Functional Partitioning for Low Power Distributed Systems of Systems-on-a-chip

Yunsi Fei and Niraj K. Jha
Department of Electrical Engineering, Princeton University, NJ 08544
{yfei, jha}@ee.princeton.edu

Abstract

In this paper, we present a functional partitioning method for low power real-time distributed embedded systems whose constituent nodes are systems-on-a-chip (SOCs). The system-level specification is assumed to be given as a set of task graphs. The goal is to partition the task graphs so that each partitioned segment is implemented as an SOC and the embedded system is realized as a distributed system of SOCs. Unlike most previous synthesis and partitioning tools, this technique merges partitioning and system synthesis (allocation, assignment, and scheduling) into one integrated process; both are implemented within a genetic algorithm. Genetic algorithms can escape local minima and explore the partitioning and synthesis design space efficiently. Through integration with an existing SOC synthesis tool, the proposed partitioning technique satisfies both the hard real-time constraints and the SOC area constraint of each partitioned segment. Under these constraints, our tool performs multi-objective optimization. Thus, with a single run of the tool, it produces multiple distributed SOC-based embedded system architectures that trade off the overall distributed system price and power consumption. Experimental results show the efficacy of our technique.

1 Introduction

The ever-shrinking geometries of integrated circuits have made it possible to implement an entire embedded system on an SOC. However, the functionality being implemented by embedded systems is also rapidly increasing. This frequently necessitates realizing the functionality with a number of SOCs, which involves functional partitioning of a system specification and synthesis of each partition segment on a separate chip. In single-chip synthesis, the tool may possibly optimize price, area, and power consumption under real-time constraints. However, for the synthesis of a distributed system of SOCs, area is also a constraint in order to obtain satisfactory yields. Thus, after partitioning, each segment needs to be synthesized with an SOC that satisfies the area constraint. In addition, the real-time constraints are not local to a single SOC. They apply to the whole distributed system of SOCs. Thus, synthesizing and scheduling the communication architecture among SOCs becomes necessary to make sure that the real-time constraints are met. This makes the problem much more complex.

Acknowledgments: This work was supported by DARPA under contract no. DAAB07-00-C-L516.

Design automation involving synthesis and partitioning can be broadly classified into vertical and integrated design flows [1]. In the vertical design flow, as shown in Figure 1(a), functional partitioning and structural partitioning are two possible approaches. In functional partitioning, partitioning is done first, followed by synthesis of each partition segment. In structural partitioning, synthesis is done first, followed by the partitioning of the resultant structure. A detailed comparison of these approaches, showing advantages of functional partitioning, is given in [2]. Our approach uses functional partitioning.

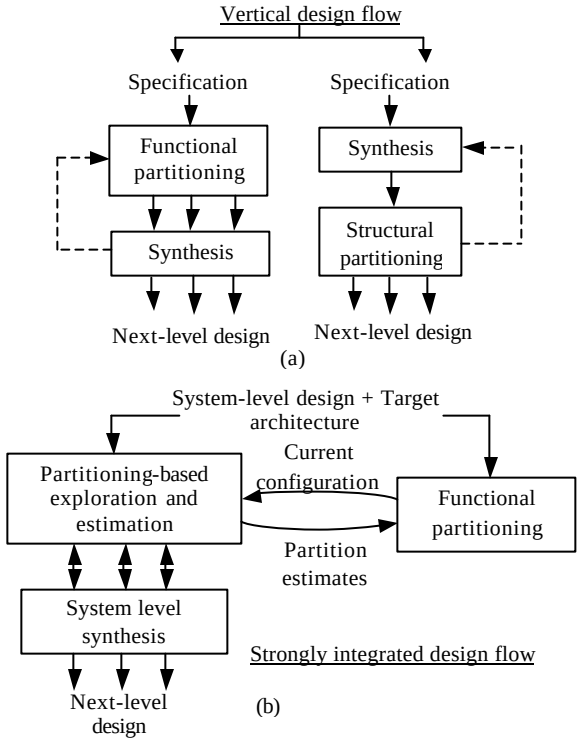

Figure 1. Synthesis and partitioning design flows

The integrated design flow can be classified into three approaches [1]: traditional heterogeneous model, strongly integrated heterogeneous model, and the homogeneous model. For the traditional heterogeneous model, since the partitioner has no *a priori* knowledge about design parameters such as the actual area and schedule, the focus of research is to provide

good design estimates while not performing complete synthesis. Previous work has employed various methods to provide design estimates at different levels of the integrated circuit design hierarchy [4]-[8].

The strongly integrated heterogeneous model has been presented in [1], [9]. As shown in Figure 1(b), in this model, both the partitioning and synthesis exploration engines maintain an identical view of the partitioned behavior. During partitioning, the design space exploration technique performs a global search in a four-dimensional design space, and the partitioner always communicates any changes in the partitioned configuration to the synthesis engines [1]. There is an exploration control interface between the partitioner and synthesizer.

In a homogeneous model, partitioning and synthesis (allocation, assignment, and scheduling) are simultaneously explored. MULTIPAR uses an integer linear programming formulation to solve the partitioning and scheduling problems simultaneously [10]. COBRA-ABS uses a heuristic simulated annealing algorithm [11]. However, unification of partitioning and synthesis into a homogeneous model leads to a large multi-dimensional design space. The cost is generally high, either in high run-time (COBRA-ABS) or inability to handle large problems (MULTIPAR).

