Co-synthesis with custom ASICs

Yuan Xie, Wayne Wolf

Electrical Engineering Department
Princeton University
Princeton, NJ 08540 USA
e-mail : {yuanxie,wolf}@ee.princeton.edu

Abstract - This paper introduces the first hardware/software co-synthesis algorithm that optimizes the implementations of ASICs that are used as processing elements for the embedded systems. Many real time embedded systems are composed of heterogeneous processing elements, such as general purpose CPUs, ASICs and FPGAs. Previous work has not considered how to select one of several possible ASIC implementations for a specific task. We have developed a heuristic iterative improvement algorithm for distributed embedded system co-synthesis. We use Monet, a behavioral level architectural exploration system, to generate multiple implementations of a behavioral description of an ASIC and to analyze their performance. To the best of our knowledge, this is the first co-synthesis algorithm that takes into account the impact of different ASIC implementations of tasks on system performance and cost in the co-synthesis process.

I. Introduction

Hardware-software co-synthesis creates an embedded system architecture to meet performance, power and cost goals[3]. This paper describes a new algorithm for the co-synthesis of 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 system functionality is specified by the periodic task graphs. The target architecture is a heterogeneous multiprocessor with multiple processing elements (PEs) of various types: CPUs and ASICs. A CPU can execute different processes, while an ASIC executes a single task. Furthermore, 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.

This paper is organized as follows. Section II reviews previous related work. Section III describes an ASIC performance analysis tool based on Monet, an architectural exploration system by the Mentor Graphics. We then present our iterative improvement co-synthesis algorithm. Section V discusses the experimental results of our co-synthesis algorithm and finally we propose the future work.

II. Previous work

Previous work in hardware software co-design has addressed various aspects of co-synthesis[1][2][3]. 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.

Two distinct approaches exist in the distributed system co-synthesis: optimal and heuristic. A typical example for optimal approaches is the SOS system[9], which uses mixed integer linear programming technique (MILP). Because of the time complexity, optimal approaches are suitable only for small task graphs and impractical. So 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[1][2]. A constructive algorithm builds the solution step by step and a complete solution is not available until the algorithm is finished [7]. There are other heuristic algorithms such as MOGAC[8], which deploys an adaptive multi-objective genetic algorithm, both cost and power consumption are optimized while hard real-time constraint is met.

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.

III. ASIC performance analysis

Working together with the Mentor Graphics' codesign consortium, we have developed an ASIC performance analysis tool in our algorithm, based on their architectural exploration system Monet, to explore the tradeoffs for

different ASIC implementations in the co-synthesis algorithm.

Monet[6] 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, it is possible to explore the design space of the ASIC implementations. Figure 1 shows the design space for a specific ASIC implementations. In the real design, we might either choose the fastest implementation with expensive cost (large area), or choose the cheapest implementation(small area) with rather slow execution time, or other implementations between these two extreme points.

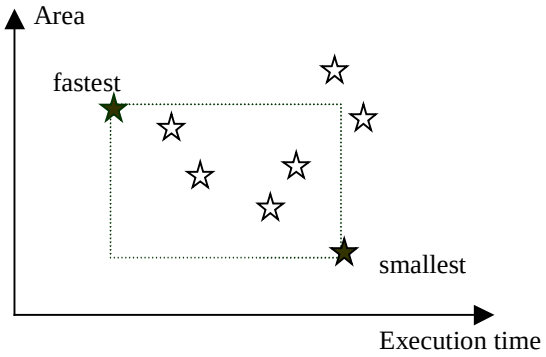

Fig.1. The design space of the ASIC implementations.

IV. Hardware/Software co-synthesis

A. Problem specification

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

The ***real-time applications*** are periodic, running at multiple rates. We use task graph model[3] to describe each application. Applications are partitioned into task graph, which is a directed acyclic graph, as shown in Figure 2. 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 → B, implies that task B 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 assume that the deadline, by which the task graph must complete its execution, is equal to the period.

