ASICosyn: Co-synthesis of Conditional Task Graphs with Custom ASICs

Yuan Xie, Wayne Wolf
Electrical Engineering Department
Princeton University
{yuaxie,wolf}@ee.princeton.edu

Abstract

This paper describes a co-synthesis system called ASICosyn, which is a co-synthesis tool for distributed real time embedded systems that considers various ASIC implementations for tasks. We use Monet, a behavioral level architectural exploration system from Mentor Graphics, to generate multiple implementations of a behavioral description of an ASIC and to analyze their performance. Mutual exclusive detection algorithm and corresponding allocation and scheduling algorithm are developed for conditional task graph to take care of the control dependencies in the task graph model.

1. Introduction

Hardware-software co-synthesis is the process of partitioning an embedded system specification into hardware and software modules to meet performance, power and cost goals [10]. This paper describes a co-synthesis tool for distributed, embedded computing systems. The algorithm synthesizes a distributed multiprocessor architecture and allocates processes to the CPUs and ASICs such that the allocation and scheduling meet the deadline of the system, while the cost of the system is minimized. The algorithm targets multi-rate periodic task graphs. The target architecture is a heterogeneous multiprocessor architecture that with multiple processing elements (PEs) of various types: general-purpose CPUs or domain-specific CPUs and ASICs. A CPU can execute different processes, while an ASIC executes a single task.

Previous work in the hardware-software co-synthesis area has addressed various aspects of this problem and provides valuable foundation to our work. However, most previous work has concentrated on solving the problem under a certain simplified assumption:

- In terms of the target architecture, some only consider the partitioning between one CPU and one ASIC, some work targets multi-processor architecture with ASICs, but do not take into account of different ASIC implementation for a specific task. Actually, for a specific task, it is possible to have different ASIC implementations: such as the fast implementation with large area or the slow implementation with small area. Our algorithm takes into account the impact of different ASIC implementations of tasks on system performance and cost in the co-synthesis process.
- In terms of the application model that used in the co-synthesis, many previous work use simple task graph model to capture the system specification, while our synthesis tool extends the model to be conditional task graph such that the control dependencies in applications can be taken care.

Elsewhere [11, 12] we have described specific features of our co-synthesis tool. This paper is a summary of our research and gives an overview of our co-synthesis system. The paper is organized as follows. The next section describes basics of hardware/software co-synthesis, Section 3 reviews previous related work. Section 4 describes different aspects of our co-synthesis system. Section 5 presents the implementation and finally we conclude and discuss possible future directions.

2. Hardware Software co-synthesis basics

Hardware/software co-synthesis tool constructs a cost-minimal distributed computing system which meets all system specification and constraints and simultaneously allocates and schedules the tasks onto the processing elements(CPUs and ASICs) in the distributed system. After the architecture is designed, lower-level synthesis algorithms can be used to implement more detail abstraction of the design. For example, a high-level synthesis tool can be used to implement the RTL level design of the ASICs.

The problem specification of our co-synthesis algorithm includes a set of real-time applications, an target architecture, and a technology library.

The real-time applications are periodic, running at multiple rates. A common model to describe the application is the task graph model [10]. Application is partitioned into task graph, which is a directed acyclic graph, as shown in Figure 1. In a task graph, nodes represent tasks that may have moderate to large granularity; the directed edges represent data dependencies between tasks. An edge, say A → E, implies that task E cannot start execution until A is finished. Data dependency edges ensure the correct order of execution. Each edge is associated with a scalar describing the amount of data that must be transferred between the two connected nodes. The task graph is executed periodically at its specified rate. We as-
sume that the deadline, by which the task graph must complete its execution, is equal to the period. The drawback of a task graph model is that it cannot capture the control flow dependencies among tasks. In order to overcome this drawback, we extend the task graph model to conditional task graph model. In the conditional task graph, an edge with an assigned condition value is a conditional edge (represented by dot lines in Figure 1). The task with output conditional edges is a branch fork task. Conditional paths meet at a branch join task. Depending on the condition, only a subset of the task graph is executed.

![Figure 1. Conditional Task graph.](image)

We use a heterogeneous shared memory multiprocessor as the target architecture as shown in Figure 2. The architecture has a number of processing elements (PES), which may be CPUs or ASICs. Each CPU has its private instruction cache and data cache. The task-level cache performance model we used is proposed by Li [6]. Each task can have many implementation options differing in PE type, cost and execution time.

