Synthesis of Hard Real-Time Application Specific Systems

Chunho Lee†, Miodrag Potkonjak† and Wayne Wolf‡

†Computer Science Dept., University of California, Los Angeles, CA
‡Dept. of Electrical Engineering, Princeton University, Princeton, NJ

E-mail: leec@cs.ucla.edu, miodrag@cs.ucla.edu, wolf@princeton.edu

Abstract

This paper presents a system level approach for the synthesis of hard real-time multitask application specific systems. The algorithm takes into account task precedence constraints among multiple hard real-time tasks and targets a multiprocessor system consisting of a set of heterogeneous off-the-shelf processors. The optimization goal is to select a minimal cost multi-subset of processors while satisfying all the required timing and precedence constraints. There are three design phases: resource allocation, assignment, and scheduling. Since the resource allocation is a search for a minimal cost multi-subset of processors, we adopted an A search based technique for the first synthesis phase. A variation of the force-directed optimization technique is used to assign a task to an allocated processor. The final scheduling of a hard-real time task is done by the task level scheduler which is based on Earliest Deadline First (EDF) scheduling policy. Our task level scheduler incorporates force-directed scheduling methodology to address the situations where EDF is not optimal. The experimental results on a variety of examples show that the approach is highly effective and efficient.*

1. Introduction

1.1. Motivation

Real-time systems can be defined as those systems in which the proper functioning of system implementations depends not only on the logical correctness of the computation, but also on the time at which the results are produced [37]. Classical examples include automobile and aircraft monitoring and control systems, a variety of command and control systems, and process control systems. Table 1 shows a set of avionics tasks which run on an avionics application specific computer [1] [38]. In [1] three different implementation solutions were derived. The number of processors used in the solutions ranges from six to eight.

Over the past few years demand for real-time systems in broadband and wireless communication, multimedia, video-on-demand, interactive television, and remote sensor security control has been growing at a remarkably high rate. Moreover, modern real-time systems including multimedia and video game systems are often multi-task multiprocessor application specific systems [3] [4] [18] [51].

Targeting a multiprocessor system for multitask application specific system is often the best or even only implementation option. When an application have tasks that exhibit very different characteristics, executing them on the same processor inevitably leads to inefficient implementation. For example, suppose that an application includes a 64-bit block cipher task such as blowfish algorithm by Schneier [41] and an 13-bit speech coding task such as GSM [1] [10]. Trace-driven simulations on SPARC using SHADE [7] reveal that 92% of the ALU operations of the 64-bit block cipher are 32-bit wide and 94% of the ALU operations of the 13-bit speech coding 16-bit wide. Executing the latter on the wide datapath required by the former task is very inefficient. Less powerful, thus less expensive, processor for the 64-bit block cipher task in conjunction with another processor for the 13-bit task can be used as long as it can execute the 64-bit block cipher task in timely fashion when the task is exclusively assigned to it. Often it is infeasible to use a uniprocessor system for the realization of all tasks due to the intrinsic timing constraints. For example, if the sum of execution times of the individual task is longer than the total available time, the only implementation option is multiprocessor platform.

Both traditional behavioral synthesis and emerging system synthesis, however, have been focused on synthesis of single task applications [11] [27] [49]. Recently, Potkonjak and Wolf developed an algorithm for the synthesis of multitask applications at the behavioral level [34]. A few research groups addressed synthesis of hard real-time systems [35] [52], but usually under restricted design scenarios.

This research project has been motivated for developing an approach to system level synthesis of hard real-time multitask multiprocessor systems. In fact, a number of semiconductor companies have been restructuring their system synthesis tools exactly for the new synthesis scenario (e.g., multiple tasks partitioned on several proces-

[1] European wireless communication speech coding standard

Task No	Task Description	Iterations per second	Instructions per second
1.	Attitude Control	20	1,228
2.	Flutter Control	250	138
3.	Gust Control	240	58
4.	Autoland	160	342
5.	Autopilot	5	200
6.	Attitude Detector	30	2,560
7.	Inertial Navigation	25	1,350
8.	VOR/DME	5	770
9.	Omega	5	800
10.	Air Data	5	200
11.	Signal Processing	0.2	1,750
12.	Flight Data	5	5,520
13.	Airspeed	16	549
14.	Graphics Display	8	3,975
15.	Text Display	10	1,900
16.	Collision Avoidance	670	32
17.	Onboard Communication	250	28
18.	Offboard Communication	4	155
19.	Data Integration	4	360
20.	Instrumentation	5	2,792
21.	System Management	0.5	2,320
22.	Life Support	0.5	2,320
23.	Engine Control	33	3,597
24.	Executive	5	200

Table 1. An example of multitask application specific system: avionics tasks characteristics (adopted from [1, 38])

sor cores). For instance, Rockwell Semiconductor has been developing synthesis infrastructure which combines several general purpose processors (ARMs) with in-house programmable DSP components [15].

1.2. Overview of the New Approach

Traditionally, scheduling has been considered the key optimization step in behavioral synthesis. For example, Gajski's statement that scheduling is the single most important behavioral synthesis step [27] has been widely quoted. Recently, however, it was realized that the scheduling in the traditional behavioral synthesis usually does not have high impact on the quality of the final implementation [49] and that other synthesis optimization tasks, such as transformations, usually make greater differences in the final results [5] [16] [22] [32] [33]. Moreover, it was reported that the available scheduling algorithms often produce optimal results [6] [36] [39]. The focus of behavioral synthesis research, therefore, shifted from scheduling to other issues of synthesis that were found to be more beneficial if addressed.

From the system level synthesis point of view, however, higher level optimizations such as task scheduling, resource allocation and task assignment optimizations will have more significant impact on final synthesis results since, unlike lower level optimizations, they tend to have broader influences throughout a design. In that regard, task level scheduling will provide numerous new avenues for both academic research and commercial product development.

The system level synthesis approach presented in this paper is consistent with the new development in CAD research community. We focus on higher level optimizations for synthesis of hard real-time multitask application specific systems based on a target system model consisting of off-the-shelf components.

We divided the synthesis task into three subtasks: resource allocation, assignment, and scheduling. Since the resource allocation is a search for a minimal cost multi-subset of processors, we adopted an A* search based technique for the first synthesis phase. A variation of the force-directed optimization technique is used to assign a task to an allocated processor. The final scheduling of a hard-real time task is done by the task level scheduler which is based on Earliest Deadline First (EDF) scheduling policy. Our task level scheduler incorporates force-directed scheduling methodology to address the situations where EDF is not optimal.

We developed a set of modular, flexible, and reusable tools for system level synthesis of hard real-time multitask multiprocessor application specific systems. Each individual synthesis technique can be used to supplement the existing components of system level synthesis tools, thanks to the modularity of our approach.

In summary, this is the first paper, to the best of our knowledge, which presents a fully modular approach for synthesis of hard real-time multitask multiprocessor application specific systems based on a target system model consisting of a set of heterogeneous off-the-shelf processors. We established computational complexity of all individual synthesis steps. This is also the first published work that explores the possibility of resource allocation optimization using the A* search technique. We have developed an EDF-based task scheduler that works well on non-preemptive tasks with arbitrary release times.

1.3. Paper Organization

The next section provides related research efforts along with their key ideas and results. In Section 3, relevant background materials are discussed. We formulate three synthesis subproblems and establish their computational complexity in Section 4. In Section 5, we describe the global design flow of our approach. We elaborate on the technical details of resource allocation, assignment, and scheduling algorithms in Sections 6, 7, and 8, respectively. We present and analyze the experimental results in Section 9. Finally, the last section summarizes the conclusion.

2. Related Works

Many classical results in behavioral and system level synthesis, hard real-time scheduling, and search and heuristic optimization techniques are directly related to this research.

A good introduction and review of the early work on behavioral synthesis are given in several papers and books [11] [27]. System level synthesis, including hardware- software codesign, also is a premier design and CAD research topic [50]. The most relevant system synthesis research subdomain is hardware-software partitioning, where a great variety of techniques have been proposed [2] [12] [13] [17] [21].

Hard-real time scheduling has been an important topic of research for three decades. The early works on scheduling of a set of periodic tasks with strict timing constraints on periodicity, arrival and required time of each task, were culminated in a classic rate-monotonic scheduling algorithm [25]. Consequently, rate-monotonic scheduling has been extensively analyzed and generalized, mainly by researchers at Carnegie-Mellon University [24] [43] [45]. The most notable practical application of real-time scheduling approaches, in particular rate-monotonic and generalized rate-monotonic scheduling algorithms, include the inclusion of rate-monotonic scheduling as the scheduling policy for the IEEE POSIX real-time operating system standard and IEEE Futurebus+ standards [19] [20] [44], and use of the generalized rate-monotonic scheduling techniques in several major advanced technology projects such as the Space Station Program and the European Space Agency's on-board operating system [45]. The strong endorsements from several research and development groups of the earliest-deadline-first and rate-monotonic scheduling as most suitable resource allocation policies for continuous media servers [28] [37] [48] and ATM switch scheduling [42] further stress importance of this hard-real time scheduling approach.

