cores with different clock periods to communicate with each other at a rate proportional to the LCM of the communicating core's periods. Unfortunately, when cores have different minimum periods and efforts are made to allow each core to run near its maximum frequency, the LCM of the periods of communicating cores can be significantly higher than the period of any individual core, *e.g.*, $\text{LCM}(5, 7) = 35$. This generally results in slow communication. A third option is asynchronous communication. Although it has a reputation for increasing communication overhead, we believe that it is the best available option for systems in which different cores are clocked at significantly different frequencies. Using asynchronous communication, communication speed is bounded only by the bus bandwidth and rate at which communicating cores can transmit and receive information. Using asynchronous communication has the additional advantage of making inter-core clock skew irrelevant. Past work provides a framework for automatically synthesizing asynchronous interface protocols [25].

Given that one is using asynchronous communication, the selection of clock frequencies for the cores comprising a single-chip system need not be constrained by communication considerations. However, there are a number of other problems which must be dealt with. Supplying each core with an arbitrary clock frequency would require a large number of frequency generators, *e.g.*, analog timers based on RC delay or crystal oscillation. These components are difficult to integrate with conventional CMOS IC processes. Using discrete components is a poor option because each additional external component increases the price and area of an embedded system. Thus, a clocking approach which requires only one frequency source but allows nearly arbitrary frequencies to be delivered to each core would be advantageous.

For each i between 1 and n, inclusive,
  there is an array, A_i, of size $Nmax$,
  which contains integers.

Each of these integers is the current denominator
  for the numerator equivalent to its index.

Optimize E for the current M's.

For all i's between 1 and n, inclusive, if $I_i = Imax_i$:
    j ranges from 1 to $Nmax$, inclusive
    Find the j for which $j/(A_{ij}+1)$ is maximal
    Increment A_{ij}
    Set $D_i \leftarrow A_{ij}$
    Set $N_i \leftarrow j$

Figure 3: Clock selection kernel

We use an approach in which a single external oscillator is used to supply a base frequency. A cyclic counter or interpolating clock synthesizer associated with each core is used to divide this frequency by an integer, in the case of a cyclic counter, or multiply the frequency by a rational number, in the case of an interpolating clock synthesizer [26]. A description of the clock selection algorithm used in MOCSYN follows, which is capable of dealing with interpolating clock synthesizers. As the cyclic counter clock selection problem is a special case of the interpolating clock synthesizer clock selection problem, the algorithm used in MOCSYN is capable of solving either problem.

Given: A maximum external clock frequency ($Emax$) and a maximum frequency associated with each of the n cores ($Imax_1, Imax_2, \ldots, Imax_n$).

Each core's clock frequency multiplier is a rational number, $M_i = N_i/D_i$, with a positive integer numerator N_i less than or equal to a user-supplied maximum, $Nmax$, and a positive integer denominator, D_i. A core's internal frequency, I_i, is equal to the external frequency (E) multiplied by its multiplier, M_i.

MOCSYN maximizes the average of the ratios of the core frequencies (I_i) to the core frequency maxima ($Imax_i$), *i.e.*,

$$\left(\sum_{i=1}^{n} I_i/Imax_i\right)/n$$

It is simple to determine an optimal external frequency (E) if the value of each multiplier (M_i) is known. Obviously, for an optimal E, $\exists i \in [1, n]$ such that $I_i = Imax_i$. Thus, one need only consider a small set of E's. $\forall i \in [1, n], E_i = Imax_i/M_i$. The $\max_{i=1}^{n} E_i$ for which $\nexists_{i=1}^{n} I_i > Imax_i$ is the optimal E for a given set of M's.

The only remaining problem is to determine an optimal set of M's. It is obvious that, for any given pair of $Imax$'s, $Imax_a$ and $Imax_b$, if $Imax_a \geq Imax_b$ then an optimal $M_a \geq M_b$. This observation allows the solution space of M's to be pruned.

Initially, all D's are equal to 1 and all N's are equal to $Nmax$. Therefore, all M's are equal to $Nmax$.

To maximize the average of core frequency to maximum frequency ratios, one need only repeatedly execute a simple algorithmic kernel, while keeping track of the best set of M's, until $E > Emax$. This kernel is shown in Fig. 3. Although, given a maximum highest internal clock frequency of $Imax_a$ and a minimum highest internal clock frequency of $Imax_b$, this algorithm takes $\mathcal{O}\left(n \cdot Nmax \cdot Imax_a/Imax_b\right)$ time, in practice it is fast (see Section 4).