![Figure 2. The target architecture.](image)

The technology library provides a number of choices of the types of processing elements to construct the system that meets performance requirements. The technology library includes general parameters such as manufacture cost for CPUs. The technology description must also provides information that relates the processing elements to the tasks. A table is required which gives the worst case execution time (WCET) for the tasks on each type CPU. If a task can be implemented as ASIC, then there is a related behavioral VHDL file for this task. An architectural exploration system called Monet [7], which is provide by Mentor Graphics, is used to determine the ASIC performance and cost.

The goal of the co-synthesis algorithm is to allocate processes to PEs and choose the number and types of components in the target architecture from the technology library, such that the applications can be scheduled to meet their performance constraints (deadlines) and the total cost of the result system is minimized.

Hardware/software partitioning algorithms were the first variety of co-synthesis applications. Partitioning algorithm implements the system specification on some sort of architectural template, usually a single CPU with one or more ASICs connected to the bus. On the other hand, distributed system co-synthesis does not use an architectural template to drive co-synthesis. Instead, it creates a multiprocessor architecture for the hardware engine. The target architecture is usually heterogeneous in both its processing elements and its communication channels. It can employ multiple CPUs, ASICs and FPGAs.

Task allocation and scheduling are important aspects of the co-synthesis process. The scheduling routine is not only used to generate the final allocation and schedules for the design, but also used in the inner loop of the co-synthesis to evaluate the performance of intermediate solutions, and to help to generate new solutions. Both its result quality and its efficiency are critical to the co-synthesis algorithm.

3. Previous work

Two distinct approaches exist in the distributed system co-synthesis: optimal and heuristic. An early example for optimal approaches is the SOS system [8], which uses mixed integer linear programming technique (MILP). Because of the time complexity, optimal approaches are suitable only for small task graphs and impractical. Thus people turn to the heuristic domain to find a solution quickly and efficiently. There are two distinct approaches in the heuristic domain: iterative and constructive. The iterative approach begins with an initial solution and improves it [9]. A constructive algorithm builds the solution step by step and a complete solution is not available until the algorithm is finished [1]. There are other heuristic algorithms such as MOGAC [3], which used genetic algorithm to optimize the system cost and power. Though many aspects of co-synthesis have been addressed, previous work did not consider the impact of different ASIC implementations on the final co-synthesis result.

Many heuristic scheduling algorithms are variants and extensions of list scheduling. But most of those work consider only the data dependencies in the task graph model, though scheduling of control and data flow graphs has been a very active research area in high level synthesis.
Only recently, people in the system level synthesis pay attentions to handle the control dependencies in the task graph model. Eles [4] described the conditional task graph model and used a list-scheduling algorithm to generates different schedule tables for all processing elements in the architecture. The schedule table lists all schedules for different condition combination in the task graph. Their algorithm assumes that the allocation of the tasks is fixed for each task, and enumerates all possible schedules for all condition combinations. Therefore it is not suitable for control-intensive application. Kuchcinski [5] used constraint programming techniques to model the scheduling problem of conditional task graphs. A commercial constraints solver is then used to find the solution. This approach is similar to ILP approach and is not suitable for large task graphs.

4. Co-synthesis framework: ASICosyn

In this section, we describe different aspects of our co-synthesis system ASICosyn, including the ASIC performance analysis, iterative improvement co-synthesis algorithm, and mutual exclusion detection for conditional task graphs etc.

4.1. ASIC performance analysis

Working together with the Mentor Graphics’ codesign consortium, we have developed an ASIC performance analysis tool, based on their architectural exploration system Monet [7], to explore the tradeoffs for different ASIC implementations in the co-synthesis framework.

![Figure 3. The design space of the ASIC implementations](image)

Monet [7] allows the user to explore architectural alternatives rapidly and to interactively make and evaluate tradeoff decisions at the behavioral level, before designs are committed to an RTL description. It is the first tool to combine interactive, graphical trade-off analysis with state-of-the-art automatic scheduling, allocation, and sharing. The user specifies the desired clock cycle time, expected control step to finish the process, as well as the I/O timing constraints. Then based on Monet’s synthesis result, we determine the worst case execution time (performance) and the area (cost) of the ASIC.

