System-level 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

ABSTRACT

We present a new, system-level approach for the synthesis of multi-task hard real-time applications off-the-shelf processors. The goal is to select a set of processors with minimal cost while satisfying timing constraints. In our approach there are three design phases: resource allocation, assignment (partitioning), and scheduling. With the observation that the resource allocation is a search for a set of processors which requires the minimum cost, we adopted A* search based technique. For the assignment we use a variation of the force-directed optimization technique to determine the effect of assigning a task onto a processor. The final scheduling of hard-real time tasks is done by our scheduler based on the Earliest Deadline First (EDF) scheduling. Our task-level scheduler is a unique modification of the EDF in that we applied the force-directed scheduling methodology to address the situations where the EDF is not optimal. The experimental results on a variety of examples show that this approach is highly effective on a variety of examples.

1.0 Introduction

1.1 Motivation

Real-time systems can be defined as those systems in which the proper functioning of the implementation depends not only on the logical correctness of the computation, but also on the time at which the results are produced [Ram94]. Classical examples include automobile and airplane monitoring and control systems, a variety of command and control systems, and process control systems. The number of modern real-time systems has been growing exponentially, mainly in connection with wireless and broadband communication. Multimedia, video-on-demand, interactive television, and remote sensor security control are among the most popular applications today and form a fast growing segment of the market. Modern real-time systems are often intrinsically meant to be running multiple task applications. For example, a video server has to handle simultaneous requests from several users and it should be able to assemble and deliver both video and sound components in response to such requests.

Both traditional behavioral synthesis and emerging system synthesis have been focused on synthesis of single task applications [Gaj92, Wal91, McF90]. It is fairly recent development that a technique for synthesis of multi-task applications at the behavioral level has been proposed [Pot95]. Also only a few research groups addressed synthesis of hard real-time systems [Pra94, Yen95], usually under restricted designed scenarios.

Our goal of the research presented in this paper is to develop a modular, flexible, and reusable synthesis tools for system level synthesis of multi-task hard real-time application specific systems. Therefore, the synthesis techniques presented in this paper can be used to supplement the existing works on system level synthesis. The single most prominent example among the new generalized optimization techniques is that our approach utilizes the powerful A* search for resource allocation.

The research presented in this paper could be also viewed from another interesting and important angle. Scheduling has been traditionally considered key backbone task of behavioral synthesis. For example, Gajski's statement that scheduling is the single most important behavioral synthesis step 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 and that other synthesis tasks, such as transformations, usually make greater differences in the final results. Moreover, it was reported that the available scheduling algorithms often produce optimal results.

The focus of behavioral synthesis research, therefore, shifted from the scheduling to the other issues of synthesis that were found to be more beneficial if addressed. However, it seems that we are on the brink of renaissance in the scheduling (and resource allocation and assignment[1]). Task-level scheduling will provide numerous avenues for high impact research and industry-relevant tool development.

1.2 What is New?

This is the first paper, to the best of our knowledge, which presents a fully modular approach for synthesis of hard real-time applications. Also this is the first published work that explores the possibility of resource allocation optimization using A* search technique. We established computational complexity of all individual synthesis steps.

1.3 Paper Organization

The rest of the paper is organized in the following way. We first provide background material and formulate three synthesis subproblems and establish their computational complexity. In Section 4, we describe the global design flow of the new approach. We elaborate about the technical details of resource allocation, assignment, and scheduling algorithms in Sections 5, 6, and 7 respectively. In Section 8 we present and analyze the experimental results. The last two section summarize the related research efforts and the key ideas and results.

2.0 Preliminaries

In this section we summarize key information about design abstractions and models used in the research described in this paper.

We assume that all tasks are defined on semi-infinite or very long streams of data. All tasks are periodic. For each task, three timing constraints are imposed: the period, the start time (the earliest time when all required data for one iteration are available) and the finish time (the latest time by which task has to be completed, i.e. output data for a given iteration provided to the user). All the instances in a task share the identical periodic timing constraints. For each task execution time or upper bound on the execution time on each of the available processors is tabulated. Since the tasks are independent, there is no

1. We use terms assignment and partitioning interchangably.While they denote the same design activity, term assignment is used traditionally at the behavioral level while term partitioning at the system level.

communication cost. Note that independence assumption among tasks is actually done with no loss of generality, since the atomic dependent tasks can be always treated as a single composite task.

