Allocation and scheduling of conditional task graph in hardware/software co-synthesis

Yuan Xie and Wayne Wolf
Electrical Engineering Department
Princeton University
Princeton, NJ 08540, USA
{yuanxie,wolf}@ee.princeton.edu

Abstract

This paper introduces an allocation and scheduling algorithm that efficiently handles conditional execution in multi-rate embedded system. Control dependencies are introduced into the task graph model. We propose a mutual exclusion detection algorithm that helps the scheduling algorithm to exploit the resource sharing. Allocation and scheduling are performed simultaneously to take advantage of the resource sharing among those mutual exclusive tasks. The algorithm is fast and efficient,and so is suitable to be used in the inner loop of our hardware/software co-synthesis framework which must call the scheduling routine many times.

1. Instructions

Hardware/software co-synthesis partitions an embedded system specification into hardware and software modules to meet performance, power and cost goals.A common model to describe the system specification is the task graph [1]. The task graph has a structure similar to a data flow graph, except that the tasks in a task graph represent larger units of functionality. Allocation and scheduling of a set of data-dependent tasks, which are described by task graphs, on a multiprocessor architecture has been intensively researched. But most previous work that used a task graph model that has no control dependency information can only capture data dependency in the system specification. Recently, some researchers in co-synthesis domain have tried to use conditional task graph to capture both data dependencies and control dependencies of the system specification [7][8]. However, because of the time complexity, their algorithms are not very suitable for large task graphs or to be used in the inner loop of the co-synthesis procedure.

This paper describes an allocation and scheduling algorithm that is used in the inner loop of our co-synthesis process of distributed, embedded computing systems[11]. The co-synthesis process synthesizes a distributed multiprocessor architecture and allocates processes to the target architecture, such that the allocation and scheduling of the task graph meets the deadline of the system, while the cost of the system is minimized. The algorithm targets periodic multirate task graphs. The target architecture is a heterogeneous multiprocessor architecture that consists of multiple processing elements (PEs) of various types: general-purpose CPUs or domain-specific CPUs and ASICs. The allocation and scheduling algorithm has been implemented as part of our co-synthesis tool [11], which is the first co-synthesis tool that takes into account the impact of different custom ASIC implementations of tasks on system performance and cost in the co-synthesis process.

This paper is organized as follows. Section 2 reviews previous related work. Section 3 describes the problem formulation and section 4 introduces a method to detect mutual exclusion among tasks. We then present our scheduling and allocation algorithm. Finally we discuss the experimental results of our algorithm.

2. Related Work

Previous work in hardware/software co-design has addressed various aspects of HW/SW co-synthesis. Hardware/software partitioning algorithms implement 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 creates a multiprocessor architecture. 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 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.

Some scheduling problems can be modeled as integer linear programming (ILP) problems. An ILP solver is used to find the optimal solution. An earlier example for optimal approaches is the SOS system, which used mixed integer linear programming technique (MILP) [5]. Because of the time complexity, optimal approaches are suitable only for small task graphs and impractical. Heuristic approach is by far the most widely used approach. Many heuristic scheduling algorithms are variants and extensions of list scheduling [1][2][4]. But most of those works 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 [9][10]. Recently, people in the system level synthesis pay attentions to handle the control dependencies in the task graph model. Eles *et al.* described the conditional task graph model and used a list-scheduling algorithm to generates different schedule tables for all processing elements in the architecture [7][13]. The schedule table lists all schedules for different condition combination in the task graph. Their algorithm has two limitations: 1. They assume that the allocation of the tasks is fixed for each task; 2. Their algorithm has to enumerate all possible schedules for all condition combinations, thus it is not suitable for control-intensive application. Kuchcinski *et al.* [8] used constraint programming techniques to model the scheduling problem of conditional task graphs, and then used commercial constraints solver to find the solution. This approach does not take advantage of heuristics unique to co-synthesis.