Using our ASIC performance analysis tool, one can explore the design space of the ASIC implementations. Figure 3 shows the design space for a specific ASIC implementations. In the real design, we might either choose a fastest implementation with expensive cost (larger area), or choose a slower implementation with smallest area, or other implementations between these two extreme points.

4.2. Co-synthesis algorithm

In this section, we describe different aspects of our co-synthesis system ASICosyn, including the ASIC performance analysis, iterative improvement co-synthesis algorithm, and mutual exclusion detection for conditional task graphs etc.

4.2.1. Outline of the co-synthesis algorithm

Our co-synthesis algorithm uses an iterative improvement strategy to search the design space. The outline of the algorithm consists of the following steps:

1. Pre-process and find an initial solution.
2. Iteratively reduce ASIC numbers and CPU cost.
3. ASIC cost reduction procedure.
4. Allocate and schedule tasks and data transfers for the final design.

In step 1, the pre-processor calls our ASIC performance analysis tool, which is based on Monet architectural exploration system. Given a behavioral VHDL description, our ASIC performance analysis tool calculate the speed (performance) and the area (cost) for the fastest ASIC implementation and the smallest ASIC implementation for each task which can be implemented as ASIC and put such information into the technology library. We assume that the cost of ASIC is proportional to the area of ASIC, which is reasonable in the system-level design.

The initial solution is constructed by assigning each task in the task graphs the fastest PE that is available for the task. This is done by looking up the technology library. If the PE is a CPU, then the instruction and data caches of the task’s program or data size are added to that CPU. If the PE is an ASIC, then the task is implemented as the fastest ASIC, whose parameters (the worse case execution time and the area) are extracted from our ASIC performance analysis tool in the pre-process procedure. The performance of the initial solution is evaluated, assuming the communication delay between PEs is zero. If it cannot meet the deadline constraints, there exists no feasible design given the current technology library, and the algorithm stops without a solution.

4.2.2. Iterative cost reduction procedure

Since the ASIC implementation is usually faster than the software implementation in CPU for a specific task,
most of the PEs in the initial architecture will be ASICs (or all ASICs). The iterative cost reduction is the most critical step in the co-synthesis algorithm. It searches for an improved design by moving tasks from ASIC to CPU and cutting CPU cost and cache cost iteratively.

A single iteration of the cost reduction step consists of two procedures: the ASIC-to-CPU procedure and the CPU cost reduction procedure. The ASIC-to-CPU procedure tries to move tasks from ASIC to CPU (from hardware to software), while the CPU cost reduction procedure reduces the CPU cost and cache cost.

In ASIC-to-CPU procedure, we use two heuristics to select tasks on ASICs and move them from ASICs to CPUs. The first heuristic is related to the difference between the hardware speed and software speed. If the difference is small, it implies that implementing the task as ASIC cannot achieve much speedup compared to implementing the task on CPU. Therefore this task is a good candidate to be slowed down by moving it from ASIC to CPU. The second heuristic uses the task graph's critical path. If a node is not on the critical path of the task graph, it implies this node is a good candidate to be moved from ASIC to CPU. Since the critical path moves as the tasks are moved around PEs, we use a simple method to decide the critical path and decide if a node is on the critical path [11].

The CPU cost reduction procedure tries to reduce the cost of the CPUs by eliminating lightly loaded CPUs after moving the tasks on those CPUs to other CPUs. The CPUs in the current design are ordered by their workload. The algorithm starts from the most lightly loaded CPU. For each CPU, we identify the tasks on it that can be executed on other CPUs; these tasks are then moved to the other CPUs that provide the best performance for the tasks; the cache sizes of the other CPUs increase to accommodate the tasks that are newly moved there. The CPU is removed if it becomes empty. When the tasks on a CPU cannot be moved to other CPUs, the algorithm tries to replace the current CPU with a cheaper alternative. Finally an attempt is made to cut its instruction and data cache size.

4.2.3. ASIC cost reduction.

After the iterative improvement procedure, the number of CPUs and the number of ASICs in the target architecture are fixed. But until now, for those tasks implemented as ASICs, we use the fastest implementation. Our ASIC cost reduction procedure tries to reduce the ASIC cost, by trying to replace the fastest ASIC implementations with the smaller but slower ASIC implementations. There are two types of slacks can be utilized to slow down the ASICs and reduce the area of ASICs: 1. global slack, which is the minimum slack between the deadline and the completion time of the tasks. 2. local slack, which is the minimum slack between the ASIC completion time and the starting time of its successor tasks.