Two implementation constraints are imposed. The first constraint is that no preemption is allowed. Preemption drastically simplifies many synthesis problems. For example, while scheduling is an optimization problem of polynomial complexity when preemption is assumed, it is NP-complete problem when no preemption is allowed [Sta95]. However, very high context switching times for modern computing systems, clearly indicate that switching cost is most often prohibitevely high. Note that once switching cost is taken into account, many optimization problems reestablish their computational intractability.

The second restriction is that all instances of a periodic task should be executed on the same processor. An instance of a task refers to an execution of a task within the least common multiple of periods (LCM) of the given set of tasks. For example, if the LCM of the periods of tasks is 30 and the period of a task is 5, then there are 6 instances of the task within the LCM of the periods. Each of six instances has its own timing constraints and all the instances are to be scheduled on the same processor.

3.0 Problem Formulation and Complexity

In this section we formulate the optimization problems associated with three system-level synthesis steps and establish their computational complexity. Note, that no preemption is allowed, due to high context switching cost [Sta95]. The targeted synthesis problems can be defined in the more formal terms as follows using the standard Garey-Johnson [Gar79] format:

Allocation: A set of *k* processors and a set of n independent periodic hard real-time tasks are given. Each processor has an associated cost. Each task has its period, execution time for implementation on any of the available processors and the relative periodic start and deadline timing constraints for each iteration. Select a multisubset of processors (subset where some processors can be include more than once) so that each tasks is assigned to exactly one processor and that the sum of costs of the selected processors is at most K.

Assignment (Partitioning): A set of k processors and a set of n independent periodic hard real-time tasks are given. Each task has its period, execution time for implementation on any of the available processors and the relative periodic start and deadline timing constraints for each iteration. Assign each task to one of the processors in such a way that all tasks can be scheduled within their timing constraints.

Scheduling: A set of n independent periodic hard real-time tasks is given. Each task has its period, its execution time, and the relative periodic start and deadline timing constraints for each iteration. The goal is to generate non-preemptive schedule of the tasks so that all timing constraints for all tasks are satisfied.

Note that the assumption that the tasks are independent is not restrictive, since a set of tasks which are dependent can be treated as a new large task. We proved that the allocation problem is NP-complete by transforming the equal subset problem [Gar79] into a restricted instance of the resource allocation problem using Karp's polynomial transformation techniques. We also proved using the same technique that assignment is NP-complete problem. As a starting point in the proof we used the knapsack problem [Gar79]. For the sake of brevity, proofs are not presented in this version of the paper. Finally, note that the hard-real time scheduling problem with non-preemptive scheduling policy and arbitrary start times is also NP-complete, as proven previously [Sta95].

4.0 Algorithm Overview

As we mentioned in the previous sections, the synthesis problem of minimal cost implementation of a set of real-time tasks can be viewed as a set of several layered NP-complete problems. The difficulty of the optimization problem is more related to interaction between the NP-complete problems, then the computational complexity of individual problems for which numerous optimization technqiues are available as convenient starting points. To take advantage of this observation, we divided the synthesis problem into three distinctive subproblems. The division of the problem into subproblems enables to enforce modularity, flexibility, and reusability of algorithms and software. The basic premise of the synthesis approach is to develop an effective and efficient heuristics to deal with each subproblem and combine them in some reasonable manner so that 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 following pseudo code:

```
System-level synthesis of hard real-time systems
repeat
 Allocation();
 Assignment();
 Scheduling();
until (a set of feasible schedules is generated);
```

The allocation subtask proposes a set of allocated processors to the assignment and scheduling procedures. The assignment procedure, in turn, assigns tasks to processors and passes the result over to the scheduling procedure. Finally, the scheduling procedure generates a feasible schedule for each allocated processor if there exists one. If there is no feasible schedule, the allocation procedure enters again at the point where it left before and the procedures are repeated until a set of feasible schedules is obtained. Note that the A* search provides a backtracking provision. We did not exercise this option in our experimentation since the result without backtracking is of high quality, as indicate by lower bound estimations and high run-times when it is used.

The allocation subtask finds a set of resources by searching the solution space using the A* search strategy [Dec85, Rus95]. The 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 of the different processor choices. Recall that in the A* search, the search follows the most promising direction. This decision is made by looking at the sum of the cost of current partial solution and optimistic estimate of the complete solution. Our A* heuristic which guides the search is based on the relaxed assignment and scheduling. The heuristic uses three components: the cost of current solution, the utilization of the proposed set of allocated processors and the estimated additional cost to take care of the tasks that cannot be scheduled on the proposed processors. Once the allocation procedure reaches a tentatively feasible solution it hands the allocation result over to the partitioning procedure.

