Critical Path Driven Cosynthesis for Heterogeneous Target Architectures

Bjørn-Jørgensen, Peter; Madsen, Jan

Published in:
International Workshop on Hardware/Software Codesign

Link to article, DOI:
10.1109/HSC.1997.584573

Publication date:
1997

Document Version
Publisher's PDF, also known as Version of record

Critical Path Driven Cosynthesis for Heterogeneous Target Architectures

Peter Bjørn-Jørgensen Jan Madsen

Department of Information Technology
Technical University of Denmark

Abstract

This paper presents a critical path driven algorithm to produce a static schedule of a single-rate system onto a heterogeneous target architecture. Our algorithm is a list based scheduling algorithm which concurrently assigns tasks to processors and allocates nets to interprocessor communication. Experimental results show that our algorithm is able to find good results, as compared to other methods, in small amount of CPU time.

1. Introduction

Embedded systems are usually implemented using a mixture of technologies including off-the-shelf components, such as microprocessors, and dedicated hardware, such as full- or semi-custom ASICs. This results in a heterogeneous architecture, in which also the communication links between the components uses different technologies, i.e. point-to-point communication and busses with various bandwidths.

In this paper we address the problem of cosynthesis of single-rate systems onto a heterogeneous target architecture. In particular, we solve the problem of mapping a system behavior onto a given target architecture, i.e. performing a task scheduling.

Our scheduling objective is to minimize the schedule length, taking into account interprocessor communication overhead due to data dependencies between tasks and the heterogeneity of the target architecture.

2. Related Work

Many multiprocessor scheduling approaches have taken interprocessor communication into account, e.g. [1, 2, 3, 6, 7], but only very few have addressed the scheduling onto a *heterogeneous* target architecture, e.g. [5, 8]. Most of these approaches are based on list scheduling and, hence, mainly differs in the way they select among the tasks which are ready to execute. The system behavior is typically represented as a directed acyclic graph of tasks (or processes)[1] called a task graph.

A popular algorithm is the Highest Levels First with Estimated Times (HLFET) [1]. The *static level* (SL) of a task is defined as the largest sum of computation times along any path from the task to an end-task. Among the tasks ready to execute, the algorithm selects the one with the largest SL, i.e. the most critical, and schedules it on the first available processor. Communication is not taken into account during task and processor selection.

Wu and Gajski [6] propose two scheduling algorithms. The Most Critical Path algorithm (MCP) and the Mobility Directed (MD) algorithm. The MCP algorithm first computes the *latest start time* (LST) for each task. Then each task is associated with a list of tasks containing itself followed by its successors in decreasing order of LST. The algorithm then constructs a list of tasks in an increasing lexicographical order of the task lists. While there are tasks in the list, the algorithm selects the top task and schedules it on the processor which results in the *earliest start time* (EST).

The MD algorithm selects tasks based on the relative mobilities (RM) of the tasks. RM is defined as the difference between EST and LST of a task divided by the computation time. The algorithm schedules the task with the smallest RM on the processor with the earliest time slot large enough to hold the execution of the task.

Hwang et. al. [2] propose an algorithm called Earliest Task First (ETF). Among the tasks ready to execute, the algorithm selects the task and processor which results in the earliest start time, EST.

The Dominant Sequence Clustering (DSC) [7] algorithm has been proposed by Yang and Gerasoulis. It defines the dominant sequence (DS) length of a task as the longest path from any entry-task to any end-task containing the task itself. The path length is calculated as the sum of computation and communication times along the path. The task with the highest DS is selected. If the task is not ready, the algorithm

[1] In this paper we will use the term *task* for the smallest computing entity, as opposed to *process* which is used by some authors.

0-8186-7895-X/97 $10.00 © 1997 IEEE

selects among its predecessors the one with the highest DS. The EST determines where to schedule the selected task.

Kwok and Ahmad [3] presents an algorithm called Dynamic Critical-Path scheduling (DCP). The algorithm first schedules all tasks on different processors in order to find EST and LST. The task with the minimum LST-EST is then selected and scheduled on the processor that have the earliest time slot large enough to execute the task. The algorithm continues until all tasks are scheduled.