The limitation of the above homogeneous models stems from the way partitioning and synthesis are simultaneously tackled. In this paper, we propose a homogeneous model, which uses an effective search algorithm, a genetic algorithm, at the system level. In this model, partitioning is performed simultaneously with allocation and assignment. This step yields a distributed system of SOCs. A global schedule is then obtained for the distributed system. Our algorithm optimizes overall system price and power consumption under real-time and SOC area constraints. We do not employ an exploration control interface between partitioning and synthesis as in [1]. This leads to an efficient and scalable algorithm which produces high-quality distributed systems of SOCs in relatively small run-times.

The rest of the paper is organized as follows. Section 2 provides background material and describes the system synthesis framework on top of which the partitioner is built. Section 3 presents the proposed framework for integrated functional partitioning and system synthesis. Section 4 presents a genetic algorithm for partitioning, as well as synthesis, and their integration. Section 5 presents experimental results, and Section 6 the conclusions.

2 Problem Definition and System Synthesis Framework

In this section, we define the problem of functional partitioning and some terms used in system synthesis, and also describe the system synthesis framework.

2.1 The functional partitioning problem

At the system level, functional partitioning is the process of dividing the system-level functional specification into several partition segments such that each segment can be synthesized on a separate chip.

Several basic issues need to be addressed in a functional partitioning system [3], as shown in Figure 2. The first one deals with the specification abstraction level, which can be system level, behavior level, register-transfer level, or logic level. Our work is at the system level, and the specification is in the form of a set of task graphs, which will be described in the next subsection. The second issue is the granularity of the functional objects into which the input specification is decomposed. For certain input abstraction levels, there is only one reasonable granularity of decomposition. For example, in our partitioning scenario, decomposition is limited to the level of a task, since a task is the basic atomic functional block. After decomposition, the partitioning algorithm maps the functional objects to system components from a resource library. The partitioning result is evaluated based on the given constraints and estimation of the optimization objectives. Finally, the output of functional partitioning can be used as an input to a synthesis tool for implementing each of the partitioned segments. There should also be a flow of control to specify the sequence of decisions made within partitioning and designer interactions, which will be discussed in Section 3.

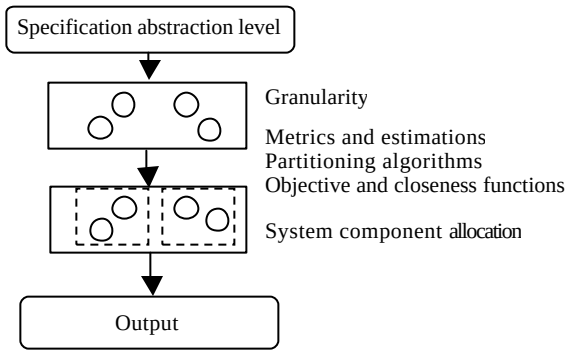

Figure 2. Basic functional partitioning system

Formally, the system-level constraint-driven functional partitioning problem can be stated as follows:

Given a set of task graphs $G = <V, E>$, in which V (vertex) represents tasks, E (edge) represents the data dependence relationship and data flow between tasks, a functional partitioner partitions G into a number of non-overlapping sub-graphs P_1, P_2,…, P_l ($l \le m$, the maximum number of chips allowed) such that $A_i \le AC$, $1 \le i \le l$, $T_j \le Dl_j$, $1 \le j \le k$, where k is the number of tasks with a deadline, A_i is the area of each chip, AC is the chip area constraint, T_j is the finish time of task j with a specified deadline Dl_j; while multiple objectives, such as system price and power, are optimized.

2.2 The system synthesis tool

Hardware-software co-synthesis is a sub-problem of SOC synthesis. Given a system specification and resource library (the available types of processing elements and communication links) as inputs, a co-synthesis algorithm needs to solve the following problems: allocation, assignment, scheduling, and performance/power evaluation. First, the co-synthesis algorithm selects the number and type of hardware and software processing elements upon which the tasks will execute (allocation). Then the algorithm assigns each task to a processing element and each communication edge to a communication link (assignment). Finally, schedules are produced for both processing elements and communication links, such that the real-time constraints for all task graphs are met.

Optimal co-synthesis is a difficult problem, since

allocation/assignment, and scheduling are each NP-complete for distributed systems [12]. Hence, optimal co-synthesis based on mixed integer linear programming or exhaustive exploration can only be applied to problems of small size [13]. There are three popular classes of co-synthesis heuristics: constructive, iterative, and genetic. A constructive approach builds a system from bottom-up, adding components to the system incrementally [14]. An iterative approach starts with a feasible solution, either randomly produced, or obtained in a deterministic pre-processing step [15]. It then perturbs the solution and attempts to improve its quality iteratively. A genetic algorithm improves a set of feasible solutions through reproduction, crossover, and mutation [16], [22].

We next define various terms that are used in co-synthesis.

