Scheduling under Data and Control Dependencies for Heterogeneous Architectures

Alex Doboli
Department of ECECS
University of Cincinnati
Cincinnati, OH 45221-0030
adoboli@ececs.uc.edu

Petru Eles
Dept. of Computer and Information Science
Linkoping University
581 83 Linkoping, Sweden
petel@ida.liu.se

Abstract

This paper presents a list-scheduling algorithm for graphs with data and control dependencies. We assume that tasks are partitioned between hardware resources as scheduling takes place after partitioning in our co-synthesis tool. Control dependencies are introduced by if statements, and model complementary functionalities. A detailed discussion of our algorithm is presented. Extensive experimental work shows the effectiveness of our method for generating close-to-optima schedules in short run-times.

1. Introduction

Hardware/software co-synthesis describes the translation of a system-level specification into an implementation that consists of separate hardware and software units and their interface elements, under certain design constraints. A typical co-synthesis tool performs three exploration steps, iteratively [3]: resource allocation, partitioning of functionality on resources, and scheduling. Resource allocation decides the number and type of processing elements (general purpose processors, ASICs), as well as the number and type of communication units (buses, shared memory). Partitioning maps system functionality to processing elements, and interprocessor communications to communication units. Finally, scheduling sequentializes task executions on processing elements, and communications on communication units. Any effective scheduling algorithm requires precise knowledge of task execution times. For multiprocessor systems, task execution time dramatically depends on the resource type. Thus in order to compute tight estimations of task execution times, scheduling has to follow partitioning.

This paper presents a static, non-preemptive scheduling algorithm for partitioned task-graphs. We extend the list-scheduling algorithm [4] for handling control dependencies, apart from data dependencies, so that the worst-case execution time of a graph is minimized. We assume that tasks are partitioned between resources, as scheduling takes place after partitioning in the co-synthesis tool. Thus, the priority function used by the algorithm considers the available information about partitioning. Second, we suppose that tasks interact through both data and control dependencies. Data dependencies describe data flows between tasks. Control dependencies express the behavior, where some parts of the global functionality execute, if and only if, specific control conditions are true. Conditional behavior enables conditional resource sharing, and is very effective for real-life applications, i.e. communication or process control systems.

This algorithm is part of a co-synthesis tool that performs hierarchical exploration of the design space. One of its main goals compared to other existing tools, is to allow efficient design decisions for conditional resource sharing. Currently, it considers a fixed architecture, and searches after the implementation with shortest execution time. Partitioning maps time-critical, intense computational tasks to hardware, while less critical tasks are assigned to software. Additionally, it tries to minimize communications between different processors, while maximizing system concurrency.

The next section describes some related work undertaken in this area. The problem is defined in Section 3 with help of an example. Section 4 discusses the scheduling decisions to be considered when handling control dependencies. Section 5 presents our algorithm for scheduling task-graphs with data and control dependencies. Section 6 details experimental results, and finally, section 7 presents our conclusions.

2. Related work

Non-preemptive scheduling was proven to be an NP-complete problem [9]. Thus, attention shifted to finding efficient heuristics that provide close to optima solutions in a short time. Static list-scheduling [13] [4], at any execution step, selects the ready task with highest priority. Priority reflects the optimization goal considered for scheduling, which is usually the length of the global schedule. For partitioned graphs, critical path by its own can be irrelevant, mainly because of the requirement to execute tasks assigned to the same resource, sequentially. Besides, present list-scheduling methods handle only data-flow graphs.

Resource allocation, partitioning, and task scheduling are formulated as an integer linear programming problem in [1]. The problem is described as a set of equations, in-equations, and a cost function (implementation cost) that is minimized. This method is simple and finds optimal solutions, but its run-time increases exponentially with the size of the graphs. Moreover, it does not handle control dependencies.

Scheduling of control and data flow graphs has been a very active research area in HLS [4]. Path-based scheduling [2] produces all possible execution paths, schedules them independently, and combines them in a final schedule. A list-scheduling method based on condition vectors is introduced in [8]. A general limitation of the two algorithms is that they do not take careful scheduling decisions across basic block boundaries. [11] discusses a speculative computation based scheduling algorithm, that optimizes the high probability execution paths. If the path is predicted incorrectly, then a prediction error correction phase takes place. The correction phase overhead might become large for designs at levels of granularity higher than instruction level. In general, HLS considers a lower level of granularity, as opposed to our work, that uses a coarse-grain, task-level approach, specific to system-level synthesis.