3. Problem formulation

Many **real-time applications** are periodic, running at multiple rates. We use a **task graph** model [3] to describe each application. Application is partitioned into task graph, which is a directed acyclic graph, as shown in Figure 1. In a task graph, each node represents task that may have moderate to large granularity; the directed edges represent data dependencies between tasks. An edge, say $A \rightarrow D$,implies that task D 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 tasks, which decides the communication time between the tasks if they are allocated onto different PE. We assume that if two tasks are allocated onto the same PE, the communication time is 0. An edge with an assigned condition value is a conditional edge (represented with dot lines in Figure 1). The task with output conditional edge is a branch fork task. Conditional paths meet at a branch join task. For example, in Figure 1, A is a branch fork task, and E is a branch join task. Depending on the condition, one of the out-branches of task A ($A \rightarrow B$ or $A \rightarrow C$) is activated. The task graph is executed periodically at its specified rate. For simplicity, in this paper, we assume that the deadline, by which the task graph must complete its execution, is equal to the period. The deadline can actually shorter or longer than the period.

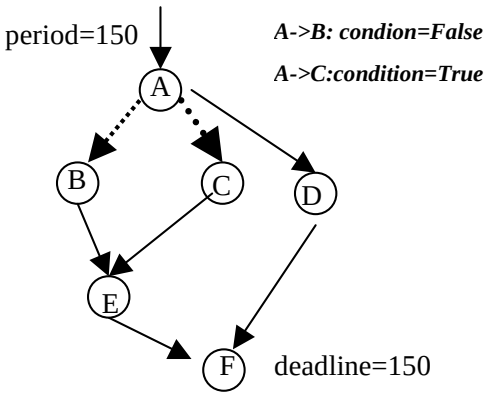

Figure 1. Conditional task graph

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, which are communicating with each other via communication links (such as a bus). 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.

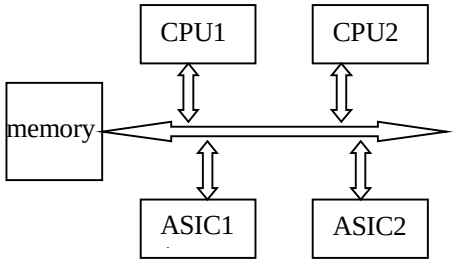

Figure 2. The target architecture.

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. Part of the technology library for the example in Figure 1 is shown in Table 1. The WCET of a task on CPU can be estimated using the techniques described in [14]. 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

[6], which is provided by Mentor Graphics, is used to determine the performance and cost of the ASIC. By using Monet as an estimation tool for the custom ASICs, our co-synthesis system [11] can explore the tradeoffs of different ASIC implementations during the co-synthesis process.

Tasks	WCET on CPU1	WCET on CPU2
A	10	20
B	20	18
C	30	20
D	30	10
E	10	20
F	30	20
Cost of CPU1=100		Cost of CPU2=150

Table 1. Part of the technology library.

Given the conditional task graph, target architecture and technology library, the co-synthesis algorithm produces an allocation of tasks on target architecture and constructs the static global schedule of the tasks on specified processing elements.

4. Detection of mutual exclusion

In the 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 at the same time. For example in Figure 1, depends on the condition value, task B or task C is executed but not both. If task B and task C are allocated to the same CPU, they can have overlapping execution time when they are scheduled, since they are mutual exclusive.

We can use a branch labeling method, which is similar to the branch numbering procedure described in [9], to identify the mutually exclusive tasks. Each task in the task graph is associated with a branch information structure, which is defined as following:

```
struct branch_info{
        int level;
        branch_label[i];
        branch_condition[i];
        };
```

level the number of branch fork tasks that have to be executed before reaching this task.

branch_label[i] the name of the i*th* level branch fork task.

branch_condition[i] the condition value for the i*th* level branch.