Sih and Lee [5] propose the Dynamic List Scheduling (DLS) algorithm which select tasks based on the largest Dynamic Level (DL) which is defined as SL - EST, and schedules them on the processor which results in the earliest start time, EST. The algorithm is extended to handle heterogeneous target architectures by associating to each task the average of executing the task on the different processors. Hence, SL is now the maximum sum of average execution times. Further, three values that account for processor variation, descendant consideration and a cost are associated with the task not being executed on the preferred processor.

One of the problems, with the execution model used by many of these algorithms, is the possibility to schedule two dependencies on a communication net in the same time slot, without increasing the total communication delay.

3. Problem Formulation

Our aim is to map an embedded systems behavior onto a given target architecture which may be heterogeneous.

3.1. Target Architecture

A heterogeneous target architecture is represented by a graph, $G_A = (V_A, E_A)$ in which each vertex describes an architecture component and the edges describe interconnections among the architecture components. Each architecture component may be a *processor*, p, or a *device*, d. A processor represents an active component, e.g. a CPU, an ASIP, or an ASIC. A device represents a passive component which may be controlled by a processor, e.g. an IO-device, a sensor, a display, or a memory. An edge, n_i, represents a *net* connecting two or more architecture components, i.e. a bus. Each net n_i is annotated with a *bandwidth*, b_i, which represents the amount of data which can be transferred over the net at any time. It is assumed that each processor has dedicated communication hardware so that computation can be overlapped with communication, i.e. a new task may be invoked on the processor while the results of the previous task are transferred to another processor. It is further assumed that there exists at least one net between any two processors in the architecture.

Example 1: Figure 1 shows a target architecture containing 3 processors and 3 devices. Throughout this paper, we will

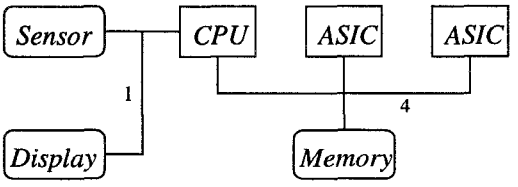


Figure 1. Target architecture with devices.

however, use a somewhat simpler example to illustrate our scheduling algorithm, i.e. the target architecture shown in figure 2. Note, that this architecture has no devices.

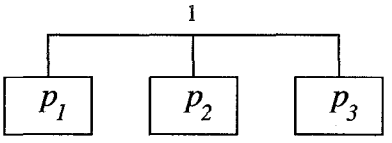


Figure 2. Simple target architecture.

□

3.2. System Specification

The behavior of an embedded system is described by a *task graph*, $G_T = (V_T, E_T)$, which is a partially-ordered set of *tasks* represented as a directed acyclic graph. Hence, each vertex, $\tau_i \in V_T$, in the task graph represents a task describing a single thread of execution which cannot be preempted, i.e. an atomic operation. An edge, $e_{i,j} \in E_T$, describes a data dependency between the tasks τ_i and τ_j. Each edge is annotated with the amount of data which have to be transferred between the two tasks, denoted as $d_{i,j}$. Without loss of generality [8], we assume that the data transferred on each edge emanating from a process are the same.

We assume that an estimate of the execution time of task τ_i on processor p_k, $t_e(\tau_i, p_k)$, is available at compile-time. Further, a task may require one or more devices for its execution, these devices will be allocated the processor during the complete execution of the task. If a task τ_j cannot be executed on a processor p_l, for instance because the required devices are not connected to the processor, the execution time is set to infinite.

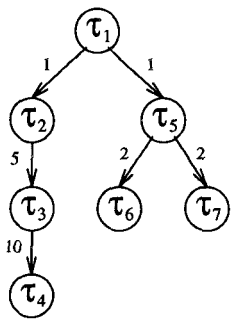
The communication time between two processors is defined as,

Definition 1 *Let τ_i and τ_j be two data dependent tasks scheduled on p_k and p_l respectively. The communication time is then given by,*

$$t_{c_{k,l}}(\tau_i, \tau_j, n_m) = \begin{cases} \lceil \frac{d_{i,j}}{b_m} \rceil & \text{if } k \neq l \\ 0 & \text{if } k = l \end{cases}$$

where b_m is the bandwidth of net n_m which connects p_k and p_l.