Task graph: A task graph is a system-level description model. It is a directed acyclic data-flow graph as shown in Figure 3. A node in a task graph represents a coarse-grained task, e.g., discrete cosine transform or FFT. An edge in a task graph is associated with communication between two tasks, and its label denotes the amount of data that needs to be transmitted along the edge. A task with incoming edges can only be executed after it receives all the data from its parent tasks. For each sink task, there is an associated deadline. Deadlines may also be specified for some other tasks. The period associated with a task graph denotes the interval between successive executions of the task graph. An embedded system may be specified in terms of a set of task graphs, each with a different period. Such a system is called multi-rate. Deadlines may be smaller than, equal to or greater than the periods.

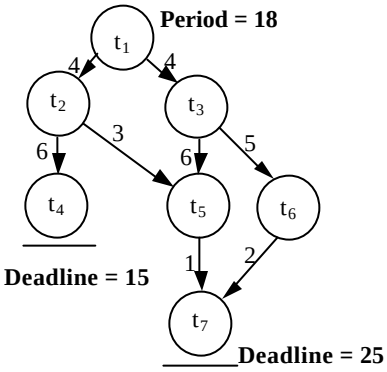

Figure 3. Task graph

Hyperperiod: In a multi-rate system, the least common multiple (LCM) of all periods is called the hyperperiod. It is well known that a valid system schedule can be obtained by repeatedly executing the task graphs in the hyperperiod [17].

Resource library: There are many system components available to execute the tasks and communication events between tasks. A core can execute one or more tasks, and a link (point-to-point contact or bus) allows two or more cores or SOCs to communicate with each other. A resource library includes cores, links and memories that will be used for SOC synthesis. Various characteristics are specified for each core, such as price, width, height, maximum clock frequency, a variable indicating whether or not its communication is buffered, and an energy consumption per cycle dedicated to communication. For each task run on each valid core in the resource library, the worst-case execution time and average power consumption are specified. This information can be obtained through the techniques such as those provided in [18]-[21].

Our system synthesis framework: We have built our functional partitioner on top of the system synthesis framework shown in Figure 4 [22]. This framework employs a hierarchical genetic algorithm. It has two loops: an outer cluster loop for allocation and an inner architecture loop for assignment and scheduling. A cluster of architectures contains the same allocation, but different task assignments. The clock selection step determines the clock frequency for each core, assuming asynchronous communication among cores. In the initialization step, the algorithm initializes basic data structures. In the outer loop, it uses a genetic algorithm to change core allocations. In the inner loop, it changes task assignments also using a genetic algorithm. A block placement floorplanner ensures that cores which have high communication priority are located next to each other. A bus structure that trades off potential bus contention for ease of routing is produced. Scheduling of tasks and edges follows. At this point, it can be verified if the real-time constraints are met. Finally, inferior solutions are eliminated before evolving the set of solutions from one generation to another. If there is no improvement for a given number of generations, synthesis halts.

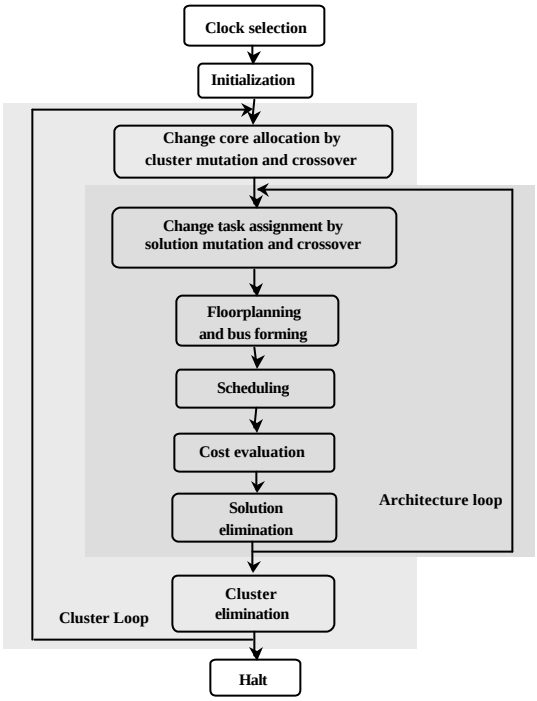

Figure 4. System synthesis overview

3 The Partitioning and System Synthesis Framework

In our integrated framework, it is hard to separate the partitioner from the system synthesizer. An overview of this framework is given in Figure 5. We use a homogeneous partitioning model. After partitioning, allocation, and assignment, we can obtain a global system schedule, and then evaluate an array of costs, such as system price, power consumption, area constraint violation and real-time constraint violation, *etc.* Hence, we do not need to build a special estimator or exploration control interface [23] between the partitioner and

synthesizer. We use a genetic algorithm to optimize system price and power under SOC area, and real-time constraints. The output of our integrated partitioning and synthesis framework is distributed systems of SOCs.

The process of partitioning and synthesis is also a process of design space exploration. If we define an SOC implementation as a design point, then system synthesis has a three-dimensional design space with tasks (edges), cores (buses) and time as the three dimensions. Assignment and scheduling correspond to projecting this space on to different two-dimensional surfaces. In the integrated partitioning and synthesis process, the design point is four-dimensional, with another dimension being the partitioned segments implemented on different SOCs in the distributed system. Although this leads to a more complex search problem, by integrating partitioning and synthesis using a genetic algorithm as the search and optimization engine, our proposed technique can escape local minima and simultaneously optimize system power and price efficiently. We discuss this algorithm in greater detail in Section 4.2.