Linear interpolating clock synthesizers are compatible with standard digital design tools and processes. Their use provides a significant advantage: one can distribute a base global clock frequency which is well below the maximum local clock frequencies, thereby reducing power consumption in the global clock distribution net. However, interpolating clock synthesizers are more complicated than cyclic counters. In addition, they are likely to require more area [26]. If one chooses to use cyclic clock division counters, instead of linear interpolating clock synthesizers, the same clock selection algorithm is used. However, $Nmax$ is set to 1.

3.3 Initialization

At the start of an optimization run, each cluster's core allocation is initialized. One of three initialization routines is randomly selected:

1. Add one core of a randomly selected type.
2. Add one core of each type.
3. Repeatedly add cores of random types until a random number (ranging from one to twice the number of core types) has been added.

MOCSYN ensures that there is at least one core capable of executing each type of task in the input task graphs. It checks each task and adds an appropriate core to the allocation if none of the cores currently in the allocation are capable of executing the task. Each architecture's tasks are assigned using the task assignment algorithm described in Section 3.4. The global temperature of the genetic algorithm is set to one. This temperature decreases during the run of the algorithm and is used to control the probability of making a core allocation or task assignment change which decreases the quality of a solution [23]. At the beginning of an optimization run, random changes are made. As time progresses, MOCSYN becomes greedier, making only those changes which result in an improvement in core allocation or task assignment quality. This property increases the probability that MOCSYN will escape local minima [27].

3.4 Core allocation and task assignment

Core allocations are optimized by MOCSYN's genetic algorithm. The allocations change via crossover and mutation of the data structure which maintains the core allocations for each cluster of architectures. When mutation occurs, a core is added or removed from the core allocation. The probability of adding a core is equivalent to MOCSYN's global temperature, which gradually changes from one to zero during the run of the algorithm. This results in allocations being more likely to increase during the start of a run and more likely to be pruned near the end of a run. After removing a core, MOCSYN ensures that there is at least one core capable of executing each type of task present in the input task graphs using the method described in Section 3.3.

Core allocation crossover swaps portions of the core allocations of two clusters with each other. MOCSYN uses a novel method for selecting portions of the core allocations to be swapped. The probability of the allocations of two types of cores remaining together during a crossover, *i.e.*, the probability that both are swapped or that both are not swapped, is proportional to the similarity between the data describing the core types, *e.g.*, prices, execution time vectors, and power consumption vectors.

Task assignment mutation causes tasks to be reassigned to different cores. A task graph is randomly selected and a number of tasks within the graph are randomly selected for reassignment. The number of tasks to be reassigned is equivalent to the number of tasks in the task graph multiplied by the global temperature. For each task selected, a new core assignment is determined. A core's Pareto-rank is the number of other cores for which at least one property is inferior. First the properties of each core, when executing the given task, are used to determine the Pareto-ranks of all the cores capable of executing the task. Execution time, energy consumption, core area, and weight, a measure of the time required to execute the tasks already assigned to a core, are used in the Pareto-ranking process. An array of cores is sorted in order of increasing Pareto-rank. The core to which the selected task is re-assigned is determined by indexing into the sorted array by $\left\lfloor (1 - \sqrt{flat\_rand}) \cdot array\_size \right\rfloor$ where flat_rand is a uniform random variable ranging from [0, 1) and array_size is the size of the sorted array of cores.

Task assignment crossover causes the task assignments of one or more task graphs to be swapped between two architectures. The probability of the task assignments of two task graphs remaining together during a crossover is proportional to the similarity between the data describing the task graphs, *e.g.*, periods and deadlines. A similar algorithm is used for both core allocation crossover and task assignment crossover.

3.5 Link prioritization

This subsection describes the algorithm used by MOCSYN to prioritize links. Communication priority is composed of two values: slack and communication volume. *Slack* is the difference between the earliest finish time and latest finish time of a task. Thus, it is the amount of time by which a task's execution can be delayed, from its earliest possible execution time, without causing any other tasks to miss their deadlines. Earliest finish times are computed by considering task execution times during a topological search of the task graph, starting from the node with no incoming edges. Latest finish times are computed by conducting a backward topological search of the task graph, starting from the nodes which have deadlines.