Example 2: Figure 3 shows a simple task graph and the processor execution times of each task when executed on the processors of figure 2.



	p_1	p_2	p_3
τ_1	2	1	3
τ_2	5	4	7
τ_3	1	4	5
τ_4	11	14	1
τ_5	4	4	4
τ_6	7	5	5
τ_7	5	5	5

Figure 3. Task graph and processor execution times.

□

4. The HCP Scheduling Algorithm

The scheduling problem when taking interprocessor communication into account contains two main aspects: assigning tasks to processors, and allocating nets for interprocessor communication. Our scheduling strategy addresses both problems concurrently, and is based on the classical list scheduling algorithm, e.g. [4]. In the following we present the Heterogeneous Critical Path (HCP) scheduling algorithm, as outlined below,

HCP$(G_T, G_A) \equiv$
{
  $\forall \tau_i \in V_T, \forall p_j \in V_A$: calculate $\texttt{cpl}(\tau_i, p_j)$;
  update L;
  while $L \neq \emptyset$ **do** {
    τ_i = `select_task(L)`;
    p_j = `select_processor(L)`;
    `schedule`(τ_i, p_j);
    update L; }
}

First a priority metric is calculated for each task-processor combination. A list L contains the tasks which are ready to execute, i.e. their predecessors have all been scheduled. A task is then selected based on the priority metric, and assigned to a the best processor when taking communication into account. Finally, the task is scheduled on the processor and the possible communication is scheduled on the appropriate net. In the following we will describe each step in the algorithm in more detail.

4.1. Task Selection

Tasks are selected from L according to their *critical path length* (CPL), i.e. the task with the largest CPL is selected.

The CPL of a task τ_i is the *shortest* path in the task graph from τ_i to any end-task, that is, a task with no successors. The path length is calculated as the sum of execution times of the tasks on the path and communication times of inter-task dependencies under the assumption that *all* tasks are scheduled on their *preferred* processors and *all* data dependencies are scheduled on their preferred nets. The preferred processors and nets are those resulting in the shortest critical path length.

Definition 2 *The critical path length of a task τ_i scheduled on a processor p_k is defined as:*

$$\texttt{cpl}(\tau_i, p_k) = t_e(\tau_i, p_k) + \min_{\tau_j \in V_{T,succ}, p_l \in V_A} (t_{c_{k,l}}(\tau_i, \tau_j, n_m) + \texttt{cpl}(\tau_j, p_l))$$

where $V_{T,succ}$ is the set of direct successor tasks of τ_i.

Example 3: Using the definition of CPL, we get the following values for the task graph of figure 3.

	p_1	p_2	p_3
τ_1	16	15	16
τ_2	16	15	13
τ_3	12	15	6
τ_4	11	14	1
τ_5	11	9	9
τ_6	7	5	5
τ_7	5	5	5

The critical path length of the end-tasks (τ_4, τ_6 and τ_7) are the same as their execution times. As an example of calculating CPL, consider the calculation of $\texttt{cpl}(\tau_3, p_1)$:

$$\texttt{cpl}(\tau_3, p_1) = 1 + \min(0 + 11, 10 + 14, 10 + 1) = \mathbf{12}$$

□

From the list of ready tasks L, the task with the *longest* minimum CPL is selected, i.e. this task is the most critical in order to obtain a short overall scheduling length. I.e. the selected task τ_i is found as,

$$\tau_i = \max_{\tau_j \in L}(\min_{p_k \in V_A}(\texttt{cpl}(\tau_j, p_k)))$$

Example 4: Consider our example from figure 3 and assume that τ_1 has been scheduled on processor p_2. The ready list contains two tasks, τ_2 and τ_5, and hence,

$$\max(\min(16, 15, 13), \min(11, 9, 9)) = \mathbf{13}$$

which means that task τ_2 is selected.

□

4.2. Processor Selection

Having selected a task τ_i, we have to select the processor p_k on which to schedule it. Here we have to consider two aspects:

- The *earliest start time* (EST) at which τ_i can start its execution on p_k when taking into account the communication from predecessors of τ_i and their schedule on the available nets. This term we will denote $\texttt{est}(\tau_i, p_k)$.
- The possible influence on the overall schedule length by scheduling τ_i on p_k. This is expressed as the critical path length, $\texttt{cpl}(\tau_i, p_k)$.