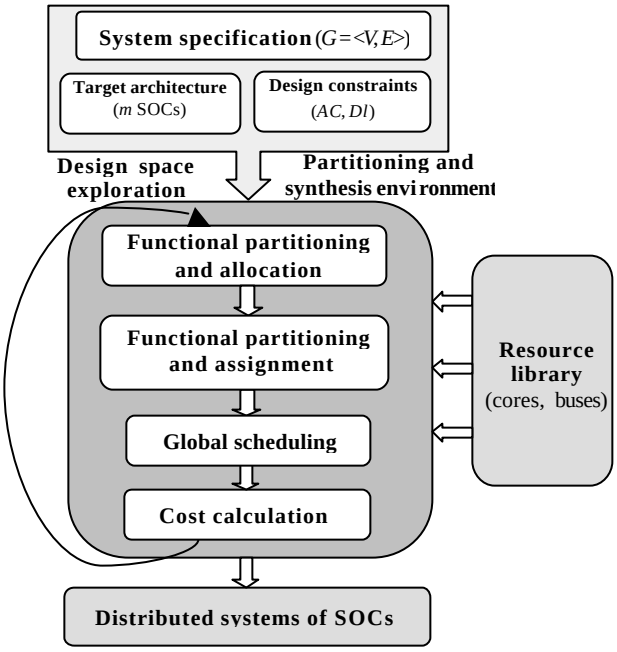

Figure 5. Partitioning and synthesis framework

4 A Genetic Algorithm for Partitioning and System Synthesis

In this section, we first describe a basic genetic algorithm, and then show how it can be used in our context.

4.1 The genetic algorithm

There have been some studies using genetic algorithms for the graph-partitioning problem [24]. A genetic algorithm starts with a set of initial solutions (chromosomes), called the population. The population evolves iteratively from generation to generation using the following operators: mutation, crossover, and selection. At the end, when some stopping criteria are met, the algorithm returns the best solutions encountered till then as the output.

Crossover and mutation enable efficient design space exploration. In crossover, two members are selected from a population and features from them swapped to produce two offspring. In mutation, a feature of a member is mutated to some other feature with some fixed probability. This allows newer parts of the design space to be explored. When the offspring are included in the population, the population size increases. The selection operator ranks the members and selects the best among them to reduce the population size back to the original size.

Genetic algorithms excel at multi-objective optimization, i.e., the simultaneous optimization of multiple costs, such as price and power. Our genetic algorithm stops when the number of generations without any improvement in solution quality exceeds a given threshold. This is a commonly used criterion for genetic algorithms.

4.2 A genetic algorithm for joint partitioning and synthesis

In this subsection, we explain the manner in which solutions are represented and optimized by a genetic algorithm for the integrated partitioning and synthesis framework.

Our algorithm is an improved algorithm based on simulated annealing and genetic algorithm. Hence, it is called an evolutionary algorithm. Similar to simulated annealing, a geometrically decreasing global temperature is maintained. The three genetic algorithm operators are used both for partitioning and system synthesis. The overview of the algorithm was shown in Figure 5. There are two hierarchical loops embedded in this framework (not shown for simplicity). Initially, we generate a pool of solutions randomly. The solutions are encoded in strings and classified into several clusters, each cluster having the same allocation and partitioning, i.e., within a cluster, the instances of each core type on each SOC is fixed. However, within a cluster, different solutions have different assignments. Partitioning is actually executed in two steps. In the first step, it is combined with allocation, and in the second step with assignment. Similar to the system synthesis framework shown in Figure 4, in the outer cluster loop, we use the genetic operators on the allocation and partitioning strings, and in the inner loop, we use them on the assignment and partitioning strings. Some new solutions are generated after these operations. Then after floorplanning, bus formation, and global scheduling for each solution, system costs are evaluated and the solutions ranked according to their costs. Some inferior solutions are pruned from the pool and a new generation of solutions emerges. In each pass of the outer loop, the inner solution loop executes multiple times. The outer loop continues to execute until a number of iterations has been carried out without an improvement, at which point, the global temperature is lowered, resulting in greedier optimization. When a large number of iterations have passed without an improvement in the solution quality, the algorithm is halted and all the best solutions reported.

In the following subsections, we present the details of the algorithm and framework.

4.2.1 Functional partitioning and allocation

We use strings to encode the mapping between cores and SOCs, and the mapping between tasks and cores. For allocation and the first step of partitioning, we use several integer arrays to represent each cluster. The number of arrays is equal to the maximum number of SOCs allowed, and the length of each array is equal to the number of core types. An entry in the array denotes the number of instances of the corresponding core type.

For example, suppose the maximum number of SOCs allowed is three and the number of core types in the resource library is nine. Figure 6 shows three arrays, one for each SOC. On SOC_1, represented by array S_1, there is one instance of core type 2, three instances of core type 4, and so on. The mapping between cores and SOCs represents the first step of partitioning and allocation.

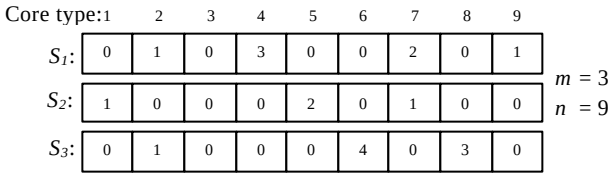

Figure 6. Cluster allocation encoding

Crossover at the cluster level operates on two randomly selected arrays representing two clusters. An associated array of length n is generated to indicate which entries of the two arrays should be swapped. For example, in Figure 7, the entries for core types 1, 2, 3 and 4 are swapped between arrays S_1 and S_2.