The assignment procedure assigns a task at a time to a processor. The assignment heuristic is based on the force-directed scheduling [Pau89] 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 force-directed approach provides a good means to measure the effect of assigning a task onto a processor. It is constructed in a such a way to take into account the distribution of demands of computing resources and the running time of tasks on each allocated processor.

Our task-level scheduler is based on the Earliest Deadline First (EDF) scheduling. The EDF is optimal scheduling approach in many cases when preemption is allowed [Sta95]. It is also optimal in several cases of non-preemptive scheduling with arbitrary release times and deadlines. The heuristic used in the scheduler, therefore, is basically to address the situations where the EDF is not optimal. We developed the force-directed delay-based EDF which can be viewed as a variation of the EDF. Experimentation indicates that the modified EDF is so effective that the running time of the scheduling algorithm is essentially proportional to the number of instances of tasks, which is the same complexity of the EDF.

The next three sections describe the three steps, resource allocation, partitioning, and scheduling, in synthesis in more detail.

5.0 Resource Allocation

The allocation algorithm adopts the A* search strategy [Rus95]. The goal is to allocate processors that will implement individual tasks with minimal cost and, at the same time, they will be highly utilized. The heuristic function used in 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 run time of the task. For example, given a set of tasks and processors as in Table 1, the minimum implementation cost of each task is computed as $M[i] = min_j\{C[j]*E[i][j]/T[i]\}$ where *M[i]* is the minimum implementation cost for a task *i*, *C[j]* is the cost of a processor *j*, *E[i][j]* is the computation time of a task *i* on a processor *j*, and *T[i]* is the period of a task *i*. The result is given in the Table 1.

	P1	P2	P3	P4	T (period)	CP1	CP2	CP3	CP4	Min (P)
t1	7	5	3	6	5	-	20	18	-	18 (P3)
t2	8	5	4	9	6	-	16.7	20	-	16.7 (P2)
t3	7	4	2	5	10	10.5	8	6	5	5 (P4)
t4	5	7	3	8	10	7.5	14	9	8	7.5 (P1)
t5	9	8	5	7	15	9	10.7	10	4.7	4.7 (P4)
t6	10	9	4	9	30	5	6	4	3	3 (P4)
cost	15	20	30	10						Sum: 54.9

Table 1: Synthesis problem and implementation cost on each processor: Pi, tj, and CPi denote available processors, tasks, and implementation cost respectively. "-" in [i][j] indicates the task i cannot be implemented on the processor j.

The sum of *M[i]*'s signifies the lower bound of the implementation cost. That is, we have to spend at least 54.9 to implement all the tasks. 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 will be normally 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.

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 checks 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, which is done in the following way.

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 onto 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 the Table 1. Allocating *P1* and *P2* produces the

.	P1	P2	T	S	D	A
t1	3	2	5	1	3	3
t2	2	3	5	1	3	3
t3	3	5	10	3	7	5
cost	16	10				

Table 2: An instance of synthesis problem: T- period, S start time, D- deadline, and - available time.

probability table shown in the Table 2. Each execution time shown in the Table 2 before and

	P1	P2	P1- estimated cost	P1- estimated cost
t1	2/5	3/5	(2/5)*3	(3/5)*2
t2	3/5	2/5	(3/5)*2	(2/5)*3
t3	5/8	3/8	(5/8)*3	(3/8)*5

Table 3: Probability and execution time table for relaxed assignment and scheduling

after multiplication by the probability. The numbers in the table are estimated execution times of corresponding tasks on the combined superprocessor which combines all the computing capacities of the allocated processors.

With the observation that the utilization factor reveals an upper bound [Liu73] for preemptive schedulability and the fact that the non-preemptive scheduling is more difficult (in both terms of checking the schedulability and having a feasible schedule) [Sta95], we can incorporate it in our heuristic function as the means to estimate if a set of allocated processors can be a feasible solution. For example, the utilization of the processor set of *P1* and *P2* given in the Table 5 is (2/5)*(3/5) + (3/5)*(2/5) + (5/8)*(3/10) = 0.67. The utilization provides a good measure as to 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 having a feasible schedule [Pot95], we need to know what portion of the task sets can be scheduled on the allocated processors to proceed with the search. To do that, we perform a relaxed scheduling on the partitioned task set using our force-directed EDF which will be described in the section 7. In the process, if we encounter a task that cannot be scheduled, the task is thrown away and the scheduling continues until it finishes identifying a set of tasks that is scheduled. As described in the section 7, 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 way. With the set of tasks that is scheduled, the utilization is computed. By combining the utilization and the estimate of the future cost that is required for the tasks that were thrown away, we get an estimate of the overall implementation cost. The estimate is given by $\sum_j C[j] + \sum_j C[j](1-U) + \sum_j min_j\{C[j]*E[i][j]/T[i]\}$ where *C[j]* refers to the cost of processor *j*, *U* utilization of allocated processors.