A* search and force directed heuristics are often used as optimization mechanisms for computationally intractable problems [40]. It has been proved that it is the best informed search strategy in the sense that any other search strategy which also guarantees optimality, requires at least as much run time as A* search does. Force-directed heuristics have been widely used at the several levels of abstraction in the design process. The approach was pioneered by Soukup [46], who used it as a core optimization routine in the epitaxial growth algorithm for IC and board constructive placement. Paulin and Knight [31] developed a force-directed approach for data-flow graph scheduling, which due to its clear intuitive foundations and strong performances have been used by many behavioral synthesis schedulers [27] [49].

3. Preliminaries

This section reviews the definitions of terms used in the following presentation. Also in this section we describe assumptions and the formulation of our design abstractions and models. Task model and communication model are explained in detail. As we use the force-directed optimization quite intensively, we provide a highlight of it at the end of this section.

3.1. Definitions

As in behavioral level synthesis, we solve scheduling, allocation, and assignment (partitioning) problems to synthesize an implementation. Resource allocation refers to selection of processors from a pool of available processors. Assignment or partitioning means, given a set of allocated processors, each task in a task set is assigned to exactly one of the allocated processors. Task scheduling refers to generation of feasible schedules for the assigned tasks on the allocated processors.

Start time (or release time) refers to the earliest time when all required data for an iteration of a task are available. Finish time (or deadline) means the latest time by which a task has to be completed, i.e., output data for a given iteration should be available to its user.

As in the force-directed scheduling [31], we use the notion of time frame extensively. Time frame is defined as the interval between the earliest possible start time and latest acceptable finish time of a task.

3.2. Task Model

Our basic model for a hard real-time task follows the definitions used in real-time systems research. We assume that all tasks are defined on semi-infinite or very long streams of data. A task consumes one set of input and produces one set of output at each iteration. All tasks are periodic. For each task, three timing constraints are associated with it: period, start time and finish time [47]. Each task execution time or upper bound on the execution time on each of the available processors is assumed to be known through profiling and other techniques [7] [26] [29]. An iteration of a task refers to an occurrence of execution of the task. Within the least common multiple of periods of a set of given tasks (*LCM*), one or more iterations of a task can occur. For example, if *LCM* is 30 and the period of a task is 5, there are 6 iterations of the task within the *LCM*. All the iterations of a task have the identical periodic timing constraints. Note that, given a set of tasks, the same task occurrence pattern repeats itself every *LCM*. This means that a feasible schedule for the first *LCM* can be repeatedly used for the following timing blocks of size *LCM* [37].

3.3. Communication Model for Tasks with Precedence Constraints

Since a synthesis problem with no precedence constraints is a special case of more general problems (i.e., problems with precedence constraints), in the following sections, we discuss the special case first and then generalize the solutions.

For the general case, the communication cost is assumed to be uniform between any pair of processors and any pair of tasks. The networking cost (hardware cost) is proportional to the number of processors allocated. We believe that this is a good approximation which abstracts implementation details of communication subsystem so that our focus on resource allocation, assignment, and scheduling is maintained. In case that a communication subsystem cannot guarantee the uniform communication cost assumption, we can modify time frames of tasks that have precedence constraints in a manner similar to [23].

Each allocated processor has a sufficient amount of buffer to hold incoming data for pending tasks. In the worst case, we need $m-1$ input buffers at each processor where m is the number of allocated processors since due to timing constraints, a system should be designed in such a way that a task is not executed before its successor consumes its output. Each buffer is assumed to be large enough to store the largest incoming data item.

Since we use generalized force-directed heuristics for the partitioning and scheduling, combined with the EDF based scheduling, the algorithms developed without considering task precedence constraints are already general enough to handle cases where task precedences are imposed. The only modification made to the algorithm to handle problems with dependency constraints is explained in Section 5.

3.4. Implementation Constraints

Two implementation constraints are imposed. First, no preemption is allowed. Introduction of preemption to a scheduling problem greatly simplifies many synthesis problems [47]. While a scheduling problem can be an optimization problem of polynomial time complexity when preemption is assumed, it might be a NP-complete problem when no preemption is allowed [47]. However, considering high context switching costs for modern computing systems [8], preemptive scheduling can be prohibitively expensive for some processors. Note that once the switching cost is taken into account, many optimization problems reestablish their computational intractability.

The second restriction is that all the iterations of a task should be executed on the same processor and no further division of a task is possible (no parallelism in a task).

3.5. A Highlight of the Force-Directed Optimization

The force-directed optimization is a global optimization technique. It is reported that any global optimization requires at least the complexity of the force-directed heuristics. The optimization algorithm consists of three major steps: determination of time frames, creation of distribution graphs, and calculation of self forces.

Distribution graphs are created by taking summations of the execution probability of each task for each time slot. The execution probability is computed by dividing the execution time of a task by its time frame. Force is defined by the product of the distribution graph and change in distribution of a task when the time frame of a task is changed by scheduling or assigning a task to a hardware resource. When distribution is reduced (negative change), the force is negative; when distribution is increased (positive change), the force is positive. Positive force implies more crowded time frames and negative force the opposite. Self-force is the sum of changes in force when the time frame of a task is changed. Self-force captures the overall impact to scheduling or assignment of a task set when the time frame of a task is changed. By computing and comparing self-force values for each task in a task set, it is possible to schedule or assign a task at a time.

4. Problem Formulation and Complexity

In this section we formulate the optimization problems associated with three system-level synthesis steps and establish their computational complexity. The targeted synthesis problems can be defined in formal terms using the standard Garey-Johnson [14] format.

Problem: Processor Allocation

Instance: A set of l processors and a set of n periodic hard real-time tasks are given. Each processor has an associated cost (price). The execution time of a task when implemented on an available processor is known. The relative periodic release and deadline constraints for each iteration are given.

Question: Is there a multisubset of processors (subset in which some processors can be include more than once) such that each task is assigned and scheduled on exactly one processor and that the total cost of the selected processors is at most K?

Problem: Task Assignment

Instance: A set of m processors and a set of n periodic hard real-time tasks are given. The execution time of a task when implemented on an available processor is known. The relative periodic release and deadline constraints for each iteration are given.

Question: Is there an assignment of each task to exactly one of the processors such that all tasks can be scheduled within their timing constraints?

Problem: Task Scheduling

Instance: A set of n_i periodic hard real-time tasks and a processor i are given. The execution time of a task when implemented on the processor i is known. The relative periodic release and deadline timing constraints for each iteration are given.

Question: Is there a feasible non-preemptive schedule for the task set such that all the timing constraints for all the tasks are satisfied?

```
Synthesis ( ) {
   while ( no feasible schedule ) {
      Allocation();
      Assignment();
      Scheduling();
   }
}
```

Figure 1. The system-level synthesis of hard real-time application specific systems

We have proved that the Processor Allocation problem is NP-complete by transforming the equal subset problem [14] into a restricted instance of the problem using Karp's polynomial transformation techniques. We have also proved using the same technique that the Task Assignment is NP-complete. As a starting point in the proof we used the knapsack problem [14]. Finally, the Task Scheduling problem with non-preemptive scheduling policy and arbitrary release times is also NP-complete, as proven previously in [47]. We say that a task j has release time r_j if its execution cannot start before time r_j.

5. Synthesis Approach

As we described in the previous sections, the synthesis problem of minimal cost implementation of a set of real-time tasks can be viewed as several layered NP-complete problems. This observation is the direct result of our recognition that many optimization techniques are available as convenient starting points for individual optimization problems although the interaction between them is very difficult. To take advantage of this observation, we divided the synthesis problem into three distinctive subproblems. The division of the problem into subproblems enables us to naturally enforce modularity, flexibility and reusability of algorithms and software. The basic premise of the synthesis approach is that by developing effective and efficient heuristics to deal with each subproblem and combining them in a reasonable manner, local optima can be escaped while preserving the advantages of solving a relatively small subproblem at a time. The overall synthesis flow is given by the pseudo code in Figure 1.

The allocation subtask selects a set of processors in such a way that the cost of selected processors is minimized and the task set can be assigned and scheduled on the selected set of processors. A fast estimation algorithm is used to check the feasibility of the tentative solution produced by the allocation algorithm. The assignment procedure assigns tasks to allocated processors. Finally, the scheduling procedure generates a feasible schedule for an allocated processor if there exists one. If there is no feasible schedule for the set of allocated processors, the synthesis process repeats with the next best allocation until a set of feasible schedules is obtained.