Unlike other hardware/software co-synthesis work, our scheduling method emphasizes the more general goal of conditional resource sharing. In this paper, we extend list-scheduling for coping with branch-related control constructs by taking efficient decisions across branch structure boundaries. The peculiarity of the co-synthesis flow motivated us to use partitioned graphs, and consider partitioning information for defining priorities for scheduling.

3. Problem formulation

Representation of partitioned task-graphs with data and control dependencies

A *data and control dependency graph* (DCG) is the input for our scheduling algorithm. *Figure 1* presents a DCG example with 10 nodes. Nodes denote either computations (i.e. nodes 2, 3, ... 8), or data communications between nodes on different processors (i.e. communication between nodes 2 and 3). The graph has a single *root node* and a single *end node*, both with execution times 0. Nodes are connected through directed arcs to express an imposed execution order: a target can start its execution only after its source completed. In order to express control dependencies, some arcs are annotated with condition values. In *Figure 1*, conditions are depicted in italics. Node 2 computes condition *cond 1*, and depending on its value, *one* of its out-branches

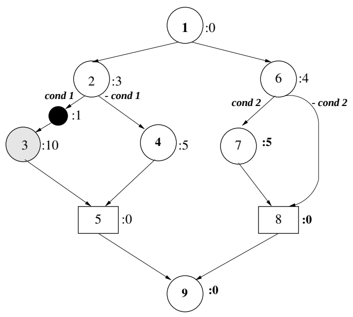

Figure 1. Example of a Data and Control Dependency Graph.

is selected. For a *true* value, the graph traversal activates communication between nodes 2 and 3, followed by node 3. Node 4 is executed for a *false* condition value (*false* is indicated by - *cond 1*). Nodes 5 and 8 are fictitious *join nodes*, which are created for the purpose of outlining control structures. Each node is annotated with its execution time, and its partitioning to resources is represented through shading. Communications are shown as small, black circles.

For this example, we assume that all resources serially execute the nodes assigned to them. In general, application specific processors (*hardware processors*) can concurrently run their assigned tasks as they maintain multiple execution flows. General purpose, programmable processors (*software processors*) and communication units (buses, etc.) execute serially their assigned functionalities.

The semantics of our execution model is similar to that of a single rate system [7]. It assumes that each node is executed not more than once for each traversal of the graph.

Scheduling with data and control dependencies

Figure 2 presents the Gantt chart for a correct, possible schedule for the example in *Figure 1*. This figure does not include activations of nodes 1, 5, 8, and 9, as these are fictitious. Graph execution always starts with the root node (i.e. node 1), which has 0 execution time. At time 0, the current set of computed conditions is void, and node 2 is selected to execute next on processor 1. Node 2 computes the value for condition *cond 1*. Thus, at time step 3, when node 2 ends, the value of condition *cond 1* becomes known on processor 1. Hence, processor 1's current set of computed conditions changes, as it now includes *cond 1*. Processor 2 has still an empty set of computed conditions, because the value of *cond 1* has not yet been communicated to it. When node 6 finishes, the set of computed conditions for processor 1 changes again, as it also contains condition *cond 2*.

Examining scheduling sequences for different condition combinations, it becomes obvious that they can include different nodes. Even for the same nodes, they might plan distinct scheduling times, i.e. node 4 has different schedul-

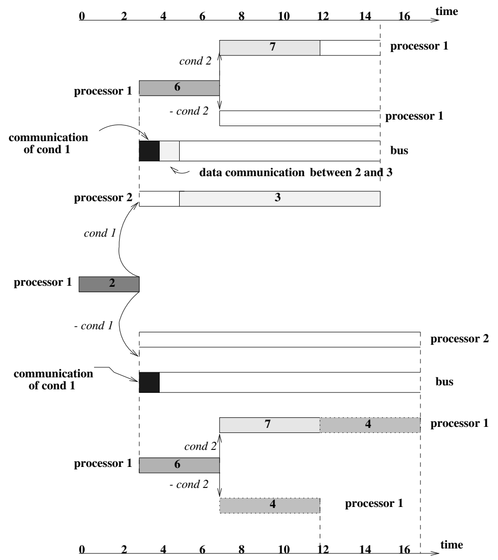

Figure 2. Schedule for the DCG in Figure 1.

ing times for condition combinations {- *cond 1, cond 2*} and {- *cond 1, - cond 2*}.