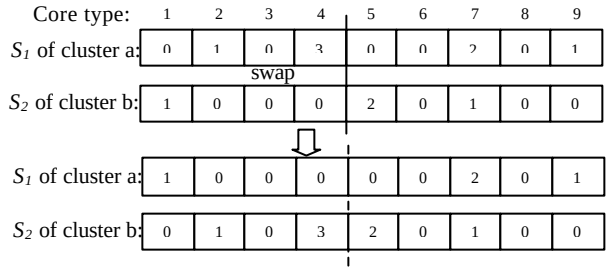

Figure 7. Cluster crossover

The mutation operator randomly selects an array and an entry in this array, and changes its value, either by adding 1 or subtracting 1. This corresponds to the addition or deletion of an instance of that core type in the SOC. Another mutation operator moves some cores from one SOC to another SOC. This operation guides the allocator to place as many cores onto one SOC as possible.

4.2.2 Functional partitioning and assignment

After allocation and the first step of partitioning, the number and types of available cores on each SOC are known. The next step is to assign the tasks from the task graphs to the cores. Thus, partitioning, the mapping between tasks and SOCs, is fulfilled by the first two steps, as shown in Figure 8.

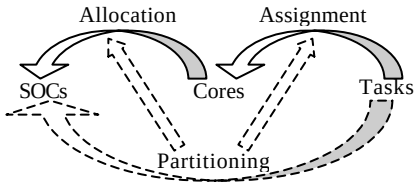

Figure 8. The mapping relationship between allocation, assignment, and partitioning

The algorithm selects a valid core for each task (i.e., a core which can execute the task). The assignment is encoded in a two-dimensional vector *task_pe*. Assume that the system specification has p task graphs, each with q_i tasks, $1 \le i \le p$. Then the entry in *task_pe*[i][j], $1 \le i \le p$, $1 \le j \le q_i$, is a three-tuple index, indicating the core type, core instance and the SOC to which the task is assigned.

The mutation and crossover operators for task assignment then operate within one cluster, on solutions containing different assignments. For crossover, two solutions and several task graphs are selected randomly. For each task in these graphs, the *task_pe* entry is swapped between the two solutions. This operation changes not only the assignment of task to cores, but also the partitioning of tasks to SOCs. Mutation acts on a randomly selected solution, by selecting a task graph and changing the assignment of a number of its tasks. Since the inter-SOC communication delay is much bigger and the power consumption is greater than on-chip communication, task assignment mutation is guided by a heuristic designed to minimize inter-SOC communication link bandwidth requirement. It uses a metric called distance for this purpose. When a task is to be reassigned, all its neighbors (other tasks with which it communicates) are checked. The distance metric indicates the amount of communication data transmitted on the inter-SOC link if the task were to be assigned to a core on a particular SOC. The distance values are determined for different SOCs on which the task can run. This array of distance values is used to determine the attractiveness of an SOC to implement the task in question, in order to reduce the inter-SOC communication.

The algorithm next runs the floorplanner to place the cores on each SOC. This gives an estimate of each SOC area. A bus-forming algorithm is then used to form the necessary busses on each SOC. This balances ease of routing and bus contention [22].

4.2.3 Global scheduling

Global scheduling refers to scheduling of the distributed system of SOCs. In the single-chip synthesis system, there is a local scheduler which schedules the task and events on all the cores and links (busses) in the chip. In a multi-chip system, communications may take place between tasks assigned to cores on different SOCs. Thus, a global schedule needs to be obtained, based on local SOC scheduling and inter-SOC link scheduling.

Since the number of SOCs is likely to be small, we assume that all the SOCs are connected by a global bus. Since scheduling is NP-complete for distributed systems [12], we resort to a heuristic list-based scheduling algorithm, which schedules the tasks in the hyperperiod based on task priorities, which depend on the execution time, communication time, and deadlines [22].

4.2.4 Cost calculation

After partitioning and synthesis, we obtain a pool of distributed systems of SOCs. The pool contains several clusters, each of which contains several solutions. The genetic algorithm calculates the cost of each solution, and prunes those with large costs, thus evolving from generation to generation and exploring the multiobjective solution space. In this subsection, we describe the manner in which the costs are calculated.

In the single-SOC synthesis system, the total area of a single chip is taken as one objective, and is optimized in the cost function. In our multi-SOC partitioning and synthesis system, area is a hard constraint. A distributed architecture is declared invalid if any partition segment requires an SOC that does not

meet the area constraint. The area of an SOC is given by the total rectangular area required for its block placement, which is obtained in the inner-loop floorplanner. Compared to other functional partitioning algorithms, which put much effort into estimating the area of each partitioned segment from a separate area estimator, or some incomplete lower-level synthesis, our homogeneous partitioning/synthesis approach directly provides an area estimation from the synthesizer to the partitioner.

The real-time constraint is also a hard constraint, which means that the distributed system of SOCs is invalid if there is any deadline violation. The global scheduler helps determine if any violation occurs. However, we allow solutions with minor area and real-time constraint violations to continue to exist in the pool. We have found that such solutions frequently give rise to high-quality valid solutions in later generations.