The allocation subtask finds a set of resources by searching the solution space using the A* search strategy [9] [40]. A solution can be represented by a path in a solution tree. The root node of the solution tree represents the empty initial solution. At each step (level) of the search, one out of k branches is chosen, where k is the number

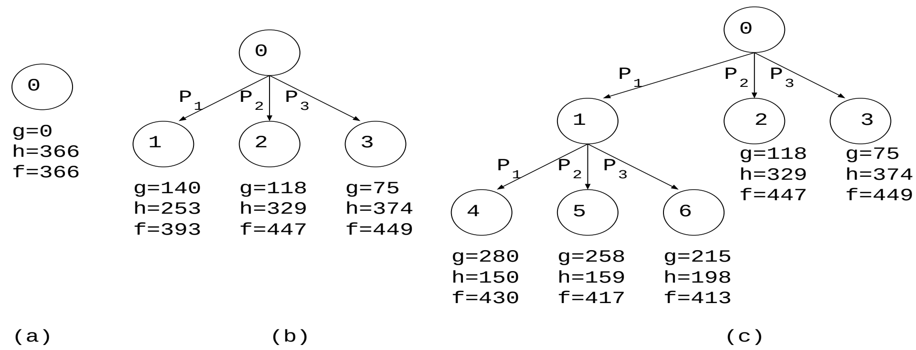


Figure 2. An example of A* search tree: (a) the initial search tree with an empty initial solution (b) A* search tree after one expansion (c) after two expansions

of different processor choices. In the A* search, the search follows the most promising direction. This decision is made by an optimistic estimate of the cost of a complete solution, given by

$$f = g + h, \tag{1}$$

where g is the total cost of current partial solution and h refers to an optimistic estimation of additional cost to complete the solution. For example, Figure 2(a) shows an initial A* search tree. The search trees after first and second expansions with three possible choices are given in Figure 2(b) and (c), respectively. Each choice is associated with a cost (g) and an estimated additional cost (h). Initially, f is consisted only of a pure estimate of h since the initial cost of the empty solution is 0. At any given time the search tree is expanded at the node with the minimum f.

The A* heuristic function h which guides the search is based on relaxed assignment and scheduling. The heuristic first performs a relaxed assignment. Based on the relaxed assignment, it identifies a subset of tasks that can be scheduled. Finally, it computes the estimated additional cost to take care of the tasks that cannot be scheduled on the proposed set of processors. Once the allocation procedure reaches a tentatively feasible solution it hands the allocation result over to the partitioning (assignment) procedure.

The assignment procedure assigns a task at a time to an allocated processor. The assignment heuristic is based on the force-directed scheduling [31]. To characterize a set of non-preemptive periodic hard real-time tasks with arbitrary release times and deadlines, we need to look at all the individual instances of the tasks. The density distribution technique from the force-directed approach provides a good means to measure the effect of assigning a task to a processor. It is constructed in such a way that it takes into account the distribution of demands of computing resources and the running times of tasks on each allocated processor.

Our task-level scheduler is based on Earliest Deadline First (EDF) scheduling. EDF is optimal scheduling approach in many cases when preemption is allowed [47]. It is also optimal in several cases of non-preemptive scheduling with arbitrary release times and deadlines [47]. The scheduler adopts heuristics to address the situations where EDF is not optimal. We developed a modified EDF, which utilizes the force-directed scheduling technique. Experimental results indicate that the modified EDF is very effective and efficient. The running time of the scheduling algorithm is nearly proportional to t0he number of instances of tasks, which is the same complexity of EDF.

When there are precedence constraints, we adjust time frames of tasks in a fashion that precedence constraints are transformed to timing constraints. Since a task dependent on another task should wait until the preceding task finishes, the earliest possible start time of the task must be later than the finish time of the preceding task. The timing constraints adjusted to meet precedence constraints are used to compute the time frames for the tasks. Since we use the EDF based scheduler, this preprocessing is sufficient to preserve precedence requirements. This is the direct application of our observation that we can express the dependency constraints in the form of timing constraints as we use generalized force-directed heuristics for the partitioning and scheduling, combined with the EDF based scheduler. The new start (release) and finish (deadline) of a task t_a, $a = 1, 2, ..., n$, is given by

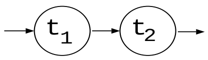
$$S_{p_y}(t_a) = \begin{cases} R(t_a), & \text{if } a = 1 \\ \max\{R(t_a), \min_{p_i \in P}\{S_{p_i}(t_{a-1}) + E_{p_i}(t_{a-1}) + C_{iy}\}\}, & \text{otherwise} \end{cases}$$

and

$$F_{p_y}(t_a) = \begin{cases} D(t_a), & \text{if } a = n \\ \min\{D(t_a), \max_{p_i \in P}\{F_{p_i}(t_{a+1}) - E_{p_i}(t_{a+1}) - C_{iy}\}\}, & \text{otherwise.} \end{cases}$$

The terms $S_{p_i}(t_a)$, $E_{p_i}(t_a)$ and $F_{p_i}(t_a)$ refer to the start, execution and finish time of a task t_a on a processor p_i, respectively. C_{ij}, $R(t_a)$, $D(t_a)$ and P refer to the communication cost between processors p_i and p_j, the release time and deadline of a task t_a, and the available processor set, respectively.

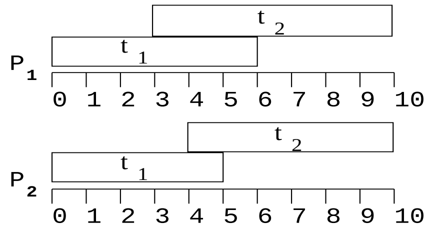
For example, consider that a task t_1 produces a data item for another task t_2 as given in Figure 3(a). The release time and deadline of both t_1 and t_2 are 0 and 10, respectively (Figure 3(b)). The task t_1 can start execution as soon as at time 0 whereas t_2 should wait until t_1 completes. In this example, the processor P_1 can execute t_1 and t_2 in 3 and 4 time units, respectively, as given in Figure 3(b). Assuming that the communication cost is a unit time, the earliest possible start time of t_2 on P_1 is given by $\max\{0, \min\{3, 5\}\} = 3$ since $S_{P_1}(t_1) + E_{P_1}(t_1) = 3$ and $S_{P_2}(t_1) + E_{P_2}(t_1) + C_{P_2P_1} = 5$. The latest acceptable finish time of t_1 on P_1 is $\min\{10, \max\{3, 6\}\}$ since $F_{P_1}(t_2) - E_{P_1}(t_2) = 6$ and $F_{P_2}(t_2) - E_{P_2}(t_2) - C_{P_2P_1} = 3$. Similarly, the earliest possible start time of t_2 on P_2 is given by $\max\{0, \min\{4, 5\}\} = 4$ and the latest acceptable finish time of t_1 on P_2 is $\min\{10, \max\{4, 5\}\} = 5$. This preprocessing ensures that no matter where each individual task is assigned, the precedence constraints are preserved. time 3 and time 6, respectively, in order to satisfy all the timing constraints. Hence, the time frames on the processor P_1 for t_1 and t_2 are $[0, 6]$ and $[3, 10]$, respectively. Since earliest possible finish time of t_1 is As for



(a)

	P_1	P_2	period	start	finish
t_1	3	5	10	0	10
t_2	4	6	10	0	10

(b)



(c)

Figure 3. An example of tasks with precedence constraints: (a) Task precedence graph (b) Task execution times and timing requirements (c) Adjusted time frames on each processors for the task set

time frames of tasks on P_2, Note that time frames on other processors are different due to communication costs involved (Figure 3(c)).

6. Resource Allocation

The goal of the allocation phase is to allocate a set of processors that will implement a set of tasks with minimal cost. The heuristic function used to guide the search is based on a relaxed partitioning and scheduling. The lower bound of the implementation cost of each task is the minimum among the products of the costs of processors and the corresponding utilization factor of a task. For example, given a set of tasks and processors in the first five columns of Table 2, the minimum implementation cost M_i of task i is computed by

$$M_i = \min_{j \in P} \{ \frac{C_j E_{ij}}{T_i} \}, \tag{2}$$

where C_j is the cost of processor j, E_{ij} the computation time of Task i on processor j, P the pool of available processors and T_i the period of task i. The result is given in the last five columns of Table 2. The last 5 columns under CP_j headings show prorated costs of task i implemented on processor j and they given by

$$CP_{ij} = \frac{C_j E_{ij}}{T_i}. \tag{3}$$

The sum of M_i's signifies the lower bound of the total implementation cost. That is, we have to spend at least 54.9 to implement all the tasks shown in Table 2. Those M_i's are goals which guide our search for the minimum cost implementation. These are the absolute minima because they do not reflect utilization factors of processors that is normally much less than 1 and it is assumed that a task will be assigned on a processor which offers the minimal cost implementation, which might not be the case for all the synthesis problems.