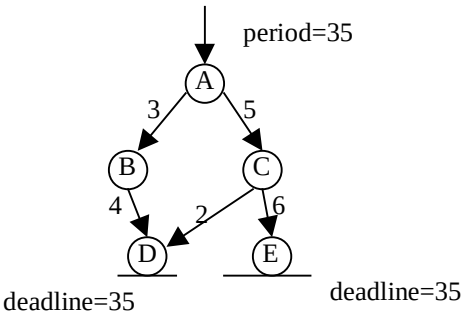

Fig.2. Task graph.

We use a heterogeneous shared memory multiprocessor as the ***target architecture*** as shown in Figure3.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[2]. Each task can have many implementation options differing in PE type, cost and execution time.

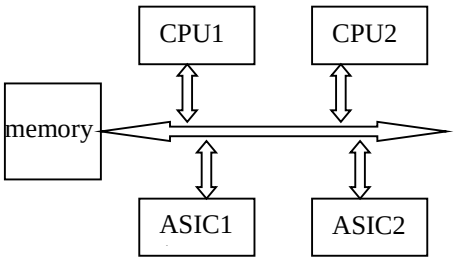

Fig.3. The target architecture

The ***technology library*** provides a number of choices of the types of CPUs and 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.

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.

B. 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[10].

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.

C. 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.

We use two heuristics to select tasks on ASIC and move them from ASICs to CPUs.

1. The first heuristic is related to the difference between the hardware speed and software speed. If the difference D (D=WCET_of_fastest_ASIC-WCET_of_fastest_CPU)is small, it implies that implementing the task as ASIC can't 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.

2. 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 **Earliest Start Time (EST)** for the node is the time at which the node can be executed as early as possible. For node_i, which is allocated to PE(node_i) , the EST(i) is defined as follows:

$$EST(i)=Max\{EST(j)+ET(j)+Com(i,j)\}$$
for all node_j, which are parent nodes of node_i

where the EST(j) is the EST of node_j, ET(j) is the execution time for node_j, Com(i,j) is the data communication time between node_j and node_i. If node_i and node_j are allocated on the same CPU, the Com(i,j) is zero. EST of node_i is simply the latest data arrival time among all its parent nodes. For start nodes, which don't have parent nodes, EST=0.

The **Critical Path Length (CPL)** is defined as the length of the critical path:

$$CPL=Max\{EST(j)+ET(j)\}$$ for all node_j in graph.

The **Latest Start Time (LST)** is the time at which the node can be executed as late as possible. The definition is:

$$LST(i)=Min\{LST(j)-ET(j)- Com(i,j)\}$$
for all node_j, which are child nodes of node_i.

For end nodes, which don't have child nodes, LST=CPL-ET(j).

After calculating the EST and the LST for each node in the task graph, if EST(node_i) =LST(node_i), then node_i is on the critical path.

A single iteration of 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 accommodates 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.

D. ASIC cost reduction.

After the iterative improvement procedure, the CPU structure and the ASIC structure is 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 implementation with the smallest ASIC implementation or the intermediate implementations. There are two types of slacks can be utilized to slow down the ASICs and reduce the area of ASICs.

- ***global slack***, which is the minimum slack between the deadline and the completion time of the tasks.
- ***local slack***, which is the minimum slack between the ASIC completion time and the starting time of its successor tasks.

```
1.  sort ASICs by decreasing cost difference
    (D=area_of_fastest –area_of_smallest)
2.  for each ASIC_i in sorted list
3.      replace the fastest ASIC with the smallest ASIC;
4.      if (meet deadline) use the smallest ASIC;
5.      else keep the fastest ASIC
6.  calculate the global slack;
7.  for each fastest ASIC_i ;
8.      slow down ASIC_i by (global_slack+local_slack_i);
9.      for each other ASICs
10.         Slow down ASIC by its local_slack
11.     call  Monet to get the total ASIC cost  COST(i)}
12. select the minimal COST(i) and select  corresponding
    ASIC implementations.
```

Fig.3. The ASIC cost reduction procedure.