Task graph edges, which signify communication, have a slack equivalent to the average of the slacks of the tasks they connect. Link priority determination is conducted before block placement and bus topology generation. Therefore, it is not possible to take communication time into account during communication priority determination. Hence, at this stage, slack is only estimated. This problem is corrected later, during link re-prioritization. Communication volume is the quantity of communication which passes along a link. Link priority is a weighted sum of the reciprocals of the slacks of the task graph edges along it and its communication volume.

3.6 Floorplan block placement

This subsection describes the block placement algorithm used within MOCSYN's inner loop. This algorithm is based on previous work. Initially, a balanced binary tree of cores is formed, based on the priority of communication between core pairs. Accounting for the priority of communication between core pairs is an extension of the historical algorithm, which considered only the binary presence or absence of communication [28]. As a result, the time complexity of the partitioning algorithm is increased from $\mathcal{O}\left(n^2\right)$ to $\mathcal{O}\left(n^2 \cdot \log n\right)$ where n is the number of cores. Cores which are adjacent in the binary tree will be adjacent in the final block placement. After forming the binary tree, MOCSYN optimally determines the orientations of all of the cores such that the aspect ratio of the IC, *i.e.*, the ratio between width and height, does not exceed a value specified by the user. Under this condition, IC area is minimized. This algorithm is based on past work and takes $\mathcal{O}\left(n \cdot \log n\right)$ time [29].

3.7 Bus formation

This subsection describes the algorithm used by MOCSYN to produce an arbitrary bus topology.

MOCSYN recalculates link priorities using an algorithm similar to that described in Section 3.5. The global wiring delay information extracted from the block placement, however, is used to estimate communication time during this calculation.

In Fig. 4, the core graph represents four cores (**A**, **B**, **C**, and **D**). The edges connecting pairs of cores represent communication, *i.e.*, no edges exist for core pairs between which there is no communication. The numeric labels on the edges denote the priority of the communication occurring between the connected cores. Thus, communication with a priority of seven occurs between core **A**

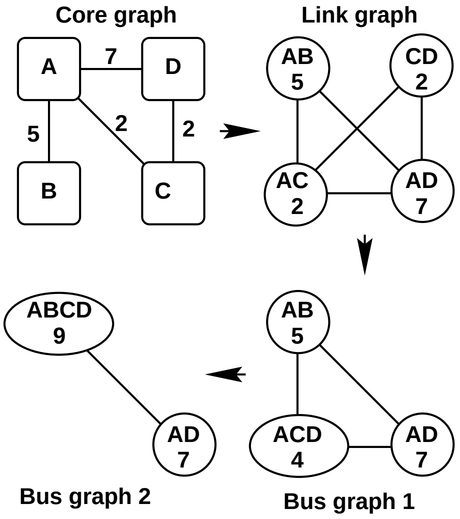

Figure 4: Bus formation

and core **D**. The first step of MOCSYN's bus formation algorithm converts this core graph into a link graph, as shown in Fig. 4. For every pair of cores between which communication occurs, a node with the priority equivalent to that pair's communication priority is added to the link graph. Link graph nodes which share at least one core are connected to each other with edges. Since core **A** communicates with core **B** with a priority of five, there is a node in the link graph, called **AB**, with a value of five. There is an edge between node **AB** and **AC** because they share core **A**. There is no edge between node **AB** and **CD** because they share no core.

The link graph is incrementally changed by merging the pair of nodes, between which there exists an edge and for which the sum of priorities is minimal, *i.e.*, less than the sum of the priorities of any other pair of nodes between which an edge exists. The new node's name is the set union of the merged nodes' names. The new node's priority is the sum of the priorities of the nodes merged to form it. Thus, when node **AC** is merged with node **CD** in bus graph 1, the resulting node has the name **ACD** and a priority of four (2 + 2). This algorithm is halted when the number of busses is less than or equal to a user-specified value.

After halting the bus formation algorithm, one is left with a bus graph in which each node represents a bus. Thus, in bus graph 2, there exists one global bus (**ABCD**) and one point-to-point link (**AD**). Note that the algorithm tends to form large common busses for multiple low-priority communications while producing smaller busses for high-priority communication. In this way, bus contention is reduced for high-priority communication while routing and multiplexing complexity is reduced for low-priority communication.