	P_1	P_2	P_3	P_4	T (period)	CP_1	CP_2	CP_3	CP_4	$Min\ (P)$
t_1	7	5	3	6	5	-	20	18	-	18 (P_3)
t_2	8	5	4	9	6	-	16.7	20	-	16.7 (P_2)
t_3	7	4	2	5	10	10.5	8	6	5	5 (P_4)
t_4	5	7	3	8	10	7.5	14	9	8	7.5 (P_1)
t_5	9	8	5	7	15	9	10.7	10	4.7	4.7 (P_4)
t_6	10	9	4	9	30	5	6	4	3	3 (P_4)
Cost	15	20	30	10					Sum	54.9

Table 2. A synthesis problem is shown at the left side of the table (the first 5 columns). Prorated implementation cost of each task on each processor is also given at the right side of the table (the last 5 columns): P_j, t_i and CP_j denote available processors, tasks and implementation cost of a task on Processor j, respectively. "-" indicates Task i cannot be implemented on Processor j.

	P_1	P_2	T	S	D	A
t_1	3	2	5	1	3	3
t_2	2	3	5	1	3	3
t_3	3	5	10	3	7	5
Cost	16	10				

Table 3. An instance of synthesis problem: T-period, S-start time, D-deadline, A-available time

The heuristic function should be computationally efficient while providing accurate prediction of the effect of allocating resources. In each allocation step, before a processor is chosen to be allocated, the procedure estimates how well the set of allocated processors will be utilized and how much more processors should be added in the following allocation steps if the processor being examined is chosen. These estimates are obtained through the relaxed partitioning and scheduling.

Note that our original problem has a set of timing constraints, the atomic execution constraint of an instance of a task (i.e., an instance of a task cannot be distributed to more than one processor), and the non-preemptive scheduling constraint. We relax the atomicity restrictions and perform partitioning. Each instance of a task is divided into several pieces based on the number of allocated processors and its execution time on each processor. For example, consider the problem given in Table 3. Assuming that P_1 and P_2 are allocated, the relaxed partitioning probability table is shown in the first two columns of Table 4. The last two columns of Table 4 show the claimed portions of processors by respective tasks based on the probability. The partitioning probability table indicates how big a chunk of a task is assigned to a processor. Thus, the numbers in any of the last two columns of Table 4 are estimated execution times of corresponding tasks on the "super processor" which combines all the computing capacities of the allocated processors.

To devise heuristics for resource allocation, we observe that the utilization factor reveals an upper bound for

	P_1	P_2	P_1 - assigned	P_2 - assigned
t_1	$2/5$	$3/5$	$2/5 \times 3$	$3/5 \times 2$
t_2	$3/5$	$2/5$	$3/5 \times 2$	$2/5 \times 3$
t_3	$5/8$	$3/8$	$5/8 \times 3$	$3/8 \times 5$

Table 4. Relaxed task assignment example: The probabilities on which the relaxed partitioning is based are given for the two allocated processors in the first two columns. The numbers shown in the last two columns are execution times of tasks on the ''super processor.''

preemptive schedulability [25] and that the non-preemptive scheduling is more difficult (in both terms of checking the schedulability and generating a feasible schedule) [47]. We can incorporate them in our heuristic function as a means to estimate if a set of allocated processors can be a feasible solution. For example, the combined utilization of the processor set of P_1 and P_2 (the utilization factor of a "super processor") given in Table 4 is $(2/5) \times (3/5) + (3/5) \times (2/5) + (5/8) \times (3/10) = 0.67$. The utilization factor provides a good measure whether a feasible schedule can be found or not.

It is also important to note that when the utilization is too high, which means that there is high probability of not being able to generate a feasible schedule [34], we need to know what portion of the task sets can be scheduled on the allocated processors to proceed with the search. We perform a relaxed scheduling on the partitioned task set using our force-directed delay-based EDF which will be described in Section 8. When no feasible schedule can be found, the first task that cannot be scheduled is eliminated and the relaxed scheduling continues until it finds set of tasks that can be scheduled. As described in Section 8, our scheduling is based on the EDF which offers the optimal length schedule if it finds a feasible schedule. Those facts enables us to assume that by eliminating tasks that cannot be scheduled, we do not run into risk of eliminating a task that would be scheduled in more efficient manner. With the set of tasks that is scheduled, the utilization is computed. By combining the current total cost and the estimate of the future cost that is required for the tasks that were eliminated from the relaxed scheduling process, we get an estimate of the overall implementation cost. The estimate C is given by

$$C = \sum_{j \in I} C_j + \sum_{i \in R} M_i, \tag{4}$$

where C_j refers to the cost of Processor j, I the set of allocated processors, M_i the minimum implementation cost of task i as given by Equation (2) and R the set of remaining tasks.

When a set of dependency constraints are imposed, we adjust time frames as described in the previous section (Section 5) before the relaxed assignment and scheduling are performed. Note that since we use the notion of "super processor" for the relaxed partitioning and scheduling, we only need to compute time frames of task for the "super processor."

```
Allocation () {
   MIN = a large number;
   do {
      for all available processors {
         (Residual, Processor, U) = Relaxed_Partitioning_And_Scheduling();
         if (Residual < MIN) {
            minProcessor = Processor;
            MIN = Residual;
         }
      }
      include the saved processor to the allocated processor set;
   } while ( ( Residual != 0 ) && ( U >= some threshold ) )
}
```

Figure 4. The resource allocation algorithm (U refers to the utilization factor)

The procedure selects a processor at a time. It computes estimations for all available processors at each allocation step. Of the estimated costs, the minimum which promises the best final solution is chosen. When the future cost component of the minimum estimate is non-positive, the actual partitioning of the task set to the set of allocated processors is performed. The allocation algorithm is summarized in Figure 4.

7. Assignment

Our assignment procedure is based on the observation that we have the best chance of generating a feasible schedule if we assign a task to one of the allocated processors in such a way that the distribution graphs (refer to Section 8 for details) on all the allocated processors are as even as possible and the differences among utilization factors of all the processors are as little as possible. We modify force-directed scheduling [31] to find a good assignment given a task set. First, we define the initial assignment probability P_{ij} of each task based on execution times of a task on each processors with which a task is tentatively assigned to processors as follows:

$$P_{ij} = \frac{E_{ij}^{-2}}{\sum_{i \in Q} E_{ij}^{-2}}. \tag{5}$$

The term Q refers to the given task set for a synthesis problem. This formula embodies the heuristic that the assignment of a task to a processor on which it requires a shorter execution time is more likely to give better results. Next, we use a modified force-directed assignment procedure to balance the loads among the processors and to make distribution graphs across all the processors be as even as possible. Based on the probabilities given by Equation (5), we compute the distribution D_{tj} of a task at time t on processor j by

	P_1	P_2	T	S	D	A
t_1	3	2	5	1	3	3
t_2	2	3	5	1	3	3
t_3	3	5	10	3	7	5

Table 5. A partitioning problem: two processors, P_1 and P_2, are allocated

	P_1	P_2	P_1	P_2
t_1	$4/13$	$9/13$	0	1
t_2	$9/13$	$4/13$	$9/13$	$4/13$
t_3	$25/34$	$9/34$	$25/34$	$9/34$

Table 6. Assignment probabilities: the first two columns show the initial assignment probabilities of tasks and the last two columns are the result of the first iteration.

$$D_{tj} = \sum_{i \in Q} d_{ij}, \tag{6}$$

$$d_{ij} = \begin{cases} \frac{P_{ij} E_{ij}}{A_i}, & \text{if } t \in [S_i, D_i] \\ 0, & \text{otherwise} \end{cases}$$

where Q refers to the given task set, and S_i and D_i the start time and finish time of task i, respectively.

We illustrate the procedure with the same example used in Section 6. Assume that two processors from the available processors given in Table 3 are allocated. Table 5 shows characteristics of tasks on the allocated processor, namely, P_1 and P_2. In Table 6, initial assignment probabilities associated with each task on each processor are shown (the first two column of the table). The last two column of the same table shows the state of the probability table after an iteration is finished resulting the assignment of Task t_1 to Processor P_2. Figures 5(a) and 5(b) show the respective probabilities of tasks multiplied by their respective distribution over the time frames during which they can be executed.

Clearly, in the example problem, it is not possible to assign all the tasks to one processor and to have a feasible schedule because there are time slots where the sum of distribution graphs is greater than 1 even though the distribution graphs are partitioned across all the processors with the probabilities based on execution times of tasks. At a glance, task t_1 and t_2 cannot be assigned on the same processor since the total execution time of the two tasks is 5 time units on both processors and the time frame of both tasks are less than 5, which means there could be no feasible schedule if the partitioning is done in that way.

We use the self-force S_{ij} of task i on processor j to measure the adverse (or advantageous) effect of assigning task i to processor j. The self-force is computed by summing up the force F_{tij} of task i on processor j at time t.