The task-graph's total execution time differs for the four possible condition combinations: 15 for {*cond 1, cond 2*} and {*cond 1, - cond 2*}, 17 for {- *cond 1, cond 2*}, and 12 for {- *cond 1, - cond 2*}. The longest schedule corresponds to condition combination {- *cond 1, cond 2*}, and represents the task-graph's *worst-case execution time*.

The execution semantics for our DCG model states that for a single traversal of the task-graph, each task is performed at most once. This requirement must be fulfilled by each produced schedule. It is equivalent to asserting that if a task *P* is scheduled for a condition set *E*, then it is necessary that during the same execution of the task-graph, *P* is not activated for another condition set *E'*. *Conflict* is the case when the requirement is not satisfied, and it corresponds to incorrect scheduling. In [5], we proved that if task *P* is scheduled for condition sets *E* and *E'*, then the scheduling of *P* introduces a conflict if $E' \subseteq E$ or $E \subseteq E'$.

Each processor of the final implementation has its own, *local scheduling table*. Each scheduling table has as many columns as the number of different condition combinations that can occur during task execution on its processor. *Our algorithm computes correct, local scheduling tables for each processor of an architecture so that the worst-case execution time is minimized.*

4. Modeling of control dependencies for scheduling of partitioned graphs

The scheduling example in Section 3 shows, that conditions can influence node scheduling in two different ways:

- A condition decides whether or not a node is executed. Such conditions are *guard conditions* for the node, and all its guards represent the *list of guards*, i.e. in *Figure 1*, node 3 has *cond 1* as a guard. Lists of guards have a static nature and are a consequence of the conditional alternatives to which a node belongs.

- A condition doesn't control a node's execution, instead, influences its scheduling time. Such conditions are called *influence conditions*, and all influence conditions of a node represent its *list of influence*. Influence conditions are useful for taking scheduling decisions across control structure boundaries. *Figure 2* shows that the scheduling time for node 4 depends on the value of condition *cond 2*. Thus, *cond 2* is an influence condition for node 4. The dependency of the scheduling time of node 4 on condition *cond 2* is a consequence of node 7, with *cond 2* in its guard list, being executed before node 4. If node 4 were to be scheduled before node 7, then its scheduling time would no longer depend on condition *cond 2*, and thus, *cond 2* wouldn't any longer be in its influence list. Influence lists depend not only on partitioning of nodes to resources, but also on previous scheduling decisions. In order to accommodate the dynamic character of influence lists with the static nature of our scheduling algorithm (refer Section 5), we reinterpret influences in a static way. We consider that a node's influence conditions are those that do *not* appear in its guard list, and fulfill one of the criteria:
  - the node is a successor of a node with the condition in its guard list
  - the node is a successor of a node with the condition in its influence list
  - the node is on the same resource as a node with the condition in its guard or influence list

Although pessimistic, this approximation permits a static computation of influence lists. Understanding the behavior of influence conditions is very important as we proved that scheduling conflicts are due only to influence conditions [5].

Figure 2 shows that condition *cond 1*, computed on processor 1, is used for scheduling decisions on processor 2. However, this condition must be communicated first. In order to communicate condition values over the architecture, the scheduling algorithm has to decide the following:

- The target processors for a condition communication: Each processor must know all condition values, required for taking scheduling decisions for its assigned nodes. Our algorithm communicates a condition value to all processors, which execute at least one node, that has the condition in its guard or influence list. For example, in *Figure 1*, condition *cond 1* is communicated to processor 2.

- The starting time of a condition communication: It is required that at a node's scheduling time, each of its guard conditions are known on the node's processor. Influence conditions have to be known if they have already been used in scheduling one of the node's predecessors. To guarantee conflict free schedules, the two constraints impose that *a condition has to be transmitted so that it is received not later than the earliest moment when on another processor, a node with one of the previous two constraints satisfied must start its execution*. The time step when a processor receives a condition value is equal to the sum between the starting time of the communication and the communication time. We assumed that communication times are identical for all condition values. Nevertheless, the rule is not sufficient as it decides a "time interval" for the start of a condition communication, rather than an exact time step. In accordance with the rest of the scheduling algorithm, we implicitly decide the condition communication time by defining a rule for calculating its priority (refer Section 5).