3.8 Scheduling

In this subsection, we describe the scheduling algorithm used in MOCSYN. Scheduling is NP-complete for distributed systems [8]. We, therefore, resort to a heuristic scheduling algorithm. MOCSYN uses a preemptive static critical path scheduling algorithm. The resulting schedule is static, *i.e.*, the time at which each event is carried out is computed by MOCSYN to determine whether or not hard deadlines are met by the schedule. Such guarantees are not possible, in general, when task priorities are allowed to vary during the operation of the synthesized architecture.

We assume that uniform buffers are distributed throughout the communication networks in order to minimize communication delay. The use of regularly distributed buffers reduces the dependency of delay on wire length from $\mathcal{O}\left(len^2\right)$ to $\mathcal{O}\left(len\right)$ where len is wire length. Given the process parameters, and V_{DD}, optimal buffer spacing is calculated. This value is used to determine the RC delay between a pair of cores [30]. This value is divided by the bus width and multiplied by the number of digital voltage transitions to determine the delay for a communication event. Core execution time is equal to the number of execution cycles divided by the core's frequency.

MOCSYN targets multi-rate embedded systems. It 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. When there are multiple copies of the same task graph, they are numbered in order of increasing start node earliest start time. Each copy's number is its *task graph copy number*. 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. MOCSYN handles this case correctly and is capable of interleaving tasks from different copies of the same task graph, as well as from different task graphs.

Before scheduling, MOCSYN assigns a priority to each task. This priority is the slack of the task. It differs from the slack computed in Section 3.5 because, at this point, a block placement has been generated. Thus, slack takes communication delay into account. Unfortunately, the effects of bus contention are unknown before scheduling is carried out.

When the scheduling algorithm begins, all 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 tasks which share the same slack by increasing task graph copy number. Tasks are iteratively removed from the end of the pending list and scheduled. After a task is scheduled, tasks which are data-dependent upon it are checked to determine whether all of their dependencies have been satisfied. Tasks which pass this test are entered into the pending list and the pending list is sorted again before scheduling the next task.

Before scheduling an individual task, t, MOCSYN schedules all of its incoming edges, *i.e.*, communication events. Each edge is scheduled on a bus connecting the core to which t is assigned and the core to which t's parent is assigned. When multiple busses are available, MOCSYN selects the bus upon which the communication event will complete at the earliest time. If either of the communicating cores does not have communication buffers, MOCSYN schedules the communication event to the unbuffered cores as well.

Every time a task is scheduled on a core, MOCSYN determines whether or not preemption is likely to result in an improved schedule. MOCSYN first tentatively schedules a task, t, to the earliest time slot on its core, which starts after its incoming edges have completed execution, and has a long enough duration to accommodate the task. MOCSYN 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 core processor before the next scheduled task, and preempting p does not change the times at which it communicates with tasks on other cores, then the preemption is carried out.

3.9 Cost

In this subsection, we describe the manner in which an architecture's costs are calculated. As mentioned before, MOCSYN optimizes architecture price, area, and power consumption under hard real-time constraints. An architecture is invalid if any task with a deadline violates that deadline. Power consumption is the sum of the energy consumptions of all of an IC's tasks executed on all its cores, throughout the hyperperiod, in addition to the energy consumed in the global clock distribution and communication networks, divided by the hyperperiod. As discussed in Section 3.8, we assume regularly spaced buffers in the global communication network. In addition, the clock network is assumed to be constructed with buffered segments. Leakage current is assumed to be negligible. This allows delay and energy consumption to be estimated as linear functions of wire length and transition count, with constant factors derived from the process parameters and V_{DD}. Ultimately, three such constant factors are computed: communication wire delay factor, communication wire energy factor, and clock energy factor. The energy consumed by the global clock network is determined by estimating the total wire length of this network, multiplying this value by the number of transitions occurring during a hyperperiod, and also multiplying by the clock energy factor. The wire length estimate is derived from a minimal spanning tree of the core positions in the block placement. This provides a conservative estimate on wire length. A Steiner tree may be used in the final post-optimization routing operation. However, computation of minimal Steiner trees is time-consuming (NP-complete). Hence, it is not used in inner-loop routing estimates. Energy consumption in the global communication networks is similarly computed. A separate minimal spanning tree is computed for each bus. The transitions required for the communication events occurring on each bus are used to compute the bus energy consumptions.