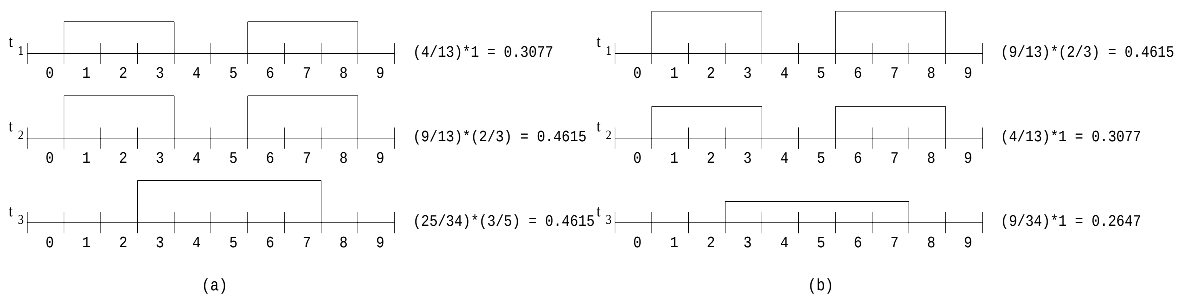


Figure 5. Distribution graph of each task on P_1 (a) and P_2 (b).

$$F_{tij} = D_{tj} x_{tij}, \tag{7}$$

$$S_{ij} = \sum_{t \in T} F_{tij} + \sum_{m \in R} \sum_{t \in T} F_{tim}, \tag{8}$$

where x_{tij} denotes the change of task i's distribution probability on a processor j at time slot t when it is assigned to a processor, T the time frame of task i, and R the set of allocated processors excluding processor j. The first term of the right-hand side of the Equation (8) measures the positive force when the task i is assigned to the processor j. If the positive force is too big relative to the negative force measured by the next term, there is no benefit in assigning the task i on the processor j.

For example, $S_{12} = 0.7693 \times (4/13) \times (2/3) \times 3 + 1.0339 \times (4/13) \times (2/3) \times 3 - 0.7692 \times (4/13) \times 1 \times 3 - 1.2104 \times (4/13) \times 1 \times 3 = -0.7178$. The tentative assignment of the task 1 to the processor 2 imposes positive force to the processor 2 and negative force to the processor 1 and the changes in distribution probabilities are computed as $x_{t12} = 1 - 9/13 = 4/13$ and $t_{t11} = 0 - 4/13 = -4/13$, $t \in$ [1, 3] and [6, 8] as shown in the computation of S_{12}. Of the values of the self-forces for the example, this is the minimum. This can be interpreted as assigning the task 1 to the processor 2 is the best choice in terms of load-balancing and maximizing schedulability.

The algorithm assigns tasks to processors one at a time. When a task is identified to be best assigned on a processor, the probability table is updated. As a result of assigning t_1 of the example shown in Table 5 on P_2, we get the updated probability table given in Table 6. After updating the probability table, the values of the self force for the rest of the tasks with the new probability table are computed using the same procedure. In our example the algorithm assigns t_2 on P_1 and finally t_3 on P_1.

As is the case for the resource allocation step, if there are dependency constraints, we precompute time frames for each task on each processor according to the procedure explained in Section 5. The new time frame ensures that the distribution of task executions on each processor is correct and gives good assignment solutions.

Table 7 shows an assignment problem with precedence constraints. Task t_2 requires data from task t_1. The time

	P_1	P_2	T	S	D	A	Time Frame (P_1)	Time Frame (P_2)
t_1	3	2	9	1	7	7	$[1,5]$	$[1,4]$
t_2	2	3	9	1	7	7	$[4,7]$	$[3,7]$
t_3	3	5	9	3	7	5	$[3,7]$	$[3,7]$
t_4	4	4	9	1	6	6	$[1,6]$	$[1,6]$

Table 7. A partitioning problem with dependency constraints: t_2 requires data from t_1 (P_1 and P_2 are allocated). The last two columns show time frames of respective tasks on each processor. The time frames are computed according to the procedure presented in Section 5

Task	P_{i1}	d_{i1}	P_{i2}	d_{i2}
1	144/(77*9)	(3/5)*P_{11}	3600/(1444*4)	(2/4)*P_{12}
2	144/(77*4)	(2/4)*P_{21}	3600/(1444*9)	(3/5)*P_{22}
3	144/(77*9)	(3/5)*P_{31}	3600/(1444*25)	(5/5)*P_{32}
4	144/(77*16)	(4/6)*P_{41}	3600/(1444*16)	(4/6)*P_{42}

Table 8. The initial tentative assignment probabilities d_{ij}'s in Equation 6 for the partitioning problem given in Table 7

frames of each task on each processor are shown in the last two columns of the table. After new time frames are computed, we use the same assignment algorithm described above. Note that when the time frames are changed the available time to each task (represented by A_i in Equation 6) changes. The initial tentative assignment probabilities d_{ij}'s in Equation 6 are computed and shown in Table 8. The initial and final distribution graph is shown in Figure 6 and 7, respectively. The assignment algorithm can be summarized by the pseudo-code given in Figure 8.

8. Task-Level Scheduling

In this section, we present the task-level scheduler in greater detail. First a set of observations and task scheduling approaches based on the observations are explained. In the following subsection, the heuristic used in the task-level scheduler is given. Finally, this section concludes with the task scheduling algorithm.

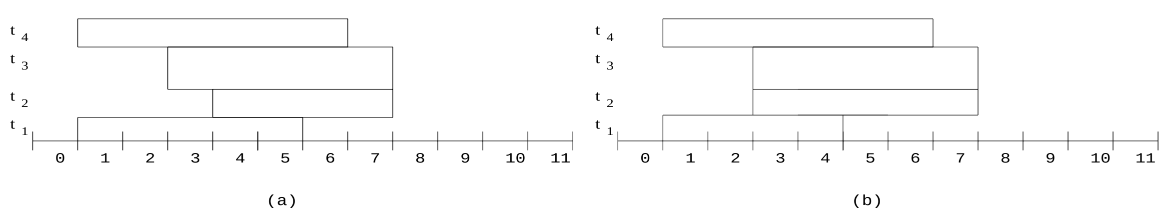


Figure 6. Distribution graphs of tasks with precedence constraints on P_1 (a) and P_2 (b)

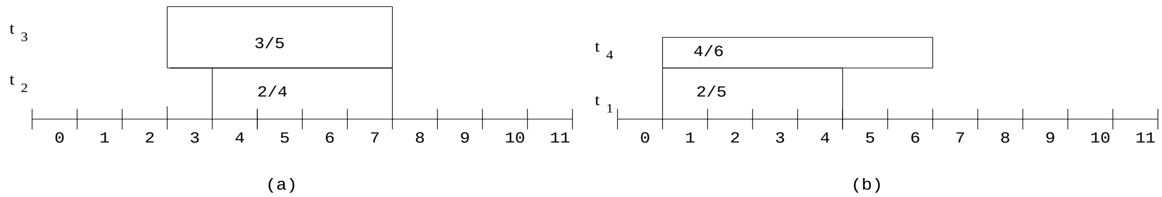


Figure 7. The result of assignment: the tasks shown in Figure 6 are assigned to one of the processors P_1 and P_2

```
Assignment ( ) {
   construct the initial assignment table;
   do {
      MIN = a large number;
      for all unassigned tasks {
         for all allocated processors {
            (Processor, Task, Self-Force) = Compute_Self_Force;
            if ( self-force < MIN ) {
               MIN = Self-Force;
               savedProcessor = Processor;
               savedTask = Task;
            }
         }
      }
      update the assignment probability table with savedProcessor and savedTask;
   } while ( the number of unassigned tasks != 0 )
}
```

Figure 8. The assignment (partitioning) algorithm

8.1. Observations and Approaches

The basis for our scheduling algorithm is the EDF scheduling policy [47]. The EDF is not always optimal for our synthesis problems. The following summary of observations, which can be easily proved or are already proved, guided us to develop an effective and efficient heuristic for the EDF-based task level scheduler.

- The scheduling problem with non-preemptive tasks and arbitrary release times cannot be optimally solved using the EDF.
- The EDF gives the shortest schedule if it is possible for the EDF to find a feasible schedule at all.
- Any sequence is optimal if all the tasks have the same start time and the same deadline.
- The EDF is optimal if all the tasks have the same deadline and different start times.
- The EDF is optimal if the deadlines are non-decreasing when the tasks are ordered in the non-decreasing order of the start times (i.e., tasks are released according to the order of their respective deadlines).

If a set of tasks does not have the characteristics for which EDF is optimal, failure of generating a feasible schedule by the EDF does not necessarily mean there could not be a feasible schedule. Based on the observations, we turn our attention to development of a heuristic which transforms the given scheduling problem into a different form in such a way that the EDF can be applied to nearly optimally find a feasible schedule.

The key idea of the transformation is that by delaying the execution of one or more tasks, a feasible schedule can be identified when the EDF is not optimal. This situation arises when a task has an earlier deadline than that of one started earlier than the task. The candidates that will be delayed should have deadlines that are later than that of our target task and start times earlier than the target task. The target task refers to the task that misses its deadline when the EDF is used. When there are more than one candidate, we select one of them in such a way that the number of unused time slots and overlaps over time slots are minimized. As will be described later, our procedure incorporates a very effective means to take this issue into account. If every deadline is met, we have a feasible schedule. The algorithm tries to find a way to make one task meet its deadline at a time by delaying execution of other tasks given that those tasks can be delayed.