The procedure examines all the cases where each and all processors available are tentatively allocated and they render a new accumulated cost, a utilization factor and a new expected future cost to complete the implementation. Of the estimated costs, the minimum which promises a good solution is chosen. When a set of processors is allocated, the cost, the utilization, and the estimated future cost is checked. If the cost is greater than the minimum and the relaxed partitioning and scheduling are completed successfully for all the given tasks, the actual partitioning of the task set onto the set of allocated processors are performed.

Allocation can be summarized with the following pseudo-code:

```
Allocation ()
repeat
  perform the relaxed assignment and scheduling for all processors;
  collect a processor at a time that is most promising;
  if the relaxed assignment and scheduling are successful
```

```
   then check the utilization factor;
     if the utilization is too high then allocate more resources;
     else go to the actual partitioning;
   else continue;
until all the tasks can be scheduled using relaxed_scheduling;
```

6.0 Assignment

Our assignment procedure is based on the observation that we have the best chance of finding a feasible schedule if we assign tasks onto the allocated processors in such a way that the distribution graphs (refer to section 7 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. Here we modify the force- directed scheduling algorithm for behavioral synthesis [Pau89] to find a good assignment of a given task set. First we define probabilities of each task based on execution times of a task on each processors with which a task is tentatively assigned onto processors as follows:

$$P[i][j] = 1/\left(E[i][j]*E[i][j]*\sum(1/(E[i][j]*E[i][j]))\right)$$

This formula assumes that it is more beneficial to assign a task onto a processor which requires a shorter execution time for the task than to assign it to a processor that executes the task slower. Next, we use a modified force-directed assignment procedure to balance the loads among the processors and to make distribution graphs on all the processors be as even as possible. In addition to the probabilities, we compute the overall distribution of the tasks over the allocated processors by: $DG(task\ i,\ processor\ j) = P[i][j]*(E[i][j]/A[i])$

We illustrate the procedure using the following example. The Table 4 shows characteristics

	P1	P2	T	S	D	A
t1	3	2	5	1	3	3
t2	2	3	5	1	3	3
t3	3	5	10	3	7	5

Table 4: An illustrative example for partitioning

of each task and each processor. In the Table 5, probabilities associated with each task on each processor are computed. Figures 1 and 2 show the respective probabilities of tasks

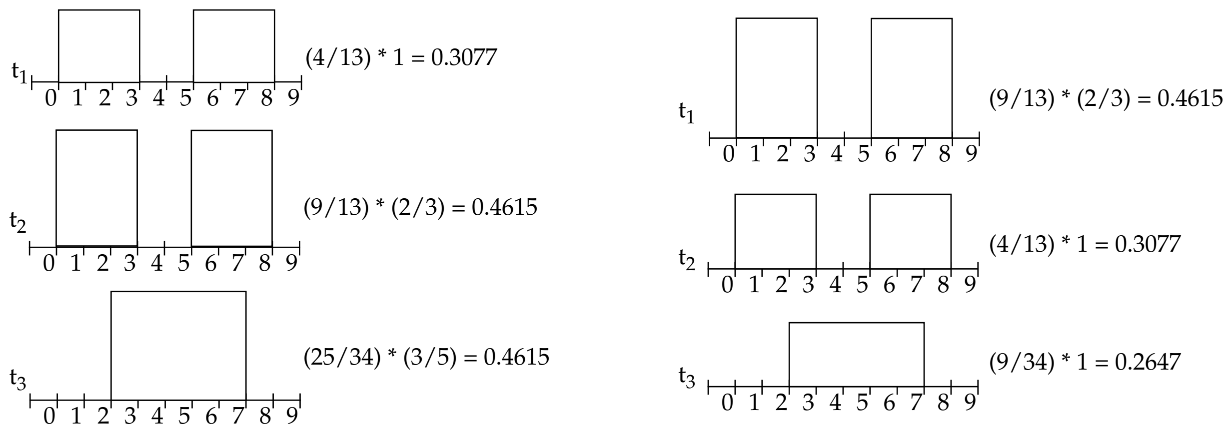

Figure 1: Distribution graph of each task on P1 (left)and P2 (right).