An architecture's price is the sum of the prices of all the cores on the IC plus the area-dependent price of the IC. The area of the IC is equivalent to the total rectangular area required for its block placement.

4 Experimental results

In this section, we present experimental results demonstrating the effects of a number of algorithms employed within MOCSYN. Section 4.1 shows the results produced by the clock selection algorithm when run on an example. In Section 4.2, we empirically determine the influence of a number of MOCSYN's specialized algorithms. Section 4.3 shows the result of running MOCSYN on a number of examples in multiobjective optimization mode.

Previous co-synthesis systems do not target the single-chip synthesis problem. As a result, there is not a body of examples with which MOCSYN's performance can be compared. It is, however, possible to experimentally determine the effects of the algorithms comprising MOCSYN. The examples discussed below attempt to determine how clock selection, block placement, and bus topology generation affect the solution of the single-chip synthesis problem. The data used in these examples are available via anonymous FTP at ftp://ftp.ee.princeton.edu/pub/dickrp/MOCSYN.

4.1 Clock selection

MOCSYN automatically selects clock frequencies for each core using the algorithm described in Section 3.2. In this subsection, we examine the results produced by this algorithm when run on an example problem.

In the interest of decreasing the power consumed in the global clock distribution network, one may reduce the frequency of the base clock. There is a trade-off between power consumption and execution time. However, this relationship is not linear. Fig. 5 shows the relationship between maximum reference clock frequency and the average proportion of maximum internal clock rates at which the cores are clocked for a set of eight cores, each of which has a random maximum internal frequency ranging from 2 to 100 MHz. Each sample point lies at the optimal reference clock frequency for a set of core multiplier values. The top solid line shows the average ratio of actual core frequencies to maximum core frequencies for linear interpolating clock synthesizers with a maximum numerator of eight. The bottom solid line corresponds to a cyclic counter clock divider. The dotted lines indicate the maximum ratio encountered before or at each frequency. The increase in power consumed by the clock reference frequency distribution network is approximately a linear function of frequency.

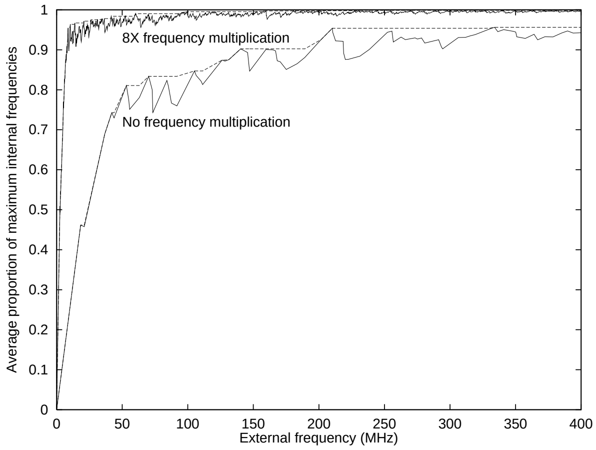

Figure 5: Clock selection quality as a function of external frequency

As shown in Fig. 5, the quality of the internal clock frequencies is a sub-linear function of reference clock frequency. If one were using an interpolating synthesizer with maximum numerator of eight for the cores in this example, choosing a maximum reference frequency greater than 100MHz would not result in a significant increase in execution speed but may have a negative impact on system power consumption.

4.2 Feature comparisons

This subsection empirically shows the influence of a number of the core-based synthesis algorithms used in MOCSYN.