We try to use less expensive (smaller area) ASIC implementations in the design where possible. All the ASICs in the current design are ordered by the difference of area between the fastest ASIC implementation and the smallest ASIC implementation. (line 1). We start from the ASIC with the largest D, try to replace that fastest ASIC with its smallest implementation (line 3). If it is not feasible, we keep the fastest ASIC implementation (line 5). After the loop (line 2—5), The ASICs in the design are either the fastest or the smallest implementation. For those fastest ASICs that can't be replaced with the smallest implementations, we begin to find the intermediate implementations. First, the global slack is calculated (line 6), then for each fastest ASIC, we calculate its local slack and slow down the ASIC speed by (global_slack +local_slack) (line 8) and slow down other fastest ASICs by their own local slack (line 10). If the schedule meet the deadline, we call our ASIC performance analysis tool to get the cost of current ASIC setting (line 11), and keep the cost and setting (line 11). But we do not change the system permanently for the time being. After the loop (line7 – line11), we choose the ASIC setting with the minimum cost and change the system permanently (line 12) and calculate the final schedule and allocation.

E. Task allocation and scheduling algorithm

Task allocation and scheduling are important aspects of the co-synthesis algorithm. The scheduling procedure generates the allocation and schedule of the design, its result is used to evaluate the performance of intermediate solutions and help generate new solution. The scheduler in our algorithm is similar to that designed by Sih and Lee [5] as well as Li[2]. This allocation and scheduling algorithm is less expensive than the other existing algorithms and gives an effective schedule. Also, it can balance the load on the hardware structure. This makes the algorithm suitable for use in the design space exploration of our co-synthesis algorithm, in which an allocation and scheduling algorithm is called repeatedly.

V. Experimental Result

We have implemented our co-synthesis algorithm, which required about 5000 lines of C++. Also, we designed the graphical user interface written by Tcl/Tk. The whole co-synthesis framework is called ***ASICosyn***. All of our experiments were run on Pentium Pro 200. We used examples from related co-synthesis research to evaluate our algorithm: PP1 and PP2 are Prakash and Parker's example1 and example2 [9], ex1 is one of Li's examples[2].

	Cost ($)	***Prakash/Parker*** CPU time on Solboume 5/e/900 (sec)	***LI/WOLF*** CPU time on Pentium Pro 200 (sec)	***COSYN*** CPU time on Sparc 20 (sec)	ASICosyn CPU time on Pentium Pro 200 (sec)
PP1	5	37	0.10	0.20	0.08
PP2	10	7.2 hr	0.26	0.54	0.20

Table 1. Comparison with other algorithms

Since PP1 and PP2 examples have no ASIC model, we first ran our algorithm by constructing the initial solution to be all CPUs. The table1 shows that the optimal algorithm is very time-consuming. While Li/Wolf[2] (an iterative improvement algorithm) and COSYN[7] (a constructive algorithm) as well as our algorithm can get the solution with the same cost as Prakash and Parker′s optimal algorithm[7] within 1 second.

The we ran ASICosyn with different ASIC implementations for PP2 and ex1. For PP2 example, Li/Wolf and COSYN, as well as Prakash and Paker's algorithm, can find a solution with cost 10. Our algorithm can get a solution with cost 8.4. For ex1 example, Li/Wolf's algorithm returns a solution with cost 140, while ASICosyn can return a solution with cost 105. If we shorten the deadline (ex1-1), their algorithm can not get a feasible

solution, while ASICosyn can still return a slightly more expensive solution by speeding up ASIC a little bit.