The detail procedure has been described in our previous work [11]. Basically, we try to use less expensive (smaller area) ASIC implementations in the design where possible. We first replace the fastest ASIC implementations with its smallest implementation if it is feasible. For those fastest ASICs that can’t be replaced with the smallest implementations, the global slack and local slack are used to find intermediate implementations, with the facilitation of our ASIC performance analysis tool based on Monet.

4.3. Mutual exclusion detection

In a conditional task graph, if two tasks belong to different conditional branch paths that have conflicting condition combination, they are mutual exclusive and it is impossible for them to be executed together in a period. For example, in Figure 1, because of the conditional variable, either B or C is activated at each period, but not both. Therefore, if task B and C are allocated to the same CPU, they can have overlapping execution time when they are scheduled. By exploring the possibility of resource sharing among those mutual exclusive tasks, we can reduce the system cost.

We have used a branch labeling method [12] to decide the mutually exclusive tasks. Each node in the task graph is associated with branch information: 1. branch level, which is the number of branch fork tasks that have to be executed before reaching this task. 2. branch labels, which are the names of the branch fork tasks. 3. branch conditions, which are the condition values for each branch level in order to reach this task. For example, in Figure 1, the branch information for task C is: branch level is 1; branch label is A; branch condition is "True".

By using this scheme, it is easy to decide if two tasks are mutual exclusive. Readers are referred to our previous work [12] for the algorithms of calculating the branch information and the decision procedure to identify the mutual exclusive tasks.

4.4. Allocation and scheduling algorithm

The efficiency and the quality of the scheduling algorithm are very important to the quality of the co-synthesis result. Because during the co-synthesis process, when the architecture space is explored and the partition of tasks on software (CPUs) and hardware (ASICs) is generated, the scheduling routine is called in the inner loop of co-synthesis to evaluate the quality of intermediate solutions, and to help generate new solutions. Since it is used in the inner loop of co-synthesis process, it is called by the synthesis procedure many times.

Unlike the algorithm proposed by Eles [4], which assumes that the allocation of tasks on CPUs is fixed, our algorithm performs allocation of the tasks on CPUs and
scheduling of the tasks on CPUs and ASICs simultaneously, such that the algorithm can take advantage of the resource sharing among those mutual exclusive tasks that belong to different branches.

Our allocation and scheduling algorithm is similar to that designed by Li [6]. However, our approach takes into account mutually exclusive tasks identified by earlier phases. Figure 4 outlines the allocation and scheduling procedure.

1. for each task, calculate static_urgency(task)
2. if task is in ready list and partitioned on ASIC
   schedule task i; goto 9
4. else
5. for each ready task i and each CPU pe-j
6. Calculate dynamic_urgency(task-i, pe-j)
7. Schedule task-i on pe-j with maximum
   dynamic_urgency value
9. update ready list and goto 2 until all tasks are scheduled.

Figure 4. Outline of allocation and scheduling algorithm

The static urgency is calculated for each task based on the maximum distance of the task to the end task of the task graph. This is similar to the priority assignment in some list schedulers. The longest branch path is used to calculate the static urgency of a branch fork task.

The dynamic urgency is defined as:

\[ DU(task, CPU) = SU(task) - \max(ready.time(task), CPU available time) - WCET(task, CPU) \]

From the definition, it is obvious that the dynamic urgency is related to the following factors:

1. Static urgency (SU). If a task's SU is high, it implies that this task is a critical task and should be given a high priority.

2. The earliest start time of this task on the CPU. Note that the ready.time(task) takes into account the communication time from its predecessor. We assume that the communication time between two tasks on the same CPU is 0. Furthermore, the mutual exclusive communication edges can share the same communication link with overlapping time slot.

3. The worst case execution time (WCET) of this task on the CPU.

When the CPU available time is calculated, if the allocated task is mutually exclusive with the ready task, then these two tasks can share the same time slot of the processing element to share the resource.

Figure 5 is the subroutine in scheduler that detects the CPU available time for a task by using the information obtained from the mutual exclusive detection procedure, which is described in section 4.

```
PE_available(Task ready_task, CPU pe) //ready_task is the task attempt to be allocated on pe
{
1. if no task scheduled on pe, return 0;
2. psi = latest allocated task on pe;
3. if ready_ps is not mutual exclusive with psi return psi.completion_time;
4. else psi = previous scheduled task on pe, goto 3;
}
```

Figure 5. Calculation of CPU available time.