The energy consumption of the distributed architecture is the sum of the energy consumed by all the tasks executed on all the cores in the different SOCs in the hyperperiod, the energy consumed in the global clock distribution and communication networks (either on an individual SOC or the inter-SOC bus), and the idle energy. The power consumption of the distributed architecture is calculated by dividing the above energy consumption by the hyperperiod as follows:

$$Power = \frac{\sum_{i,j} Energy_{T_i,C_j} + Energy_{clock} + Energy_{offchip\_comm} + Energy_{onchip\_comm} + Energy_{idle}}{hyperperiod}$$

where T_i and C_j refer to tasks and cores, respectively. Details of clock network and wire delay/energy estimation are given in [22]. For the inter-SOC global bus, the transmission time is the product of the number of packets and the average packet transmission time. The energy consumed by it is the product of its specified average power consumption and the transmission time. The idle energy consumption of the bus is also taken into account. The overall system price of a solution is the sum of the prices of all the cores on all the SOCs, the inter-SOC bus price, and the area-dependent price of the SOCs.

4.2.5 Pareto-ranking of solutions and Boltzmann trials

All the solutions in the pool are ranked based on the above cost evaluation. Since the co-synthesis problem is that of multi-objective optimization, there are more than one cost involved. Improving one cost often results in the degradation of another. Most past co-synthesis systems collapse all costs into one variable by using an array of linear weights. However, it is difficult to select the weight array appropriately without exploring the Pareto-optimal set of solutions, which are solutions that can only be improved in one area by being degraded in another. Instead, we resort to Pareto-ranking to rank the solutions, as exp lained next.

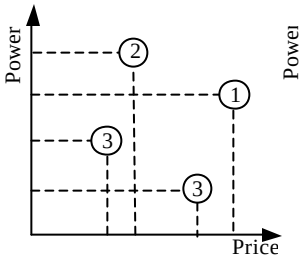

Figure 9. Pareto ranking

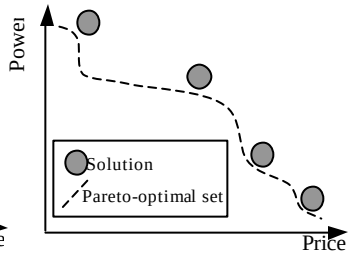

Figure 10. Multiobjective optimization

A solution dominates another if all of its features are better. A solution's Pareto-rank is the number of other solutions, in the solution pool, which do not dominate it. In Figure 9, each circle represents a solution with its power consumption and price projected onto two axes. The number inside the circle denotes its Pareto-rank. At the end of the multiobjective genetic algorithm's run, a number of non-dominated solutions are presented, as shown by the shaded circles in Figure 10. Although these non-dominated solutions are not guaranteed to be Pareto-optimal, they provide an upper bound on the Pareto-optimal set. Solutions are selected for reproduction by conducting Boltzmann trials between randomly selected pairs. Given a global temperature T, a solution with rank J beats solution with rank K with probability $1/(1+e^{(K-J)/T})$. Lower the temperature, greedier the algorithm.

5 Experimental Results

In this section, we present experimental results demonstrating the effectiveness of the integrated functional partitioning and system synthesis framework. We have implemented the framework in C++ and obtained the results on a 550 MHz Pentium Pro III with 256 MB of memory running under Linux. Since previous approaches do not solve the problem solved in this paper, there is not a body of examples that our framework's performance can be compared with. However, we have exercised the functionality of the framework with both a real-life example and several sets of large randomized task graphs to show that it can produce good-quality solutions quickly.

5.1 Signal processing case study

To investigate the practicality of our functional partitioning scheme, we ran our algorithm on a set of large task graphs drawn from a real digital signal processing system (DSP) [25]. The task graph structure is shown in Figure 11. It processes sonar data with five independent threads, each driven by its own sensors. There are 120 tasks in five task graphs. The first three task graphs are identical, with 25 tasks each; the fourth consists of tasks 76-111, and 120; and the fifth, of tasks 112-119. The tasks are classified into 20 distinct types according to their