	P1	P2	P1	P2
t1	4/13	9/13	0	1
t2	9/13	4/13	9/13	4/13
t3	25/34	9/34	25/34	9/34

Table 5: Probability table for tentative assignment.

multiplied by their respective distribution over the time frame during which they can be executed.

Clearly, it is not possible to assign all the tasks on a 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 onto all the processors according to probabilities based on execution times of tasks. At a glance, it is not a good idea to put task 1 and 2 on the same processor since there could be no feasible schedule if the partitioning is done in that way.

When we attempt to assign a task on a processor, the self-force is computed as follows: $self-force\,(task\ \ i,\ processor\ \ j)\ = \sum_{j}\sum_{k} DG\,[k, j] * X\,[i]$

For example, self-force of task 1 on P2 = 0.7693 * (4/13) * (2/3) * 3 + 1.0339 * (4/13) * (2/3) * 3 - 0.7692 * (4/13) * 1 * 3 - 1.2104 * (4/13) * 1 * 3 = -0.7178. Of the values of the self-force, this one is the minimum. This can be interpreted as assigning the task 1 onto 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 the task 1 on the processor 2 we get the probability table given in the Table 5. 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 the task 2 on the processor 1 and finally the task 3 on the processor 1.

Assignment can be summarized using the following pseudo-code:

```
Assignment ()
repeat
  for all the tasks
    for all the processors
      compute self force;
    endfor;
  endfor;
  pick a combination of a task and a processor with least self force;
until all the tasks are assigned;
```

7.0 Task-Level Scheduling

The basis for our scheduling algorithm is EDF scheduling policy [Sta95]. Clearly, the EDF is not optimal in our situation. We turn our attention to a heuristic procedure 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 execution of one or more tasks, a feasible schedule can be identified when the EDF without transformations cannot find one. This situation arises when a task has an earlier deadline than a task started earlier than the task. For example, task A started at time 0 and has to be finished before time 5 (deadline) and its execution time is 3. Task B is released at time 1, is due at time 3, and its execution time is 1. The schedule by EDF is not feasible. But if the execution of task A is delayed until the arrival of task B, then EDF finds a feasible schedule.

Our scheduling algorithm is based on the following three, easy to prove, observations:

#1. Any sequence is optimal if all the tasks have the same start time and the same deadline.

#2. The EDF is optimal if all the tasks have the same deadline and different start times.

#3. 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).

From the observations made above, we know that if a set of tasks does not have the characteristics described in the observations, there could be a feasible schedule even when the EDF fails to find one. Our algorithm addresses the situations where the EDF is not optimal. It is based on the following facts. 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. If a task misses its deadline when the EDF is used, then it would be beneficial to delay one or more tasks executed prior to the task missing its deadline. Therefore, 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 means the task that our algorithm will make meet its deadline. When a candidate is selected to be delayed, we want to make sure that the number of unused time slots and overlaps are minimized. As 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. For example, in Figure 3, tasks that started (released) before the deadline of task 1 (which is the first of any deadline) are taken to be re-scheduled. Basically, the algorithm tries to find a way to make one task meet one deadline at a time by delaying execution of other tasks given that those tasks can be delayed.

Figure 3 depicts an instance of scheduling problem. The rectangles refer to respective deadlines and start times of the tasks. The numbers inside the boxes indicate the execution times over available times. The EDF does not work for this case although there exist feasible schedules. But if task 3 yields its execution turn in favor of meeting the deadline of the task 1 (or task 2), then there is a feasible schedule and the EDF can find it. Note that in order to

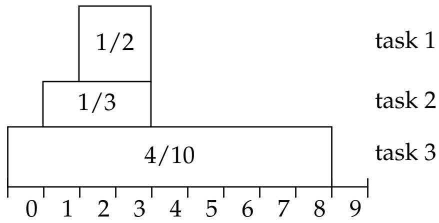

Figure 2: (a) Example #1: Distribution graphs of tasks to be scheduled

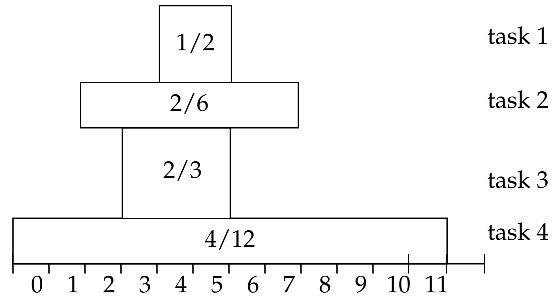

Figure 2: (b) Example #2: Distribution graphs of tasks to be scheduled

have a feasible schedule we must not use the first time slot. The schedule, however, is optimal.