For the example in Figure 3, The task A and F are not in any branch, so their branch level is 0 and does not have branch_label and branch_condition information. Task J belongs to one of the branches beginning from C, as well as one of the branches beginning from I. When condition at C is C3 and at I is I1, task J is executed. So the branch level of task J is 2, and the branch lable=[C, I], branch_condition=[C3,I1]. We can use the algorithm shown in Figure 4 to go through the task graph and calculate the branch_info struct for each task. The result is shown in Table 2. The fictitious branch joint tasks are created in Figure 3 for outlining control structure. Note that in [7][13], their conditional process graph model only consider boolean condition while our approach allows multiple conditions (such as the switch statement in C language).

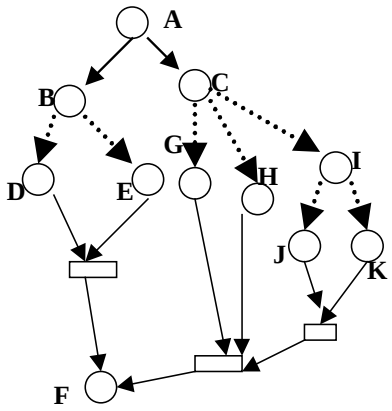

Figure 3. An example conditional task graph.

```
Branch_labeling(task  ps)
{
if  (ps is the branch join task) {
          delete ps->branch_label[ps.level] and
          ps->branch_condition[ps.level];
          ps.level--;}
foreach child task ps_child of ps{
    if(ps is the branch fork task)
      {   i =ps_child.level++;
          ps_child->branch_label[i] =ps.name;
          ps_child->branch_condition[i]
                    =branch_condition;}
   else
          ps_child has the same branch_info as ps
  }
Branch_labeling(ps_child)
}
```

Figure 4. Branch-labeling recursion algorithm outline.

Node	level	Branch_label	Branch_condition
A	0	N/A	N/A
B	0	N/A	N/A
C	0	N/A	N/A
D	1	B	B1
E	1	B	B2
G	1	C	C1
H	1	C	C2
I	1	C	C3
J	2	C I	C3 I1
K	2	C I	C3 I2
F	0	N/A	N/A

Table2. The branch_info struct for Figure3.

By using this scheme, it is easy to decide if two tasks are mutual exclusive. For any two tasks,

1. If the level of one task is 0, then they are not mutual exclusive, such as task A and task I.

2. If $task1.level = n1$, $task2.level = n2$, let N=min(n1,n2), then we compare the first N branch_label and branch_condition, beginning from $branch\_label[1]$ and $branch\_condition[1]$:

   a if $task1.branch\_label[i] \neq task2.branch\_label[i]$, they are not mutual exclusive, such as task I and task D.

   b if $task1.branch\_label[i] = task2.branch\_label[i]$ and $task1.branch\_condition[i] \neq task2.branch\_condition[i]$, they are mutual exclusive, such as G and J.

   c else, compare the i+1 level. If $i > N$, they are not exclusive.

By using the same scheme, we can decide the mutual exclusive communication edges. For example, the communication edges $C \rightarrow G$ and $C \rightarrow H$ in Figure 4 are mutual exclusive, if they are scheduled on the same communication link, they can have overlapping execution time slot. Note that two tasks on different branches might not be mutual exclusive, such as D and I.

5. Allocation and scheduling algorithm

During the co-synthesis process [2], when the architecture space is explored and the partition of tasks on software (CPUs) and hardware (ASICs) is generated, a scheduling routine is used in the inner loop of co-synthesis to evaluate the performance of intermediate solutions, and to help to generate new solutions. Since it is used in the inner loop of co-synthesis process, it is called by the synthesis procedure many times. The efficiency and the quality of the scheduling are very important to the quality of the co-synthesis result.

Our scheduling 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. This is different from the algorithms proposed in [7][13], which assume that the allocation of tasks on CPUs is fixed.