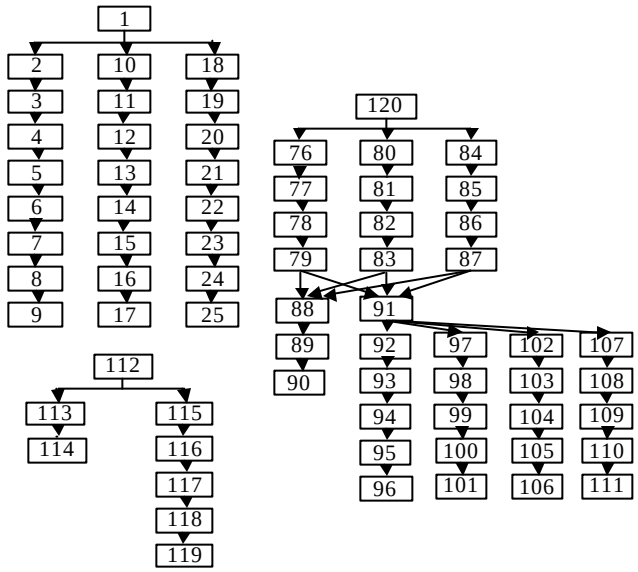

Figure 11. Structure of the DSP system

functions. For example, task 3 is an FIR filter, task 4 is an FFT function, task 5 is a Hanning Window, *etc.* There are 20 distinct types of communication edges as well, each of which transfers a different amount of data. The number of bytes transferred on these edges ranges from 1 to 32K. The periods of all task graphs and deadlines of the sink tasks are set to 3 seconds. In the resource library, we provide 10 different types of cores; each core has different attributes. Task-dependent parameters, such as a validity bit showing whether the task type can run on the given core type, the maximum execution time and average power consumption of each task on each valid core, *etc*., are also given. The communication link between different SOCs is a bus, with a packet size of 238 bits, communication speed of 40 Mbytes/s [27], and average power consumption of 0.33 W. Although off-chip communication is expensive both in time and power, this problem is alleviated by the heuristic employed in our algorithm which reduces inter-SOC communication.

Table 1. Experimental results for DSP example

	Cores on SOC1	Cores on SOC2	Cores on SOC3	System price ($)	Power dissipation (mW)
Sol 1	2 $Core_1$, 1 $Core_8$, 1 $Core_{10}$	1 $Core_7$, 1 $Core_{10}$	1 $Core_3$	878	443.3
Sol 2	2 $Core_1$, 1 $Core_3$, 1 $Core_8$, 1 $Core_{10}$	1 $Core_7$, 1 $Core_{10}$	1 $Core_3$	1050	437.8
Sol 3	2 $Core_2$, 1 $Core_4$, 1 $Core_6$, 1 $Core_7$, 2 $Core_{10}$	1 $Core_3$, 1 $Core_7$	2 $Core_6$	1297	421.9

We specify the maximum number of SOCs to be three, and the area constraint of each SOC to be 360 mm^2. Our framework yields three non-dominated solutions, as shown in Table 1. Price and power are optimized under hard real-time constraints and SOC area constraint. As one can see, among the different solutions, as the system price goes up, the power dissipation decreases, indicating a trade-off between the two. A system designer can choose one of these three solutions.

5.2 Results on randomized task graphs

We also used our framework to synthesize various randomized task graphs. The set of task graphs for each example was generated using TGFF [26], which is a randomized task graph and core generator that has been used by many researchers. TGFF generates the task graphs from a template by varying the seed for the random number generator per template. The parameters that need to be defined include the number of task graphs, average period and deadline, types and attributes of cores, whether a task type can be run on a given core type, *etc.* There are some SOC-specific features in the template, related to wire bit-width, floorplanning, inter-chip link, *etc.* For our experiments, we targeted large multi-rate task graphs. Each example contains ten task graphs with an average of 15 tasks and a variability of 12 tasks per task graph. The average task transition time *task_trans_time*, which includes communication and computation time, is set and the deadline of each sink task is equal to (*depth*+1)·*task_trans_time*, where *depth* is the number of tasks from the root of the task graph to the sink task. There are 15 core types, which have an average price of 100 USD with a variability of 80 USD. Each core has an average width and height of 6 mm and a variability of 3 mm, and a maximum frequency of 50 MHz with a variability of 25 MHz. There are 15 communication edge types, which require an average of 809.6 KB with a variability of 600 KB of data to be transferred between tasks. Tasks require an average of 16,000 cycles to execute with a variability of 15,000 cycles. Tasks dissipate an average energy of 4 nJ per cycle with a variability of 3.5 nJ per cycle. We use an SOC area constraint of 70 mm^2. The communication link is the same as the one in Section 5.1. Table 2 shows the results. The first column depicts the seed number. The second column shows the system price of various non-dominated solutions obtained. The third column gives the system power dissipation, and the last column shows the CPU time. For examples 1, 9 and 12, no valid solutions were found. Note that a valid solution is not guaranteed to exist for all TGFF task graphs.

Table 2. Experimental results for randomized task graphs

Example	System price ($)	Power dissipation (mW)	CPU time (min)
2	533	684.3	68.4
	538	544.9	
3	332	343.3	21.9
	399	323.9	
4	413	610.2	41.4
	545	251.3	
	621	235.2	
5	418	123.9	33.3
6	894	1513.1	126.1
7	307	345.3	20.7
8	459	797.1	50.4
10	659	193.9	19.3
11	456	98.0	66.9
13	463	91.7	39.8
14	493	385.1	54.0
15	395	977.3	34.2
	422	906.1	
	460	791.9	
	560	446.9	
	569	413.5	
	638	377.9	

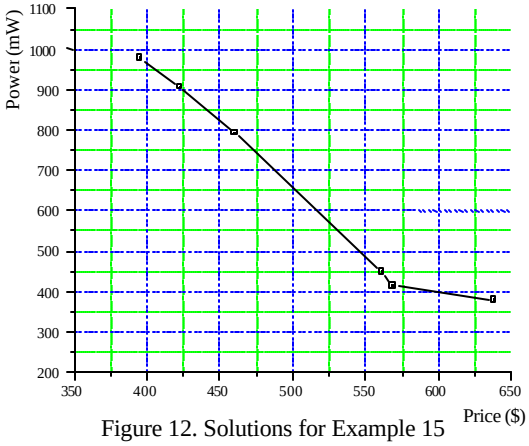

Figure 12. Solutions for Example 15

In Figure 12, we plot the price and power consumption of the six solutions for Example 15. The trade-off between power and price can be clearly seen.