- The bus to be used for a condition communication: Condition communications are the only activities in a DCG, which are not partitioned yet to any resource. This is natural, as during partitioning, it is hard to decide the condition values that have to be communicated. Thus, the scheduling algorithm has to assign a bus for each condition communication. Our heuristic rule assumes that after a condition communication becomes ready to be scheduled, it will use the first available bus. This rule tries to avoid the delay of tasks that use this condition for their scheduling.

5. List-scheduling for partitioned DCG

Static list-scheduling assigns a single priority to each node in order to solve execution conflicts [4]. If two nodes are ready for execution on the same resource, then the one with the higher priority is scheduled first. In this section, we propose a list-scheduling method that considers control dependencies too. Also, we suggest an effective priority function for partitioned graphs.

5.1. List-scheduling for DCGs

As we already described in Section 3, for task-graphs with data and control dependencies, the set of executed nodes depends on conditional values. Each possible condition combination activates a specific set of nodes, called *trace*. If traces are scheduled separately, then node priority and node execution order are specific to a trace. It is very common that the same pair of nodes have different priorities and relative execution orders for different traces. Moreover, the graph no longer has a unique total execution time, because it depends on the current condition combination. Thus, in oder to capture all mentioned aspects, we believe that *a node of a DCG should be rather characterized by a set of priorities than a single value*. This set includes all priorities that correspond to traces, which execute the specific node.

In an ideal situation, all nodes are scheduled according to their priority for the current trace. Unfortunately, for a DCG, the current trace can be identified only during late stages of the scheduling algorithm, when all conditions in the graph become known. This indeterminism is actually an unavoidable consequence of the existing control dependencies. We solve conditional indeterminism by using a heuristic rule: *nodes are scheduled according to the requirements imposed by the most constrained situations, in which they can be involved*. This strategy makes sense, as the global execution time of the task is fixed by constrained situations. By encouraging strongly constrained paths, there is also a high chance that the overall graph execution time is improved.

Condition communication priorities can not be defined similarly to the priorities for the rest of the nodes. This is mainly because the set of receivers of a condition communication are not statically captured by the DCG, but are determined only during scheduling. Our heuristic rule for condition communication priorities states that their priority is equal to the highest priority of a node that relies on the condition value for its scheduling. This rule avoids the delay of important data communications, without disfavoring task nodes that use the condition value for their scheduling.

Our list-scheduling algorithm for DCGs implements, mainly, the following steps. Before scheduling actually starts, it computes all possible condition combinations for a graph, and identifies the subgraph executed for each combination. Next, it computes node priorities for the identified subgraphs. The set of priorities assigned to one node corresponds to all condition combinations that execute it. Then, it calculates the guard and influence lists for each node.

Figure 3 depicts our scheduling algorithm. It traverses the graph and analyzes each possible condition combination. The algorithm considers only nodes executed for the current condition combination, and eliminates those mutually excluded. For a certain scheduling step, the parameter *computed conditions* indicates the set of all computed conditions, and *list of ready nodes* includes all nodes that were activated by the *computed conditions*, with all their predecessors having been executed.

Subroutine *next node* finds in *list of ready nodes*, the node to execute next. It returns *current node*, which is the ready node with the highest priority. Subsection 5.2 details the

```
void schedule (list of ready nodes, computed conditions) {
  while(current node ≠NULL and current node ≠conditional node) {
      current node = next node (list of ready nodes, computed conditions);
      current time = time moment for scheduling of current node;
      case (current node) is {
      communication:
          current table = table of processor that runs
           the source of current node;
          break;
      condition communication:
          current table = table of processor that runs
           the conditional node for current node;
          break;
      task:
          current table = table of processor that runs
           current node;
          break;
      }
      time for required conditions = time step when all required
        conditions are known to the processor running
        current node;
      mark current node into current table, at time
        maximum (time for required conditions, current time);
  }
  if (current node ≠ NULL and
     current node is a conditional node) {
      current condition = condition calculated by current node;
      schedule (list of ready nodes ∪ ready nodes from the true
         branch, computed conditions ∪ true current condition);
      schedule (list of ready nodes ∪ ready nodes from the false
         branch, computed conditions ∪ false current condition);
  }
}
```

Figure 3. List-scheduling algorithm for DCGs.

priority function that we used. If the node to be executed next is a condition communication, then subroutine *next node* decides the bus to carry it. *current time* is the earliest time moment when *current node* can start.