Table 1 shows the results of synthesizing a number of ICs using MOCSYN with various sets of features enabled. For these examples, price was optimized under hard real-time constraints. If multiobjective optimization were used, it would be difficult to compare the solutions produced by different versions of MOCSYN because each problem might have more than one solution (see Section 4.3). Each example in Table 1 is produced with the aid of TGFF [31], a randomized task graph and core generator which allows correlation between different attributes. The examples are multi-rate. Each contains six task graphs with an average of eight tasks each and a variability of seven tasks. For each task with a deadline, the deadline is equal to $(depth + 1) \cdot 7,800\mu$s where $depth$ is the distance of a task, in nodes, from the start node of a task graph. Each communication event requires an average of 256 kilobytes, with a variability of 200 kilobytes, of data to be transferred. There are eight core types, each of which has an average price of 100 with a variability of 80. Each core has an average width and height of 6 mm and a width and height variability of 3 mm. The core maximum frequency average is 50 MHz with a variability of 25 MHz. Cores have buffered communication 92% of the time. Communication on cores requires an average of 10 nJ per cycle with a variability of 5 nJ per cycle. Tasks require an average of 16,000 cycles to execute with a variability of 15,000 cycles. Task preemption takes an average of 1,600 cycles with a variability of 1,500 cycles. Tasks dissipate an average of 20 nJ per cycle with a variability of 16 nJ per cycle. 57% of the core types are capable of executing any given task type. Communication wire delay factor, communication wire energy factor, and clock energy factor are calculated based on the 0.25 μm process parameters given in the literature [32], with a V_{DD} of 2.0 V. Communication networks are assumed to be 32 bits wide. Wire delay and power consumption per μm per transition are calculated based on the use of a buffer separation distance which optimizes delay per μm. The maximum clock reference frequency is 200 MHz and the maximum interpo-

Table 1: Feature comparisons

Example	MOCSYN price	Worst-case commun. price	Best-case commun. price	Single bus price
2	181	181	181	181
3	255	255		255
4	211	211	211	211
5	154	154	154	154
6	230			
7	174	174	174	174
8	219		219	
9	182	182		182
10	901			
11	166	166	166	166
12	405	405		
13	636			
14	166	368		166
15	151	151	151	151
16	389			
17	302			
18	315			
19	320	365	320	
20	176	176	176	176
21	212			
22	170	257	257	257
23	322	323	322	
24	421	421		
25	113	113	113	113
26	440			
27	173	173	173	173
28	858			
29	169	169	186	169
30	203	203		203
32	396	406		
33	317			
34	368			
35	694			
36	269			
37				132
38	460		460	
39	179	270	179	270
40	327	327		
41	273	273		273
42	426			
43	72			72
44	182			182
45	148	148	148	148
46	344	344		344
47	404	404		285
48	135			135
49	60	60	60	60
50	291	357		211
Better	0	0	0	3
Worse	0	26	31	24

lating clock synthesizer numerator is eight. For each example, the same parameters are given to TGFF and MOCSYN. Only the random seed given to TGFF is varied, to produce different examples based on the same parameters.

The bottom two rows in Table 1 display the number of solutions which are superior and the number of solutions which are inferior to those produced by MOCSYN with all its features turned on. The first column in Table 1 shows the random seed given to TGFF to generate the corresponding example. The second column shows the price of solutions produced by MOCSYN when carrying out block placement-based wire delay estimations and using up to eight busses organized to reduce bus contention. Empty price entries indicate examples for which no solution was found. Example 11 is omitted from Table 1 because no solution was ever found for them. Note that there is no guarantee that solutions exist for all of the problems produced by TGFF. Each of the examples in this section took less than two minutes to complete on a 200 MHz Pentium Pro running Linux.

The third column shows the price of solutions under the assumption that the distance in the block placement between each pair of cores is equal to the maximum distance between any pair of cores. Although this estimate may appear conservative, it is not possible to derive a tight bound on the maximum separation between any pair of cores without carrying out block placement in the inner loop. Thus, in practice, this estimate would probably be even more conservative if an inner-loop block placer were not available. Using worst-case communication delay assumptions never resulted in superior results to block placement-based communication delay assumptions. However, MOCSYN's performance with block placement based communication delay estimates was superior to its performance with worst-case estimates for 26 examples, many of which were unsolvable when the worst-case communication delay assumption was used.

The fourth column shows the price of solutions which result from carrying out optimization under the assumption that communication events take almost no time. After the optimization run is complete, solutions which are invalid due to unschedulability are eliminated. Best-case communication delay estimates never resulted in an improvement in solution quality over block placement-based communication delay estimates. However, the use of block placement-based communication delay estimates resulted in superior solutions for 31 of the examples.