Figure 9 depicts two example scheduling problems. The rectangles indicate respective deadlines and start times of the tasks (i.e., time frames of tasks). The numbers inside the boxes indicate the execution times over available times (i.e., the size of time frames). The EDF cannot find a feasible schedule for the example shown in Figure 9(a) although there exist feasible schedule. Since the problem does not fall into any category in that the EDF is optimal, we try to find a feasible schedule by delaying one or more tasks. In this case, tasks that started (released) before the deadline of task 1 (which is the first of any deadline) are taken to be re-scheduled. By delaying task 3,

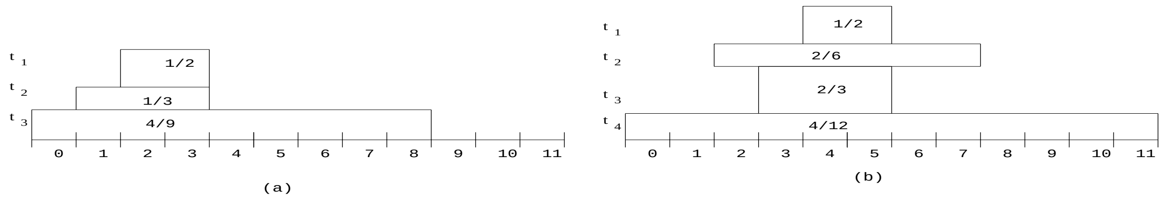


Figure 9. Examples of distribution graphs of tasks to be scheduled.

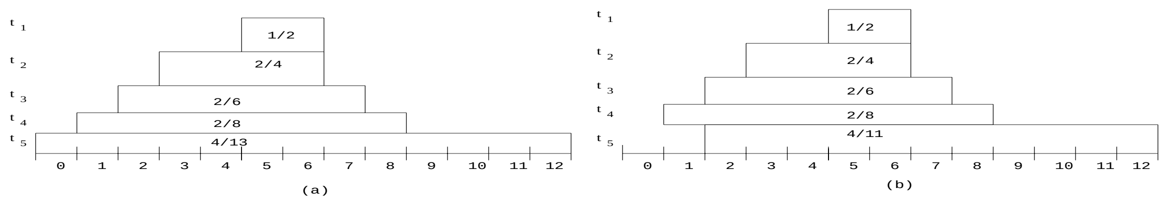


Figure 10. An example scheduling problem requiring the application of force-directed selection of a task to be delayed. (b) shows the distribution graphs after one application of task delay.

we find a feasible schedule using the EDF. Note that in order to have a feasible schedule we must not use the first time slot. The schedule, however, is optimal.

Figure 9(b) shows a scheduling example that requires more than one application of task delays. When the EDF is used without transformations, the schedule would be $t_4 \rightarrow t_3 \rightarrow t_1 \rightarrow ...$, which is not feasible. Note that there is only one task the start time of which can be delayed, namely, t_4. After changing the start time of t_4 to the start time of t_3, the EDF generates the schedule $t_2 \rightarrow t_3 \rightarrow t_1 \rightarrow ...$. Again it is not feasible. By moving the start time of t_2 to the start time of t_3, the EDF finds a feasible schedule.

In general, if a task misses its deadline when the strict EDF is used, we reshuffle start times of tasks such that the EDF can find a feasible schedule if any. The scheduling problems given in Figure 9 (a) and (b) are easy in the sense that we have only one delay candidate at a time.

Next consider the example shown in Figure 10. The schedule by the EDF without transformations would be $t_5 \rightarrow t_2 \rightarrow t_1 \rightarrow t_3 \rightarrow ...$. This is no feasible schedule since t_3 misses its deadline. In this case there are more than one tasks that can be delayed (t_4 and t_5). The choice among possible candidate tasks that can be delayed will impact the feasibility and effectiveness of the procedure. For example, if t_4 is chosen, the EDF cannot find a feasible schedule. On the other hand, if t_5 is chosen, we find a feasible schedule although the processor will be idle at time 1. By delaying the start time of t_5, the modified EDF finds a new feasible sequence $t_4 \rightarrow t_2 \rightarrow t_1 \rightarrow t_3 \rightarrow t_5$.

Generally, if there are more than one delay candidate, as in the scheduling problem given in Figure 10, scheduling is computationally difficult. We propose as an efficient heuristic the force-directed delay-based EDF that is described in the following subsection.

Our algorithm first tries EDF to find a feasible schedule. If no feasible schedule can be found by EDF, then it checks the given set of tasks to see if EDF is optimal (refer to the criteria given earlier in this subsection). If EDF is optimal, there is no a feasible schedule. Otherwise, it selects and delays a task at a time using a force-directed methodology [31] and tries to generate a feasible schedule.

8.2. Delay Candidate Selection using Force-Directed Methodology

Task distribution is defined as a probability by which a task demands a particular time slot for its execution. For example, if the start time of a task is 0, the execution time 2, and the deadline 6, then the distribution of the task is $2/7$ from time 0 through time 6. By taking the summation of the probabilities of all tasks, we obtain a distribution graph. The resulting distribution graph in our problem indicates the demand at a time slot for the resource requested by all the tasks. The distribution graph at time 2 in Figure 9(a) is $4/9 + 1/3 + 1/2 = 37/30$, which means that at least one task must be able to be scheduled not claiming the time slot 2 in order to have a feasible schedule. Interestingly, if task 3 is scheduled by the EDF, it must use the time slot 2 and there is no feasible schedule for the task set. A distribution graph D_t at time t is given by

$$
\begin{aligned}
D_t &= \sum_{i \in Q} d_i, \qquad (9)\\
d_i &= \begin{cases} \frac{E_{ij}}{A_i,} & \text{if } t \in [S_i, D_i] \\ 0, & \text{otherwise} \end{cases}
\end{aligned}
$$

where Q is the set of tasks assigned to the processor.

Each task has a force associated with each time slot in its time frame which reflects the effect of an attempted delay of a start time of a task on the overall resource demand requested by all the tasks. A force F_{ti} of task i at time t is given by

$$F_{ti} = D_t x_{ti}, \qquad (10)$$

where x_{ti} is the change of task i's probability of using the time slot t. Note that since we need to avoid fragmentation of time slots (i.e., existence of unused time slots) to maximize schedulability, unlike the self-force described by Paulin and Knight [31], the force in our problem will have positive value when the new D_t as a result of changing the start time of task i is lower than 1.0 at time slots between the original start time and a new start time of task i. Hence, x_{ti} is positive in that case even when the change *reduces* the force at the time slots.

In general, we select a task to be delayed in such a way that it minimizes the chance of causing holes (unused time slots) and the possibility of overloading time slots, it is desirable to ensure distribution at each time slot to be as close as possible to 1.0. If the value is too low, then there is the risk of fragmentation. If it is too high then there is very low chance of having a feasible schedule.

The self-force S_i associated with delaying the start time of task i is given by

$$S_i = \sum_{t \in T} F_{ti}, \tag{11}$$

where T is the time frame of task i.

To illustrate the application of the self-force to choose a task to be delayed, we use the scheduling example given in Figure 10(a). Since task 3 misses its deadline if the EDF is used without transformations as explained previously, we have two delay candidates, namely, task 4 and 5. We choose a delay candidate based on the following computation: $S_4 = \sum_t F_{t4}$, $t = 1, 2, 3, ..., 8$ and $S_5 = \sum_t F_{t5}$, $t = 0, 1, 2, ..., 12$.

From the two resulting numbers of the self-forces, we see that changing the start time of task 5 has less overall adverse effect on the schedulability. In fact, changing the start time of task 4 will lead to no feasible schedule. The point that we have to have minimal amount of unclaimed holes at the origin side of the time axis is valid in the sense that the schedulability might be hampered later on if we do not use all the possible time slots. In the example, the EDF now finds a feasible schedule.

8.3. Scheduling Algorithm

Our scheduling algorithm checks to see if there is a feasible schedule for the given set of tasks along the time axis starting from the origin. If a deadline miss is encountered, we choose one of the delay candidates and delay the start time of the selected task. The detailed scheduling procedure is given in Figure 11.