Our scheduling algorithm generates different scheduling tables for all processing elements in the architecture. The scheduling of *current node* is marked into different local scheduling tables, depending on its nature. *current table* points to this local table. Nodes that correspond to tasks are registered in the local table of their processor. Communication nodes are marked in the scheduling table of their source node processor, as these nodes initiate them. Finally, scheduling of condition communications are shown in the scheduling tables of their conditional node processor.

Some condition values required for scheduling are contained in ongoing condition communications. These values have to be communicated before the node execution starts. *time for required conditions* indicates the time step at which they are known on the processor running *current node*. Thus, in order to avoid scheduling conflicts (refer Section 4), *current node* starts at the maximum of *time for required conditions* and *current time* .

After a conditional node executes, the value of its condition becomes available, and is pointed by *current condition*. We preserve the consistency of our scheduling algorithm by branching the graph traversal according to the two possible situations. *schedule* is called recursively for nodes in the *true* branch, and separately for those in the *false* branch. This guarantees that mutually exclusive branches are not

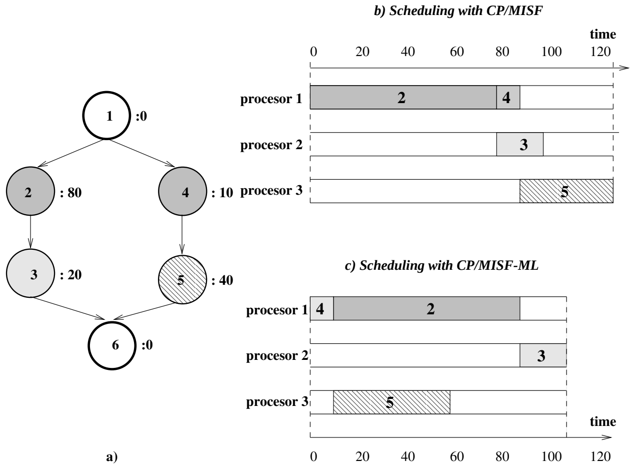

Figure 4. Comparison of CP/MISF-ML with CP/MISF.

activated simultaneously, and conditional resource sharing is achieved.

In order to complete our algorithm, we have to decide a proper priority function for scheduling partitioned graphs. The next paragraph introduces the priority we suggest.

5.2. Priority definition for list-scheduling of partitioned graph

Traditional list-scheduling priorities are defined using only critical path information [6] [13] [4]. This approach is reasonable because the critical path reflects a node's importance towards achieving a short graph-execution time. However, there is a major difference for partitioned graphs, because node execution on software processors or bus communications have to be serialized. Thus, we strongly believe that *an effective priority for partitioned graphs should also include partitioning knowledge*. Additionally, such a priority should be kept simple, so that the computational complexity of the scheduling algorithm remains low.

Our statement is motivated by the simple example in *Figure 4a*. Each node is annotated with its execution time and node partitioning is shown through shading in the figure. Nodes 2 and 4 are run on a software processor. For the simplicity of this example, we assume that all communications take 0 execution time.

CP/MISF [6] is a typical critical-path based priority function. It defines a node priority as the length of the longest path from that node to the sink. If two nodes have the same priority, then the one with the greater number of immediate successors is scheduled first. List-scheduling with CP/MISF produces the schedule in *Figure 4b* (the schedule does not represent dummy nodes 0 and 6). The total graph execution time is 130 time units, and *Figure 4b* shows that the overall