The example shown in Figure 2b cannot be solved using EDF either. But if at most two tasks yield their execution turns (that are based on the start times), all task can meet their deadlines. When the EDF is used without modifications, the schedule would be task 4 →task 3→task 1 → .., which is not feasible. Note that there is only one task the start time of which can be delayed, namely, task 4. After changing the start time of task 4 to the start time of task 3 or task 1, the EDF generates the schedule task 2→task 3→task 1 → .. Again it is not feasible. By moving the start time of task 2 to the start time of task 3 or task 1, the EDF finds a feasible schedule.

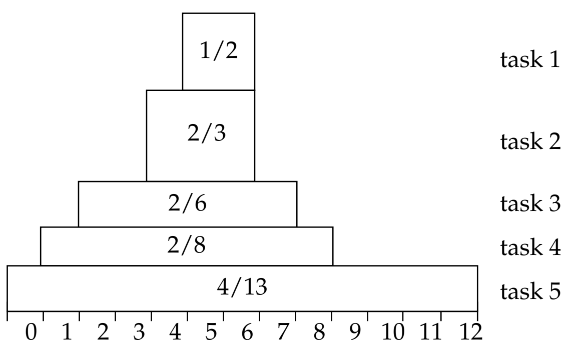

Figure 3: (a) An example of scheduling problem requiring the application of force-directed selection of a task to be delayed

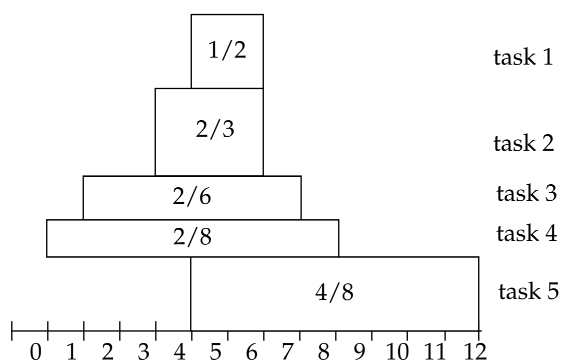

Figure 3: (b) The scheduling problem after one application of delaying a task.

Next consider example shown in Figure 3. The schedule by the EDF without transformations would be task 5→task 2→task 1 → ..,. There is no feasible schedule. By delaying the start time of task 5, the modified EDF finds a new sequence task 4 →task

3→task 2→task 1 → .. In this case there are more than one tasks that can be delayed (task 4 and 3). The choice for delaying its start time will impact the feasibility and effectiveness of the procedure. For example, if task 4 is chosen, the EDF cannot find a feasible schedule. On the other hand, if task 3 is chosen, the schedule length might be longer because the processor will be idle at time 1.

The first example shows that we can improve our chance of having a feasible schedule by checking to see if any task misses its deadline if a task which arrived prior to the task missing its deadline executes before the task. We do not check all the combinations of start time changes here. In the example, if task 3 executes according to its start time, then task 1 misses its deadline. So, it is our advantage to move the start time of task 3 to that of task 1 so that task 1 can go first. The second example, however, is more complex. No task misses its deadline if one of the prior task executes first. The third example is even more complex. To address it we propose force-directed delay-based EDF that is the described in the rest of this section.

In general, the algorithm first tries the EDF to find a feasible schedule. If it cannot, then it checks the given set of tasks to see if the EDF is optimal using the criteria given in 1-3. If it is the case, there is no a feasible schedule. Otherwise, it selects a task at a time to be delayed using modified force-directed scheduling [Pau89].