Our allocation and scheduling algorithm is similar to that designed by Li [2] and Sih [4]. However, our approach takes into account mutually exclusive tasks identified by earlier phases. Figure 5 outlines the allocation and scheduling procedure.

```
1.  for each task, calculate static_urgency(task)
2.  if there is task i in ready list is partitioned on ASIC
2.        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 maximium
8.        dynamic_urgency value
9.  update ready task list and goto 2 until all tasks  are
    scheduled.
```

Figure 5. 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. For example, in Figure 1, we assume that the communication time for each edge is 1, and task D is partitioned to be implemented as ASIC. The execution time of D as ASIC is estimated by using Monet, as we mentioned in section 3. Suppose that the execution time for D on ASIC is 5, then the static urgency (SU) of the tasks are shown as Table 3. Note that the weight for each task that allocated on CPUs is calculated as the average WCET on CPUs, user can also specify to use the mediate WCET as the weight for the task. For the tasks that are allocated to the ASIC, the weight is exactly the estimated WCET from Monet by taking the VHDL description of that task as the input. The longest branch path is used to calculate the static urgency of a branch fork task.

Task	A	B	C	D	E	F
SU	83	61	67	31	41	25

Table3. The static urgency for the example in Figure 1.

The **dynamic urgency** is defined as:

DU(task,CPU)=SU(task)
- max{ready_time(task),CPU available time}
- WCET(task, CPU)