DCG	Nr of nodes	Nr of condit.	Nr of traces	Nr of columns	Architecture			sched. length	dev. (%)	unch. traces	max. incr.(%)	time (sec)
					process.	nr buses	ASIC area					
F4-Mode1	32	5	6	11	2 X Pent.	1	-	1932	0	5	0	0.009
F4-Mode1	32	5	6	11	2 X 486	1	-	2932	0	5	0	0.009
F4-Mode2	23	2	3	5	2 X Pent.	1	-	1167	0	2	0	0.003
F4-Mode2	23	2	3	5	2 X 486	1	-	1732	0	2	0	0.003
F4-Mode3	42	7	8	15	2 X Pent.	1	-	3548	0	7	0	0.024
F4-Mode3	42	7	8	15	2 X 486	1	-	5033	0	7	0	0.024
mean value	40	2	4	7	SPARC	1	30000	830	0	3	0	0.025
mean value	40	2	4	7	SPARC	1	60000	646	0	3	0	0.025
parall. chains20	20	3	8	15	2	1	-	120	12	0	36	0.004
parall. chains60	60	5	32	63	5	2	-	136	7	24	7	0.054
parall. chains120	120	5	32	63	6	2	-	146	2	24	2	0.087
parall. quicks.	23	2	4	7	2	1	-	56600	0	3	0	0.004
fork-join	20	2	4	7	2	1	-	42	0	3	0	0.0039
Laplace equ.solv.	22	1	2	3	2	1	-	62	0	0	3	0.0027

Table 1. Results of the DCG scheduling algorithm for real-life examples.

system concurrency is very poor.

This example suggests that an efficient priority for partitioned graphs should combine both critical path and partitioning informations. In [5], we defined and motivated the priority function, which we called CP/MISF-ML (Critical Path/Most Immediate Successor First - Modified Level): *the priority of a node is equal to the longest path from its first successor (not necessarily direct successor), which is assigned to a different resource than the current node, to the sink*. According to CP/MISF-ML, node 4 has priority 40 (length of path starting at node 5, as node 5 is already on another resource than node 4). For the example in *Figure 4a*, CP/MISF-ML produces the schedule in *Figure 4c*. The major difference from the previous case is that node 4 is scheduled before 2, as it has a higher priority now. After 4 executes, node 2 starts, while node 5 runs concurrently on another resource. The total execution time of the graph is 110. This improvement is due to a higher system concurrency than for CP/MISF.

We validated CP/MISF-ML by an extensive amount of experiments [5]. In the majority of cases, CP/MISF-ML found optimal schedules for graphs with only data dependencies. Heuristic schedules were on average by 2.10% longer than optima, and deviations from optima for individual schedules were less than 21.10%. Based on our experimental results, we decided to use CP/MISF-ML as a priority function for scheduling partitioned graphs.

6. Experimental results

Our experimental work has two parts. First, we used a statistical approach. We produced an extensive number of random graphs with different structures, and with their nodes partitioned to various architectures. Second, we used a set of examples to validate the effectiveness of our algorithm for real-life situations. Our main goal was to observe the quality of the schedules produced by our algorithm. Also, we were interested in finding patterns for situations in which our method constantly offered poor solutions.

The characteristics of the statistical evaluation part are detailed in [5]. Basically, it targets the two specific factors for scheduling under the existence of control dependencies. We analyzed the way in which structuring of control dependencies influences the quality of scheduling results. This is important because control structuring defines the number of traces of a DCG, and implicitly, the control related complexity, that must be handled during scheduling. Also, we studied the dependency of schedule quality on node repartitioning to conditional branches. Node repartitioning to branches determines the potential of different traces in defining the worst-case situation.

The quality of scheduling results was defined in terms of the average and worst-case relative deviations. Relative deviation relates the difference of the global schedule length, found by our DCG scheduling algorithm, and the longest individual trace, to the longest individual trace. Individual trace lengths are found using list-scheduling for each separate, data-flow trace. The global characterization of the algorithm shows that average deviations are less then 9%. In worst-case, relative deviation was about 80%, but such situations were extremely rare. It is worth to be mentioned that for graphs with 60 nodes, for about 90% of cases, the relative deviation was 0%. We learned that the relative deviation increases with the conditional complexity. This is quite natural as the approximation introduced by the heuristic nature of this algorithm tends to be higher.

Our experimental work also considers a set of real-life examples. This is basically to prove the algorithm's effectiveness for real-life cases. Second, we were interested in having a closer look on potentially weak aspects like trace number explosion, and trace schedule degradation. *F4-Mode1*, *F4-Mode2* and *F4-Mode3* model the operation and maintenance functionality of the F4 level of an ATM switch [10]. These examples were experimented considering an architecture with two Pentium processors, and then, one with two Intel486 processors. *mean value* [12] performs the mean value analysis, and was adapted for our scheduling purposes. *parall. quicks.* is a quick-sort implementation for MIMD machines. It has an out-tree task-graph structure. *parall. chains20*, *parall. chains60*, *parall. chains120*, *fork-join*, and *Laplace equ.solv.* are adapted after [13], and represent three popular task-graph structures for distributed systems: parallel chains, fork-join, and a sequence of in-out trees.