7.1 Distribution Graph, Force, and Self-force

The distribution is defined as the 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, 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 distribution graphs. The resulting distribution graphs in our problem indicate the demand at a time slot for the resource requested by all the tasks. The DG at time 2 in Figure 2a is 2/5 + 1/3 + 1/2 = 37/30, which means at least one task must be able to be scheduled not claiming the time slot in order for us to have a feasible schedule. Interestingly, if task 3 is scheduled solely based on the start time of it, it must use the time slot 2 and there is no feasible schedule for the task set. The distribution graphs are given by $\sum_i (E[i][j]/A[i])$

Each task has a self force associated with each time slot of its time frame which reflects the effect of an attempted delay of a start time of a task on the overall demand requested by all the tasks given by *force = DG[i] * x[i]* where *DG[i]* is the current distribution value and *x[i]* is the change in the operation's probability. Note that in our scheduling problem, we want to make sure that at each scheduling step every possible time slot at the origin side of time axis be used while maximizing the schedulability. Therefore, unlike the self force described in [Pau89], self force in our problem will have positive value when DG is lower than 1.0 at the origin side of the time axis. We want to 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.

Finally, the self-force associated with delaying the start time of a task is $\sum_i force(i)$ for all affected time slots *i*.

To illustrate the application of the self force to choose a task to be delayed, we use example 3. We first obtain the distribution graphs in Figure 6. We attempt to delay the start times of task 3 and 4.

self-force(3) = $\sum_i force(i)$, i = 2, 3, ..., 7 and self-force(4) = $\sum_i force(i)$, i = 1, 2, ..., 8

self-force(3) = (7/12) * (1/3) * 2 - (15/12) * (3/12) + (1) * (1/12) + (27/12) * (1/3) * 2 + (13/12) * (1/3) = 2.0208

self-force(4) = 0.25 * 0.25 + (7/12) * 0.25 * 2 - (15/12) * (3/12) + (27/12) * (1/4) * 2 + (13/12) * (1/4) + (3/4) * (1/4) = 1.625

From the two resulting numbers of the self-forces, we see that changing the start time of task

4 has less overall adverse effect on the schedulability. In fact, changing the start time of task 3 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: task 3 -> task 2 -> task 1 -> task 4 -> task 5. Next scheduling will be continued at the time slot 14.

This procedure checks to see if there is a feasible schedule for the given set of tasks along the time axis starting from the origin of the time axis.

1. Identify the set of released tasks. That is, look for the set of tasks that are arrived before the first deadline of a task in the set. The set of tasks released prior to the first deadline is considered at a time.
2. Try EDF for the task set. If there is no feasible schedule, compute the self-force for each task that is released prior to the deadline and have a deadline later than that of the target.
3. Select a task that will be scheduled after the target task based on the values of the self-force found in #2. Adjust the start time of the picked task to the start time of the target task.
4. Repeat #2-#3 until a feasible schedule for the set is found or there is no more possible candidate that can be delayed.
5. When a feasible schedule for the set of tasks are obtained, compute the schedule length and adjust start times of subsequent tasks. Repeat the procedure from this point of time on until done or found there is no feasible schedule.

8.0 Experimental results

In order to evaluate the effectiveness of the proposed approach and the optimization algorithms we tested out implementation of the synthesis system for hard real-time systems on a number of examples. The tasks are either randomly generated or industrial DSP, communication, and control examples adopted from [Pot95]. Table 6 shows one of our experimental date, task, sets. The first four columns show run time of a given task on a 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. The Table 7 shows the execution time of each task on each processor. The tasks have associated timing constraints as shown in the Table 6. Each processor has an associated cost given in the Table 7.

	P1	P2	P3	P4	T	S	D	A
t1	1	2	5	3	20	3	12	10
t2	2	2	6	4	20	4	15	12
t3	2	4	5	7	20	7	17	11
t4	2	5	9	6	20	1	14	14
t5	4	5	3	9	20	0	18	19
t6	5	7	7	6	40	17	28	12
t7	4	7	9	5	40	4	23	20
t8	7	5	9	7	40	15	27	13
t9	2	5	8	5	40	10	30	21
t10	6	8	7	8	40	0	30	31
t11	3	8	10	10	40	3	35	33
t12	6	6	12	10	50	15	40	26
t13	2	7	10	8	50	0	35	36
t14	4	3	12	10	50	9	25	17
t15	5	6	14	10	50	0	40	41
t16	3	8	9	7	50	7	45	39
t17	7	9	10	10	50	30	40	11
t18	6	10	12	10	60	20	45	26
t19	8	5	10	6	60	15	55	41
t20	4	8	7	7	60	0	45	46

Table 6: Execution time table; T: period, S: start time (release time), D: deadline, A: available time

P1	P2	P3	P4
40	30	15	20

Table 7: Cost vector

LCM	cost lower bound	allocated cost	processor allocated	running time
600	53.87	95.00	3 (P1, P1, P3)	2 seconds

Table 8: Results

P1	P1	P3	Overall
0.76	0.82	0.49	0.69

Table 9: Utilization

P1: 1, 2, 8, 11, 13, 14, 17, 18;	P2: 3, 4, 6, 7, 9, 12, 15, 16, 20	P3:1, 5, 10, 19

Table 10: Final ask assignment

Note that the utilization of each processor is remarkably high considering we are dealing with non-preemptive tasks with arbitrary release times and deadlines. [Liu73, Leh89, Pat90].