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 6 is the subroutine in the scheduler that determines 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.ps1=latest allocated task on pe;
 3.if ready_ps is not mutual exclusive with ps1
       return ps1.completion_time;
 4.else  ps1=previou scheduled task on pe,
       goto 3   }
```

Figure 6. Calculation of CPU available time.

Table 4 shows the first several steps to schedule the task graph in figure 1 on the target architecture with one CPU1 and one CPU2. Task D is implemented as ASIC, which is decided by the iterative improvement procedure during the co-synthesis procedure[2].

	SU	DU(task, CPU)			Schedule (from,to, PE)
A	83	73, 63	-- , --	--, --	(0,10, CPU1)
B	61	-- , --	29, 31	29, 31	(12,30, CPU2)
C	67	-- , --	25, 35	-- , --	(12,32, CPU2)
D	31	-- , --	-- , --	-- , --	(11,16, ASIC)
A-- > D					(10,11, BUS)
A-- > C					(11,12, BUS)
A-- > B					(11,12, BUS)

Table 4. The first several scheduling steps for figure 1.

After A is scheduled on CPU1 from 0 to 10, tasks B,C,D are all ready to be scheduled. Since D is partitioned as ASIC, D and the communication edge from A to D are scheduled. Then, C is allocated and scheduled on CPU2 since DU(C,CPU2)=35 is the greatest among all combination of B,C with CPU1 and CPU2. When DU(B,CPU2) is calculated, since B and C are mutual exclusive, B can still be allocated and scheduled on CPU2 and have overlapping schedule time with task C. Furthermore, the communication edge $A \rightarrow C$ and $A \rightarrow B$ are mutual exclusive and so can be scheduled on the bus at the same time slot.

6. Experimental results

We have implemented our conditional task graph allocation and scheduling algorithm as part of our co-synthesis framework [11]. For the example shown in Figure 3, the schedule on two CPUs is shown in Figure 7 (we did not show the communication schedule here). By using our mutual exclusion test scheme, task G, H and J are mutual exclusive, so they can have overlap execution time slot on CPU1. Similarly, D and E are mutual exclusive, they can share CPU2 at the same time. But for task K and E, they are not mutual exclusive though they belong to different conditional branch, so K must wait until E is done.

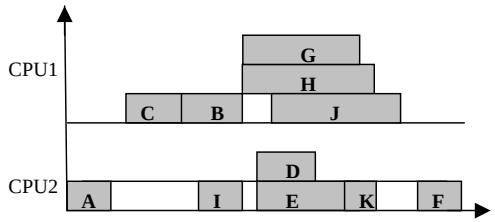

Figure 7. Scheduling result for example in fugure 2.

Compared with the algorithms proposed by Eles *et al.* and Pop[7][13], 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. For example, under a certain condition, only a subset of task graph A, C, B, D, G, F is executed. For this subset of task graph, a shorter schedule exists. But our global schedule guarantees the worst case schedule is available. For example, the global schedule is fine for subset A, C, B, E, I, J, F, which has the longest schedule length in this case. 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 online scheduling, which can produce a better schedule for a subset of task graph. Furthermore, the ASIC cost reduction procedure presented in [11] can also be used to reduce the ASIC cost, which is facilitated by Monet.

We ran our algorithm with examples from Eles *et al.* and Pop[7][13], and compared our schedule with their schedule table, which has different schedules for each subset of the task graph. We found out that our global schedule corresponds to the worse case schedule for each task in their schedule table.

We also did some statistical experiments. We modified TGFF [12] to generate an extensive number of random conditional task graphs with different structures. The task graphs were then allocated and scheduled on various architectures. Table 5 shows part of the experimental results.

	#of tasks	#of edges	# of condition	# of CPU	Ave. Time (sec)
EX1	32	56	4	3	0.02
EX2	65	96	3	3	0.07
EX3	105	168	2	2	0.14
EX3	105	168	3	2	0.15
EX3	105	168	5	2	0.18

Table 5. Part of the experimental results.

The fourth column shows the number of condition branch fork tasks in the task graph. The fifth column shows the number of CPUs in the target architecture (We did not show ASIC information here, our co-synthesis algorithm in

[11] decided the partitioning of task graph between CPUs and ASICs). The last column shows the average running time of the algorithm on a Celeron 533 computer with Linux. Example EX3 shows that even though the condition variables increase, which means the number of execution subset of the task graph increase, the running time of the algorithm does not increase dramatically. The reason is that our algorithm did not schedule each subset of the task graph corresponding to each condition combination separately. On the contrary, it calculates the mutual exclusion information for the whole task graph and then schedules the whole task graph by exploring the resource sharing among those mutual exclusive tasks.

7. Conclusion

This paper introduces an allocation and scheduling algorithm that is used in the inner loop of co-synthesis procedure. The conditional task graph is targeted with the facilitation of a mutual exclusion detection procedure.

8. Acknowlegement

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 University,* 1998.

[3] Wayne Wolf and Jorgen Staunstrup, *"Hardware/Software co-design Principles and Practice"*, Kluwer Academic Publishers, 1997.

[4] G.Sih and E.A.Lee, "A compile-time scheduling heuristic for interconnection constrained heterogeneous processor architectures", *IEEE transactions on Parallel and Distributed systems*,vol.4,no.2,pp.175-187,Feb. 1993.

[5] Prakash, Parker, "SOS: synthesis of application specific heterogeneous multiprocessor systems", *Journal of Parallel and Distributed computing*, vol. 16,pp.338-351 1992.

[6] Monet reference manual, Mentor Graphics Company.

[7] Petru Eles *et al.* "Scheduling of conditional process graphs for the synthesis of embedded systems", *Proceedings of DATE*, pp.23-26, 1998.

[8] K.Kuchcinski, "Embedded system synthesis by timing constraint sovling", *Proc. Int.Symp.on Syst.synth*,pp.50-57,1997.

[9] Said Amellal *et al.* "Functional synthesis of digital systems with TASS", *IEEE Trans. on CAD*,vol.13, No.5, pp.537-551,1994.

[10] G.Lakshminarayana,N.Jha,"Wavesched: A novel scheduling technique for control-flow intensive behavioral descriptions", *Proceeding of ICCAD*, pp.244-250, 1997.

[11] Yuan Xie,Wayne Wolf,"Co-synthesis with custom ASIC", *Proceedings of ASP-DAC2000*,pp.129-135,2000.

[12] R.P.Dick,D.L.Rhodes, W.Wolf, "TGFF: Task graphs for free", *Proc.Int. Workshop Hardware/Software Codesign*, pp.97-101, Mar.1998.

[13] Paul Pop, "Scheduling and communication synthesis for distributed real-time systems", *Ph.D. thesis, Linkopings University*,2000.

[14] Y-T.S.Li,"Performance analysis of real-time embedded software", *Ph.D. thesis,Princeton University*,1997.