Hence, the processor p_k on which to schedule τ_i is the one which satisfy the expression:

$$\min_{p_k \in V_A} (\texttt{est}(\tau_i, p_k) + \texttt{cpl}(\tau_i, p_k)) \quad (1)$$

The actual scheduling of a task on a processor is defined as follows:

Definition 3 *Let p_k be a processor on which to schedule a task τ_i, and let $\langle \tau_1, \tau_2, \ldots, \tau_n \rangle$ denote the sequence of already scheduled tasks on p_k. The schedule $\texttt{s}(\tau_i, p_k)$ is defined as the earliest possible time t at which the following equations are fulfilled:*

$$t \geq t_{end}(\tau_j, p_k) \geq \texttt{est}(\tau_i, p_k)$$
$$t \leq t_{end}(\tau_{j+1}, p_k) - t_e(\tau_i, p_k)$$

The first equation ensures that all data is ready before executing τ_i. The second equation states that there should be time for executing τ_i before the next task, τ_{j+1} in the schedule starts its execution. If no such time slot can be found, τ_i is scheduled after the last task, i.e. τ_n.

The scheduling, $\texttt{s}(e_{i,j}, n_m)$, of data dependencies on nets is defined in the same way.

Let $t_{end}(\tau_i, p_k)$ indicate the actual end-times of task τ_i when scheduled on processor p_k. Let $V_{T,pred}(\tau_i)$ denote the set of predecessor tasks of τ_i, and $E_{T,pred}(\tau_i)$ the set of data dependencies $e_{j,i}$, where $\tau_j \in V_{T,pred}(\tau_i)$. Now, for all $\tau_j \in V_{T,pred}(\tau_i)$, we select the one with the earliest end-time, i.e.

$$\min_{\tau_j \in V_{T,pred}(\tau_i), p_l \in V_A} (t_{end}(\tau_j, p_l))$$

and try to schedule the data dependency $e_{j,i}$ on all possible nets n_m, that is, nets which connect p_k and p_l. We select the net resulting in the earliest completion time of the data transfer, let $t_{end}(e_{j,i}, n_m)$ denote this time. We continue this process until all data dependencies $E_{T,pred}(\tau_i)$ have been scheduled. Then,

$$\texttt{est}(\tau_i, p_k) = \max_{e_{j,i} \in V_{T,pred}(\tau_i)} (t_{end}(e_{j,i}, n_m))$$

From equation 1 we can now select the processor on which to schedule τ_i, and we perform the scheduling of τ_i and $E_{T,pred}(\tau_i)$ according to definition 3.

If two tasks, τ_2 and τ_3, dependent on the same task τ_1, and τ_3 is to be scheduled on the same processor, p_l, as τ_2, the algorithm takes into account that the data transfer has already been allocated, i.e. $t_{c_{k,l}}(\tau_1, \tau_3, n_m) = 0$. However, the task cannot start execution before data is available.

Example 5: Consider our previous example where task τ_2 was selected. The processor on which to schedule τ_2 is determined as,

$$\min(\max(1+1)+16, \max(1+0)+15, \max(1+1)+13)$$
$$= \mathbf{15}$$

which means processor p_3. Note, that this is actually the processor on which τ_2 has its worst execution time. Figure 4 shows the complete schedule of the task graph of figure 3.

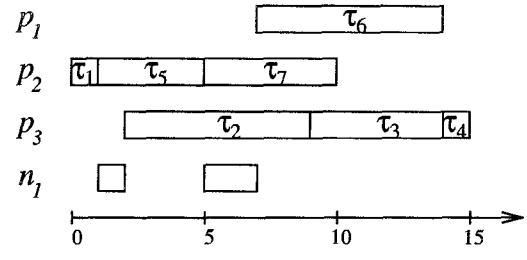


Figure 4. Scheduled task graph.

□

5. Experimental Results

The HCP algorithm has been implemented in C++ along with the DLS [5], ETF [2], and ETF2 algorithms. The DLS algorithm is implemented for heterogeneous target architectures using the generalized dynamic level GDL1, which accounts for processor variation and resource scarcity, see [5]. The GDL1 is exhaustively examined for all the ready tasks. The ETF algorithm is implemented using SL as the task priority, while ETF2 uses CPL as the task priority. ETF2 selects the task-processor pair based on the earliest end-time of the task, i.e. it includes the task execution time when choosing a task.