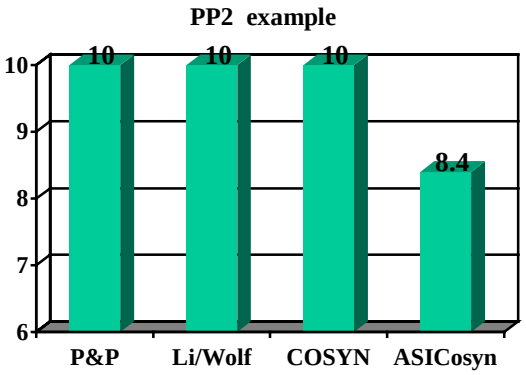

Fig.4. PP2 example.

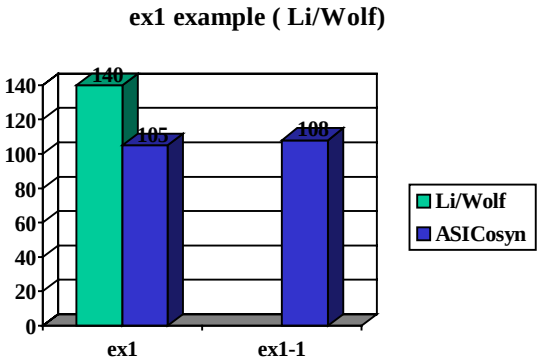

Fig.5. ex1 example.

VI. Conclusions and future work

In this paper, we described a new co-synthesis algorithm. To the best of our knowledge, this is the first co-synthesis framework that considers the impact of different ASIC implementations on both the system performance and the system cost. Our algorithm synthesizes complex multi-rate real-time applications onto a heterogeneous multiprocessor architecture to meet real-time deadlines at minimal cost. In order to explore the ASIC design space, we develop an ASIC performance analysis tool based on Mentor Graphics' Monet architectural exploration system, which can help us to explore the tradeoffs between ASIC performance and cost.

Future work includes the development and demonstration of synthesis from the extended flow graphs. Currently the task graph model in co-synthesis research is too simple to handle more complicated phenomena. We are going to extend the task graph model such that it can handle more complicated phenomena such as conditional delivery of data and time delays on delivery of data.

Acknowledgements

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

References

[1]Wayne Wolf, "An architectural co-synthesis algorithm for distributed, embedded computing systems" , IEEE transaction on VLSI, vol.5, No.2, pp. 218-229,June 1997.
[2]Yanbing Li, "Hardware-Software co-synthesis of embedded real-time multiprocessor", Ph.D. dissertation, Princeton, 1998
[3] Wayne Wolf and Jorgen Staunstrup, "Hardware/Software co-design: Principles and Practice", Kluwer Academic Publishers, 1997
[4] Wayne Wolf and Ti-Yen Yen, " Hardware-software co-synthesis of distribute embedded systems" , Kluwer Academic Publishers. 1996
[5] G.Shi and E.A.Lee, "A compile-time scheduling heuristic for interconnection constrained heterogeneous processor architectures", G.Shi and E.A.Lee. IEEE transactions on Parallel and Distributed systems,vol.4,no.2,pp.175-187, Feb.1993
[6] Monet reference manual, Mentor Graphics company.
[7] B.Dave and N.K.Jha, "COSYN: Hardware-software co-synthesis of embedded systems" , Proceedings of the 34th ACM/IEEE DAC ,pp.703-708, June, 1997
[8] R.Dick and N.K.Jha, "MOGAC: A multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems", Proceedings, ICCAD,pp.522-529, Nov.1997
[9] S.Prakash, A.Parker, "SOS: synthesis of application specific heterogeneous multiprocessor systems", Journal of Parallel and Distributed computing, vol 16, pp.338-351,1992
[10]M.Potkonjak, W.Wolf, "Cost optimization in ASIC implementation of periodic hard-real time systems using behavioral synthesis techniques",Proceedings, ICCAD, pp.446-451, Nov1995
[11]Y.Kwok, I.Ahmad,"Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors", IEEE transactions on parallel and distributed systems, vol.7,No.5, pp.506-521, May 1996.