The fifth column shows the price of solutions which result from allowing MOCSYN to carry out block placement in the inner loop to accurately estimate communication delay but allowing only a global bus to be used, instead of an arbitrary priority-based topology of up to eight busses. For three examples, this approach actually resulted in an improvement over the solutions produced when allowing up to eight busses. For these examples, there was a valid solution which contained only two or three cores, thereby reducing the amount of off-core communication. The availability of a large number of busses reduced bus contention, making it practical to use a larger number of cores. However, when only one bus was available, it became essential to minimize communication in order to minimize bus contention. Thus, with only one bus, MOCSYN was forced to concentrate its optimization efforts on solutions with allocations containing only a few cores. For a couple of examples, this focussed exploration of the solution space paid off. However, there is no guarantee that a solution with a small number of cores exists. Thus, for 24 examples, being forced to concentrate on solutions with a small number of cores resulted in inferior solutions to those produced when eight busses were available, or no solution at all. In general, the features employed by MOCSYN to synthesize core-based systems result in superior performance to simpler approaches.

4.3 Multiobjective optimization

In this subsection, the results of using MOCSYN to conduct multiobjective optimization on a number of examples are presented. When MOCSYN is run in multiobjective optimization mode, it produces a set of solutions, each of which is superior, in some way, to at least one other solution. Table 2 shows the sets of solutions produced for ten examples. These are produced with the aid of TGFF. They use the same parameters as the examples in Section 4.2 with one exception: the average number of tasks in each task graph is related to the example number (ex) in the following manner: $1 + ex \cdot 2$. Thus, the six task graphs in Example 10 each has an average of 21 tasks. The variability in the number of tasks in each task graph is always one less than the average number of tasks in each task graph. MOCSYN took less than seven minutes when run on each of these examples. For some of the examples, MOCSYN found numerous solutions which trade off price, area,

Table 2: Multiobjective Optimization

Example	Price	Area (mm^2)	Average Power (mW)
1	318	90	479.7
	358	96	443.3
	543	196	424.1
	554	182	420.3
	612	216	384.6
2	181	50	199.7
	186	65	151.1
	250	91	128.1
3	166	72	47.9
	170	78	44.1
4	181	78	260.7
	211	66	363.3
5	154	56	183.6
	276	120	170.4
	483	232	164.4
6	405	176	172.7
	462	224	127.0
7	126	36	110.1
8	219	90	41.3
9	182	60	72.7
10	781	352	114.3

and average power consumption.

5 Conclusions and future work

In this paper, we presented a method for the synthesis of core-based, single-chip, low-price, low-power, real-time, multi-rate, heterogeneous embedded systems. A multiobjective genetic algorithm, which allows exploration of the Pareto-optimal set of architectures instead of providing a designer with a single solution, was applied to a number of examples. MOCSYN's use of automatic clock selection, block placement-based communication delay estimation, and arbitrary bus topology generation allows it to solve the core-based synthesis problem.

6 Acknowledgments

We would like to thank I-Jong Lin for his suggestions regarding an elegant clock frequency selection algorithm.

References

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

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

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

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

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

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

[7] R. Weiss, "32-bit cores drive systems-on-a-chip," *Computer Design*, pp. 82–89, Sept. 1996.

[8] 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.

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

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

[11] 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.

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

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

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

[15] 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.

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

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

[18] 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.

[19] W. Wolf, "Floorplanning: The art of chip-level design," *Electronics Journal*, pp. 8–13, Oct. 1998.

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

[21] 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.

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

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

[24] J. Smith and G. De Micheli, "Automated composition of hardware components," in *Proc. Design Automation Conf.*, pp. 14–19, June 1998.

[25] M. Kishinevsky, J. Cortadella, and A. Kondratyev, "Asynchronous interface specification, analysis and synthesis," in *Proc. Design Automation Conf.*, pp. 2–7, June 1998.

[26] M. Bazes, R. Ashuri, and E. Knoll, "An interpolating clock synthesizer," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 1295–1300, Sept. 1996.

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

[28] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. Design Automation Conf.*, pp. 173–181, June 1982.

[29] L. Stockmeyer, "Optimal orientations of cells in slicing floorplan designs," *Information and Control*, vol. 57, pp. 91–101, May/June 1983.

[30] N. H. E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design, A Systems Perspective*, Addison-Wesley, Reading, MA, 2 ed., 1993.

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

[32] J. Cong *et al.*, "Interconnect design for deep submicron ICs," in *Proc. Int. Conf. Computer-Aided Design*, pp. 478–485, Nov. 1997.