Table 1 shows the resulting schedule lengths when applied to some benchmarks from the literature. A_1 is the target architecture of figure 2 while A_2 is a two processor architecture.

We have implemented a task graph generator, `taskgen`(s_t, s_d, t_e, d), somewhat like the ones described in [7] and [3]. s_t indicates the number of tasks, s_d the number of data dependencies, t_e the maximum execution time, and d

example	target	HCP	DLS	ETF	ETF2
fig.3	A_1	15	20	19	20
fig.1 in [5]	A_1	30	25	26	28
fig.2 in [5]	A_2	32	29	28	29
gausseli[3]	A_1	490	520	480	560

Table 1. Schedule lengths for some benchmarks.

the maximum size of data dependencies. For each task-processor pair an execution time is randomly selected from the uniformly distributed interval $[1; t_e]$. The size of data produced by each task is randomly taken from $[1; d]$. Data dependencies are generated by randomly choosing two tasks with no dependency.

For our experiments, we have generated and scheduled 100 task graphs for each set of parameters. The resulting schedule length is taken as the average over these 100 task graphs. In all experiments, s_d was set to twice the number of tasks, and the target architecture was the one from figure 2.

Table 2 shows the results of the four algorithms when scheduled onto a homogeneous target architecture, while table 3 shows the results on a heterogeneous target architecture. In both cases the HCP algorithm produces in average the best schedules, though there are cases where one of the other algorithms is best, as indicated in table 1. All schedules were produced in less than one second on a SUN SPARC workstation.

s_t	t_e, d	HCP	DLS	ETF	ETF2
10	10, 10	39	39	38	40
20	10, 10	68	70	68	71
50	10, 10	148	149	153	151
100	10, 10	285	294	299	294
200	10, 10	559	573	585	573

Table 2. Average schedule lengths for synthetic benchmarks on a homogeneous architecture.

6. Conclusion

We have presented a new list based scheduling algorithm which concurrently assigns tasks to processors and allocates nets to interprocessor communication. The main advantage over previous approaches is that we take communication into account while selecting tasks and processors, and that we do not allow multiple communications to take place simultaneously over the same net.

Experiments have shown that our algorithm compares well to previous approaches. However, all of the compared algorithms, including ours, fail to favorize the processor on which predecessors of the successors of a task have been scheduled.

s_t	t, d	HCP	DLS	ETF	ETF2
10	10, 10	37	39	42	40
	10, 40	94	96	136	100
	40, 10	53	59	65	57
20	10, 10	61	70	68	69
	10, 40	126	131	172	147
	40, 10	129	149	157	143
50	10, 10	145	151	154	150
	10, 40	248	255	350	271
	40, 10	401	409	422	417
100	10, 10	278	297	297	296
	10, 40	468	465	692	486
	40, 10	867	898	914	870
200	10, 10	563	581	596	589
	10, 40	919	891	1376	921
	40, 10	1808	1837	1865	1802

Table 3. Average schedule lengths for synthetic benchmarks on a heterogeneous architecture.

7. Acknowledgements

This research has partly been sponsored by the Danish Technical Research Council under the "Codesign" program.

References

[1] T. Adam, K. Chandy, and J. Dickson. A comparison of list scheduling for parallel processing systems. *Comm. ACM*, 17(12):685–690, December 1974.

[2] J. Hwang, Y. Chow, F. Anger, and C. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Computing*, 18(2):244–257, April 1989.

[3] Y. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 7(5):506–521, May 1996.

[4] G. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., Princeton Road, S-1, 1994.

[5] G. Sih and E. Lee. A compile-time scheduling heuristic for itercennection-constrained heterogeneous processor architectures. *IEEE Trans. Parallel Distrib. Syst.*, 4(2):175–187, February 1993.

[6] M. Wu and D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. Parallel Distrib. Syst.*, 1(3):330–343, July 1990.

[7] T. Yang and A. Gerasoulis. Dsc: Scheduling parallel tasks on an unbound number of processors. *IEEE Trans. Parallel Distrib. Syst.*, 5(9):951–967, Sebtember 1994.

[8] T.-Y. Yen. *Hardware-Software Co-Synthesis of Distributed Embedded Systems*. PhD thesis, Princeton University, 1996.