Our results are tabulated and presented in *Table 1*. Columns two and three show the graph characteristics: number of nodes and conditions. Column four presents the number of traces handled during scheduling, and column five indicates the complexity (number of columns) of the resulted scheduling tables. The next three columns indicate the implementation architectures we used. Column nine shows the total schedule length, resulted after running our list-scheduling algorithm, while next we indicate its relative deviation. Columns eleven and twelve point how shorter traces are affected by our algorithm. We observed how many traces remain unchanged (compared to their individual schedule), and which is the maximal relative increase that results. The last column shows the algorithm execution time on a Sun Ultra 30 computer.

Table 1 indicates that, in general, trace schedules produced by our algorithm remain unchanged compared to their individual data-flow schedules. The algorithm assigns proper priorities to nodes, and orders their execution so that schedule lengths are minimized. Second, the number of traces and the complexity of the scheduling table can remain low, even for a high number of conditions (i.e. *F4-Mode3* with 7 conditions). Mainly, this happens for graphs with a nested conditional structure. Finally, the execution time grows with the number of nodes, but it remains low even for graphs with many nodes and a complicate control structure. Thus, we believe that this algorithm can be successfully used inside a co-synthesis exploration loop.

7. Conclusions

This paper presents a list-scheduling algorithm for scheduling partitioned graphs under both data and control dependencies. Control dependencies are addressed by our algorithm by assigning to each node a list of priorities, according to all situations that activate it. The algorithm calculates a static scheduling table for each processor in the architecture, thus, the scheduling overhead of the final implementation remains low.

Experimental work proved that our list-scheduling algorithm generates high-quality schedules in a reasonable long execution time. Also, we learned that partitioning information is beneficial for improving the quality of the scheduling results in terms of both average and worst-case results. The drawback of our approach is primarily because of two aspects: modeling capability and scheduling strategies. We address only single-rate, periodic systems. Multi-rate applications require the use of loop constructs, while aperiodic tasks are addressed by real-time issues. Secondly, there is always the danger for trace number explosion, although the examples we used proved the contrary. We insist that our approach is not meant for control-dominated applications, where FSM based methods are required. Finally, for certain, random examples, the schedule degradation was very high. We identified that such cases occur mainly for graphs with high number of interacting traces, where our scheduling decisions proved to be incomplete. We suggest some sort of *look-ahead* information to improve the scheduling quality.

References

[1] A. Bender. Design of an optimal loosely coupled heterogeneous multiprocessor system. *Proceedings of the ED&TC Conference*, 1996.

[2] R. Camposano. Path-based scheduling for synthesis, vol.10, no.1. *IEEE Transactions on CAD*, pages 85–93, 1991.

[3] D. Gajski, F. Vahid. Specification and design of embedded hardware/software systems. *IEEE Design & Test of Computers*, pages 53–67, Spring 1995.

[4] D. Gajski, N. Dutt, A. Wu. *High Level Synthesis*. Kluwer Academic Publishers, 1992.

[5] A. Doboli. *Contributions to Hardware/Software Co-Design for Multiprocessor Systems*. PHD Thesis, Technical University Timisoara, 1997.

[6] H. Kasahara, S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Trans. on Computers, no.11*, pages 1023–1029, November 1984.

[7] J. Vanhoof, et al. *High-Level Synthesis for Real-Time Digital Signal Processing*. Kluwer Academic Publishers, 1993.

[8] K. Wakabayashi, H. Tanaka. Global scheduling independent of control dependencies based on condition vectors. *Proceedings of the 29th DAC*, pages 112–115, 1992.

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

[10] T.M. Che S.S.Liu. *ATM Switching Systems*. Artech House Books, 1995.

[11] U. Holtmann, R. Ernst. Combining mbp-speculative computation and loop pipelining in high-level synthesis. *Proceedings of the ED & TC*, pages 550–556, 1995.

[12] V. Srinivasan, S. Radhakrishnan, R. Vemuri. Hardware software partitioning with integrated hardware design space exploration. *Proceedings of DATE'98*, 1998.

[13] Y.-K. Kwok, I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Trans. on Parallel and Distributed Systems, vol.7*, pages 506–521, March 1996.