6 Conclusions

In this paper, we presented an efficient method for functional partitioning integrated with a system synthesis algorithm for core-based, multi-chip, low-power, real-time, multi-rate, heterogeneous embedded systems. Under area and real-time constraints, the algorithm explores the design space efficiently and provides a set of non-dominated solutions to the designer which trade system price for power dissipation.

Our integrated framework shows that the homogeneous partitioning model is a viable one for obtaining low power distributed systems of SOCs. Experimental results indicate that high-quality solutions can be obtained in reasonable run-times.

References

[1] S. Govindarajan, V. Srinivasan, P. Lakshmikanthan, and R. Vemuri, "A technique for dynamic high-level exploration during behavioral partitioning for multi-device architectures," in *Proc. Int. Conf. VLSI Design*, pp. 212-219, Jan. 2000.

[2] F. Vahid, T. Le, and Y. Hsu, "Functional partitioning improvement over structure partitioning for packaging constraints and synthesis: Tool performance," *ACM Trans. Design Automation Electronic Systems*, vol. 3, no. 2, pp. 181-208, Apr. 1998.

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

[4] M. C. McFarland, "Computer-aided partitioning of behavioral hardware descriptions," in *Proc. Design Automation Conf.*, pp. 472-478, June 1983.

[5] E. D. Lagnese and D. E. Thomas, "Architectural partitioning for system level synthesis of integrated circuits," *IEEE Trans. Computer-Aided Design*, vol. 10, no. 7, pp. 847-860, July 1991.

[6] R. Gupta and G. De Micheli, "Partitioning of functional models of synchronous digital system," in *Proc. Int. Conf. Computer-Aided Design*, pp. 216-219, Nov. 1990.

[7] F. Vahid and D. D. Gajski, "Specification partitioning for system design," in *Proc. Design Automation Conf.*, pp. 219-224, June 1992.

[8] K. Kucukcakar and A. C. Parker, "CHOP: A constraint-driven system-level partitioner," in *Proc. Design Automation Conf.*, pp. 514-519, June 1991.

[9] V. Srinivasan, S. Radhakrishnan, and R. Vemuri, "Hardware software partitioning with integrated hardware design space exploration," in *Proc. Design Automation & Test in Europe Conf.*, pp. 23-26, Feb. 1998.

[10] Y. Chen, Y. Hsu, and C. King, "MULTIPAR: Behavioral partition for synthesizing multiprocessor architectures," *IEEE Trans. VLSI Systems*, vol. 2, no. 1, pp. 21-32, Mar. 1994.

[11] A. A. Duncan, D. C. Hendry, and P. Gray, "An overview of the COBRA-ABS high-level synthesis system for multi-FPGA systems," in *Proc. FPGAs for Custom Computing Machines*, pp. 106-115, Apr. 1998.

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

[13] S. Prakash and A. C. Parker, "SOS: Synthesis of application-specific heterogeneous multiprocessor systems," *Parallel & Distributed Computing*, pp. 338-351, 1992.

[14] B. P. Dave, G. Lakshminarayana, and N. K. Jha, "COSYN: Hardware-software co-synthesis of heterogeneous distributed embedded systems," *IEEE Trans. VLSI Systems*, vol. 7, no. 1, pp. 92-104, Mar. 1999.

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

[16] D. Saha, R. S. Mitra, and A. Basu, "Hardware software partitioning using genetic algorithm," in *Proc. Int. Conf. VLSI design*, pp. 155-160, Jan. 1997.

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

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

[19] K. S. Khouri, G. Lakshminarayana, and N. K. Jha, "High-level synthesis of low power control-flow intensive circuits," *IEEE Trans. Computer-Aided Design*, vol. 18, no. 12, pp. 1715-1729, Dec. 1999.

[20] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step toward software power optimization," *IEEE Trans. VLSI Systems*, vol. 2, no. 4, pp. 437-445, Apr. 1994.

[21] S. Gupta and F. Najm, "Power modeling for high-level power estimation," *IEEE Trans. VLSI Systems*, vol. 8, no. 1, pp. 18-29, Feb. 2000.

[22] R. P. Dick and N. K. Jha, "MOCSYN: Multiobjective core-based single-chip system synthesis," in *Proc. Design Automation & Test in Europe Conf.*, pp. 263-270, Mar. 1999.

[23] S. Govindarajan and R. Vemuri, "Tightly integrated design space exploration with spatial and temporal partitioning in SPARCS," in *Proc. Int. Conf. Field-Programmable Logic & Applications*, Aug. 2000.

[24] T. N. Bui and B. R. Moon, "Genetic algorithm and graph partitioning," *IEEE Trans. Computers*, vol. 45, no. 7, pp. 841-855, July 1996.

[25] C. M. Woodside and G. G. Monforton, "Fast allocation of processes in distributed and parallel systems," *IEEE Trans. Parallel & Distr. Syst.*, vol. 4, no. 2, pp. 164-174, Feb. 1993.

[26] 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.

[27] VIC068A: VMEbus Interface Controller, http://www.cypress.com/cypress/prodgate/busi/vic068a.html.