If the given synthesis problem has precedence constraints, we need to schedule all the allocated processors at the same time whereas we can generate a schedule for all the assigned tasks of a processor at a time otherwise. In general, we keep generating a schedule for a processor until a task that requires data from another task is encountered. When a task has a dependent task, we adjust the time frame of the dependent task such that the communication cost is taken into account. From the point of an individual processor, the scheduling procedure for synthesis problems with precedence constraints is the same as those with no such constraints: the only information

```
Scheduling ( ) {
   schedule_found = FALSE;
   delay_candidate = TRUE;
   while ((!schedule_found) && (delay_candidate)) {
      deadline_miss = EDF( );
      if (deadline_miss) {
         candidate = Candidate(deadline_miss);
         if (!candidate) delay_candidate = FALSE;
         else Adjust_Start(candidate);
      }
      else
         schedule_found = TRUE;
   }
}
```

Figure 11. The scheduling algorithm

the scheduler uses is given by time frames of tasks; and we embedded all the precedence requirements in the problem in the form of time frames.

For example, consider the result of a partitioning given in Figure 6(b). We start generation of a schedule for P_1. After scheduling t_3, we cannot proceed any more since t_2 is dependent on t_1. At this point we suspend scheduling of P_1 and go on with the next processor. As soon as t_1 is scheduled on P_2, we adjust the time frame of t_2 to take the communication cost into account.

9. Experimental results

In order to evaluate the effectiveness of the proposed approach and the optimization algorithms we tested our implementation of the synthesis system for hard real-time systems on a number of examples. The tasks are either synthetic or industrial DSP, communication, and control examples adopted from Potkonjak and Wolf [34]. Table 9 shows one of our experimental task sets with no precedence constraints. The first four columns of the table show run times of tasks on the given processors and the last four columns show period, start time, deadline, and available (deadline - start time) time for each task. The problem consists of 20 tasks and 4 processors to choose from. Each processor has an associated cost given in Table 10.

The synthesis result is shown in Table 11, 12 and 13. The LCM of periods of tasks, lower bound cost computed by taking the summation of $M_i's$ given in Equation 2, actual cost, allocated processors and the run time of synthesis algorithms are summarized in Table 11. Note that the utilization factor of each processor shown in Table 12 is remarkably high considering that the task set consists of non-preemptive tasks with arbitrary release times and deadlines [24] [25] [30]. Task assignments are shown in Table 13.

Table 14 presents a synthesis problem with precedence constraints. Processor costs are given in Table 15. The precedence constraints are represented in Figure 12. The results for the problem exhibit the similar characteristics to ones for problems with no precedence constraints. The utilization factors for allocated processors are summarized in Table 16. The task assignment is shown in Table 17.

Summaries of further experimental results are given in Table 18 and 19 for task sets with no precedence constraints and with precedence constraints, respectively. Improvement factors of the solution by our approach over ones by a greedy heuristic algorithm are shown for comparison purposes. As shown in the table, the utilization factors of resources are high and improvements over the greedy solutions are always by at least 20%, which indicate high quality results.

	P_1	P_2	P_3	P_4	T	S	D	A
t_1	1	2	5	3	20	3	12	10
t_2	2	2	6	4	20	4	15	12
t_3	2	4	5	7	20	7	17	11
t_4	2	5	9	6	20	1	14	14
t_5	4	5	3	9	20	0	18	19
t_6	5	7	7	6	40	17	28	12
t_7	4	7	9	5	40	4	23	20
t_8	7	5	9	7	40	15	27	13
t_9	2	5	8	5	40	10	30	21
t_{10}	6	8	7	8	40	0	30	31
t_{11}	3	8	10	10	40	3	35	33
t_{12}	6	6	12	10	50	15	40	26
t_{13}	2	7	10	8	50	0	35	36
t_{14}	4	3	12	10	50	9	25	17
t_{15}	5	6	14	10	50	0	40	41
t_{16}	3	8	9	7	50	7	45	39
t_{17}	7	9	10	10	50	30	40	11
t_{18}	6	10	12	10	60	20	45	26
t_{19}	8	5	10	6	60	15	55	41
t_{20}	4	8	7	7	60	0	45	46

Table 9. The execution time and timing constraints table of an experimental synthesis problem with no precedence constraints

P_1	P_2	P_3	P_4
40	30	15	20

Table 10. The cost vector of the processors available for the experimental synthesis problem in Table 9

LCM	lower bound cost	actual cost	allocated processors	running time
600	53.87	95.00	3 (P_1, P_1, P_3)	2 sec.

Table 11. The summary of results for the problem in Table 9

P_1	P_1	P_3	average
0.76	0.82	0.49	0.69

Table 12. The utilization factors of the allocated processors for the problem in Table 9

P_1: 1, 2, 8, 11, 13, 14, 17, 18	P_1: 3, 4, 6, 7, 9, 12, 15, 16, 20	P_3: 1, 5, 10, 19

Table 13. The final task assignment to allocated processors for the problem in Table 9

	P_1	P_2	P_3	P_4	T	S	D	A
t_1	3	4	3	8	150	25	144	120
t_2	10	10	8	14	150	25	144	120
t_3	12	10	5	14	150	25	144	120
t_4	13	12	7	15	150	25	144	120
t_5	16	15	7	17	150	25	144	120
t_6	13	15	12	19	150	25	144	120
t_7	15	12	6	14	150	25	144	120
t_8	30	26	21	30	150	25	144	120
t_9	20	17	12	21	150	25	144	120
t_{10}	15	14	14	19	150	25	144	120
t_{11}	7	6	2	10	50	0	48	49
t_{12}	4	5	3	9	50	0	48	49
t_{13}	9	9	3	8	50	0	48	49
t_{14}	14	12	3	16	50	0	48	49
t_{15}	12	12	10	18	75	15	65	51
t_{16}	15	10	6	14	75	15	65	51
t_{17}	14	12	6	12	75	15	65	51
t_{18}	10	8	4	12	75	15	65	51
t_{19}	5	5	5	8	75	15	65	51
t_{20}	15	15	7	14	75	15	65	51
t_{21}	4	4	4	8	50	10	44	35
t_{22}	3	4	3	8	50	10	44	35
t_{23}	5	5	5	6	50	10	44	35
t_{24}	7	5	6	9	50	10	44	35

Table 14. The execution time table of an experimental synthesis problem with precedence requirements

P_1	P_2	P_3	P_4
40	50	60	35

Table 15. The cost vector of the processors available for the example in Figure 14

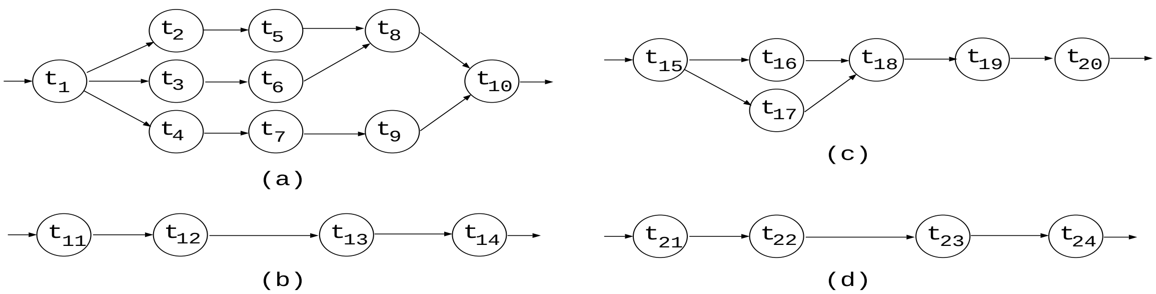


Figure 12. Precedence constraints of the experimental synthesis problem shown in Table 14: circles indicate tasks and dependencies are shown by arrows

P_1	P_3	average
0.94	0.773	0.857

Table 16. The utilization factors of allocated processors for the problem in Table 14

P_1: 1, 2, 6, 8, 10, 15, 19, 21, 22, 23	P_3: 3, 4, 5, 7, 9, 11, 12, 13, 14, 16, 17, 18, 20, 24

Table 17. The final task assignment for the example in Figure 14

Number of Tasks	32	34	36	38	40	44	46	48	50	52
Greedy Solution	560	605	615	810	665	680	845	890	1015	855
Optimized Solution	430	435	450	610	445	530	525	610	680	545
Cost Reduction (%)	23	28	27	25	33	22	38	31	33	36
Resource Utilization	0.692	0.634	0.613	0.713	0.573	0.676	0.572	0.607	0.799	0.603

Table 18. Summary of experimental results for task sets without precedence constraints: results are compared to the respective greedy solutions

Number of Tasks	25	30	34	36	38	45	48	50	52	55
Greedy Solution	570	510	615	910	790	820	855	910	885	920
Optimized Solution	340	395	495	595	605	560	535	570	695	605
Cost Reduction (%)	40	23	20	35	23	32	37	37	21	34
Resource Utilization	0.621	0.687	0.680	0.619	0.776	0.616	0.586	0.609	0.789	0.790

Table 19. Summary of experimental results for task sets with precedence constraints: results are compared to the respective greedy solutions

10. Conclusions

A system-level approach for the synthesis of multi-task hard real-time application specific systems using a set of heterogeneous off-the-shelf processors is presented. The approach takes into account task precedence constraints using the notion of time frames. The optimization goal is to select a minimal cost multi-subset of processors while satisfying all the required timing and precedence constraints.