4.5. Data arrival time for a task

Previous research work assumes that a task cannot start until all its inputs are available. But it is not true for real-life example. For example, task has two inputs x and y. If the task is put on CPU, implementing as software, some functionality only depends on input x (the shading part in Figure 6) can start immediately when input x is available, without waiting for the arrival of input y.

Figure 6. Macro mechanism to model different data arrival time.

On the other hand, if the task is implemented as ASIC, the available time of input x and y can be the input as timing constraint to our performance estimation tool, which is based on Monet, such that for different data available time, we will have different ASIC implementation with different performance and cost. We modify the process model to use macro mechanism to tackle this problem. Users can define macro in the task graph. A macro task consists of several processes, which depend on different inputs. The task graph can be expanded to be a larger graph. Each node in the expanded graph is a process, except that:

1. If a macro task is implemented as software, the macro will be expended such that the whole task graph becomes bigger, and all processes of a macro must be put on the same CPU.
2. If a macro task is implemented as ASIC, its performance and cost is estimated as a whole by Monet, while the data arrival time information is taken by Monet as timing constraint.
3. The edges inside the macro represent "block context switching time", while a normal edge represent communication time between tasks.
4.6. Aperiodic task

Research on scheduling aperiodic task has been done intensively in real-time domain. Aperiodic task can be divided into two categories: hard aperiodic which has hard deadline and soft aperiodic task which has soft deadline. The scheduling algorithm can also be divided as static or dynamic (online scheduling). Online scheduling algorithms use resource reclaim, slack stealing or other methods to make decision to accept or reject the incoming aperiodic task. It is only applicable to soft aperiodic tasks.

In ASICosyn, only hard aperiodic task graph is considered. We use execution time slot reservation method [1] to handle the aperiodic tasks. In this method, a static scheduling is constructed, in which the execution time slots are reserved for aperiodic task. If an aperiodic task's execution time is $u$, and the deadline for the aperiodic task is $D$, then the number of reserve slots during a certain period $T$ is $[T/D - u]$.

5. Implementation

We have implemented our co-synthesis algorithm in C++. We designed the graphical user interface using Tcl/Tk. The whole co-synthesis framework is called ASICosyn. From the graphics user interface, users can input and edit the system specification, such as the task graph and technology library. Then intermediate synthesis datum are generated and feed into co-synthesis system. The result of the co-synthesis, such as the architecture and allocation and scheduling of the task graphs, will be display in the graphics user interface.

To evaluate our co-synthesis framework, we have run example task graphs from literature, such as examples used by Prakash and Li [8, 6]. We also test task graphs that are generated by TGFF, a random task graph generation tool developed in Princeton [2]. Note that those examples are not conditional task graphs and we have to modify them to be conditional task graphs. Our experiments show that ASICosyn is efficient and fast, taking into account of the impact of different ASIC implementation on the co-synthesis result, as well as the conditional execution in the task graphs.

Compared with the algorithm proposed by Elies [4], which generated schedule table for each condition combination, our global schedule is not necessary to be the optimal one for a subset of the task graph. But our global schedule guarantees the worst case schedule meets the system deadline requirement. This guarantee is important during co-synthesis, which has to find out the architecture that fits all cases. After the co-synthesis process, we can use on-line resource reclaim to perform on-line scheduling, which can produce a better schedule for a subset of task graph.

6. Conclusions and future work

In this paper, we described a co-synthesis tool called ASICosyn, which is the first co-synthesis framework that considers the impact of different ASIC implementations on both the system performance and the system cost. We extend the task graph model to handle the conditional delivery of data, with the facilitate of a mutual exclusion detection procedure. Corresponding allocation and scheduling algorithm that take advantage of the resource sharing among mutual exclusive tasks are developed.

While researchers have made a great deal of progress in the co-synthesis area, there remains a lot of challenges range from development of more accurate, more sophisticated models that can capture the characteristics of realistic applications to the development of larger, more realistic benchmarks that would be a great benefit in driving co-synthesis research in realistic directions.

7. Acknowledgement

This work was funded by Mentor Graphics with additional funding from NSF.

References