Since many optimization techniques are available as convenient starting points for individual optimization problems, we divided the synthesis process into three distinctive subproblems: resource allocation, assignment (partitioning), and scheduling. Although the interactions between them are complex, the division of the problem into subproblems offers many advantages. The approach enables us to easily apply available optimization techniques. It naturally enforces modularity, flexibility and reusability of algorithms and software. The basic premise of the synthesis approach is that by developing effective and efficient heuristics to deal with each subproblem and combining them in a reasonable manner, local optima can be escaped while preserving the advantages of solving a relatively small subproblem at a time. We adopted an A* search based technique for the resource allocation. For the assignment we use a variation of the force-directed optimization technique to assign a task to an allocated processor. The final scheduling of hard-real time tasks is done by our scheduler based on the Earliest Deadline First (EDF) scheduling policy. The task-level scheduler is a unique modification of EDF in that we applied the force-directed scheduling methodology to address the situations where the EDF policy is not optimal. We demonstrated the effectiveness of the approach through extensive experimental results.

References

[1] J.A. Bannister and K.S. Trivedi. Task allocation in fault-tolerant distributed systems. *Acta Informatica*, 20(3):261–281, 1983.

[2] E. Barros, W. Rosenstiel, and X. Xiong. A method for partitioning UNITY language in hardware and software. In *Proceedings of Euro-DAC '94*, pages 220–225. IEEE Computer Society Press, 1994.

[3] J. Borel. Technologies for multimedia systems on a chip. In *1997 IEEE International Solid-State Circuits Conference*, pages 18–21, 1997.

[4] R.W. Brodersen. The network computer and its future. In *1997 IEEE International Solid-State Circuits Conference*, pages 32–36, 1997.

[5] A. P. Chandrakasan et al. Hyper-LP: A design system for power minimization using architectural transformations. In *Proceedings of ICCAD '92*, pages 300–303, Santa Clara, CA, November 1992. Intl. Conf. Computer-Aided Design.

[6] S. Chaudhuri and R. A. Walker. Computing lower bounds on functional units before scheduling. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(2):273–279, June 1996.

[7] R. F. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report SMLI 93-12, UWCSE 93-06-06, Computer Science and Engineering, University of Washington, 1993.

[8] H. S. Dana. Adding fast interrupts to superscalar processors. Technical Report CSG Memo 366, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge MA 02139, USA, December 1994.

[9] R. Dechter and J. Pearl. Generalized best-first strategies and the optimality of A*. *Journal of the ACM*, 32(3):505–536, 1985.

[10] ETSI. European digital cellular telecommunications system (phase 1): Work programme reference: Gsm 06.10, 1992.

[11] D. D. Gajski et al. *High-level Synthesis: Introduction to Chip and System Design*. Kluwer Academic, Boston, 1992.

[12] D. D. Gajski, S. Narayan, L. Ramachandran, F. Vahid, et al. System design methodologies: Aiming at the 100 h design cycle. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(1):70–82, March 1996.

[13] D. D. Gajski, F. Vahid, , and S. Narayan. A system-design methodology: Executable specification refinement. In *Proceedings of Euro-DAC '94*, pages 458–463. IEEE Computer Society Press, 1994.

[14] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, NY, 1979.

[15] L. Guerra. Personal communication, June 1997.

[16] L. Guerra, M. Potkonjak, and J. Rabaey. High-level synthesis for reconfigurable datapath structures. In *Proceedings of ICCAD '93*, Santa Clara, CA, 1993. Intl. Conf. Computer-Aided Design.

[17] R. K. Gupta and G. De Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design & Test of Computers*, 10(3):29–41, 1993.

[18] C-H. Huang, J-Y. Yen, and M. Ouhyoung. The design of a low cost motion chair for video games and mpeg video playback. *IEEE Transactions on Consumer Electronics*, 42(4):991–997, 1996.

[19] IEEE. *Real-Time Extensions to POSIX*. IEEE, New York, NY, 1991.

[20] IEEE. *Futurebus+ Recommended Practice*. IEEE, New York, NY, 1993.

[21] T. B. Ismail, K. O'Brien, and A. Jerraya. Interactive system-level partitioning with PARTIF. In *Proceedings, Euro-DAC '94*, pages 464–468, 1994.

[22] R. Karri and A. Orailoglu. Transformation-based high-level synthesis of fault-tolerant ASICS. In *Proc. 29th ACM/IEEE Design Automation Conference*, pages 662–665, 1992.

[23] E.L. Lawler. Optimal sequencing of a single machine subject to precedence constraints. *Management Science*, 19, 1973.

[24] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithms - exact characterization and average case behavior. In *IEEE Real-Time System Symp.*, pages 181–191, 1986.

[25] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of ACM*, 20(1):46–61, 1973.

[26] J. Liu, M. Lajolo, and A. Sangiovanni-Vincentelli. Software timing analysis using HW/SW cosimulation and instruction set simulator. In *Proceedings of the Sixth International Workshop on Hardware/Software Codesign (CODES/CASHE'98)*, pages 65–69, 1998.

[27] M. C. McFarland, A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–317, 1990.

[28] R. Nagarajan and C. Vogt. Guaranteed performance of multimedia traffic over the token ring. Technical Report 439210, IBM-ENC, Heidelberg, Germany, 1992.

[29] V. Nirkhe and W. Pugh. Partial evaluation of high-level imperative programming languages, with applications in hard real-time systems. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–280, 1992.

[30] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[31] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of ASICS. *IEEE Transactions on CAD*, 8(6):661–679, June 1989.

[32] M. Potkonjak and J. Rabaey. Maximally fast and arbitrarily fast implementation of linear computations. In *Proc. ICCAD '92*, pages 304–308, Santa Clara, CA, 1992. IEEE Intl. Conf. Computer-Aided Design.

[33] M. Potkonjak and J. Rabaey. Optimizing resource utilization using transformations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(3):277–292, March 1994.

[34] M. Potkonjak and W. H. Wolf. Cost optimization in ASIC implementation of periodic hard real-time systems using behavioral synthesis techniques. In *ICCAD95*, pages 446–451. International Conference on Computer-Aided Design, 1995.

[35] S. Prakash and A. C. Parker. Synthesis of application-specific multiprocessor systems including memory components. *Journal of VLSI Signal Processing*, 8(2):97–116, October 1994.

[36] J. M. Rabaey and M. Potkonjak. Estimating implementation bounds for real time DSP application specific circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(6):669–683, June 1994.

[37] K. Ramamritham and J.A. Stankovic. Scheduling algorithms and operating system support for real-time systems. *Proc. of the IEEE*, 82(1):55–67, January 1994.

[38] R.S. Ratner, E.B. Shapiro, H.M. Zeidler, S.E. Wahlstrom, C.B. Clark, and J. Goldberg. Design of a fault tolerant airborne digital computer. In *Computational Requirements and Technology*, volume 2. SRI Final Report, NASA Contract NAS1-10920, 1973.

[39] M. Rim, A. Mujumdar, R. Jain, and R. de Leone. Optimal and heuristic algorithms for solving the binding problem. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(2):211–225, June 1994.

[40] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1995.

[41] B. Schneier. Lecture notes in computer science 809: Fast software encryption. In R. Anderson, editor, *CAMBRIDGE SECURITY WORKSHOP*, pages 191–204. Springer-Verlag, 1994.

[42] D. B. Schwartz. ATM scheduling with queuing delay predictions. *Computer Communication Review*, 23(4):205–211, October 1993.

[43] L. Sha and J. B. Goodenough. Real-time scheduling theory and Ada. *IEEE Computer*, 23(4):53–62, April 1990.

[44] L. Sha, R. Rajkumar, and J. Lehoczky. Real-time scheduling support in Futurebus+. In *IEEE 11th Real-Time Systems Symposium*, pages 331–340, 1990.

[45] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. *Proc. of the IEEE*, 82(1):68–82, January 1994.

[46] J. Soukup. Circuits layout. *Proc. of the IEEE*, 69(10):1281–1304, October 1981.

[47] J. A. Stankovic, M. Spuri, M. Di Natale, and G. C. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6):16–25, June 1995.

[48] R. Steinmetz. Analyzing the multimedia operating systems. *IEEE Multimedia*, 2(1):68–84, 1995.

[49] R. A. Walker and R. Camposano. *A Survey of High-level Synthesis Systems*. Kluwer Academic, Norwell, MA, 1991.

[50] W. H. Wolf. Hardware-software co-design of embedded systems. *Proc. of the IEEE*, 82(7):967–989, 1994.

[51] H. Yasuda. Multimedia impact on devices in the 21st century. In *1997 IEEE International Solid-State Circuits Conference*, pages 28–31, 1997.

[52] T.-Y. Yen and W. H. Wolf. Communication synthesis for distributed embedded systems. In *ICCAD95*, pages 288–294. International Conference on Computer-Aided Design, 1995.