Table 11 shows improvement of new approach over one where the best random resource allocation is used. For the generation of the best random solution 100 trials are conducted. The utilization of resources are high, improvements over the best random solution are always by at least 25% and indicate high quality results.

Number of Tasks	32	34	40	44	46	48
Best Random Solution	580	580	745	720	830	830
Optimized Solution	430	435	445	530	525	610
Cost reduction (%)	25.9	25	40.3	27.4	36.8	26.5
Resource Utilization	0.692	0.634	0.575	0.676	0.572	0.607

Table 11: Experimental Results

Table 11 shows improvement of new approach over one where the best random resource allocation is used.

9.0 Related Research

The directly related research are ones conducted in behavioral and system level synthesis, hard real-time scheduling, and search and heuristic optimization techniques.

A good introduction and review of the early work on behavioral synthesis are given in several papers and books [McF90, Gaj92]. System level synthesis, including hardware-software codesign, also are premier design and CAD research topics [Wol94]. The most relevant system research subdomain is hardware-software partitioning, where a great variety of techniques have been proposed [Bar94, Gaj94, Gup93, Ism94, Gaj96].

Hard-real time scheduling efforts has been an important topic of research for three decades. The early work on scheduling of a set of periodic tasks with strict timing constraints on periodicity, arrival and required time of each task, culminated in a classic rate-monotonic scheduling algorithm [Liu73]. Consequently, rate-monotonic scheduling has been extensively analyzed and generalized, mainly by researchers at Carnegie-Mellon university [Leh89, Sha90a, Sha94]. The most notable practical application of real-time scheduling approaches, an 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 [IEE91, IEE93, Sha90b], and use of the generalized rate-monotonic scheduling techniques in several major advance-technology projects such as Space Station Program and the European Space Agency on-board operating system [Sha94]. The strong endorsement of several research and development groups of the earliest-deadline-first and rate-monotonic scheduling as most suitable resource allocation policies for continuous media servers [Nag92, Ram94, Ste95] and ATM switch scheduling [Swa93] 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 [Rus95]. 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. Force-directed heuristics have been widely used at the several levels of abstraction in the design process. The approach was pioneered by Soukup [Sou81], who used it as a core optimization routine in the epitaxial growth algorithm for IC and board constructive placement. Paulin and Knight [Pau89] developed a force-directed approach for data-flow graph scheduling, which due to its clear intuitive foundations and strong performances have been very used by many behavioral synthesis schedulers [McF90, Wal91].

10.0 Conclusions

An approach for synthesis of application specific systems which implement a set of hard real-time tasks using general purpose processors is presented. The approach has three steps: A* search based resource allocation, force-directed assignment, and earliest-deadline first-based scheduling. The experimental results show the high effectiveness of the approach.

11.0 References

[Bar94] E. Barros, W. Rosenstiel, and X. Xiong, "A method for partitioning UNITY language in hardware and software," in *Proceedings, EuroDAC '94*, pp. 220-225, IEEE Computer Society Press, 1994.

[Dec85] R. Dechter, J. Pearl, "Generalized best-first strategies and the optimality of A*", *Journal of the ACM*, Vol. 32, No. 3, pp. 505-536, 1985.

[Gaj92] D. D. Gajski, et al: "*High-level synthesis: introduction to chip and system design*", Kluwer Academic, Boston, 1992.

[Gaj94] D. D. Gajski, F. Vahid, and S. Narayan, "A System-Design Methodology: Executable Specification Refinement," in *Proceedings, Euro-DAC '94*, pp. 458-463, IEEE Computer Society Press, 1994.

[Gaj96] D.D. Gajski, S. Narayan, L. Ramachandran, F. Vahid and others: "System design methodologies: aiming at the 100 h design cycle", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol.4, No.1, pp. 70-82, March 1996.

[Gar79] M. R. Garey, D. S. Johnson, "*Computers and Intractability: A Guide to the Theory of NP-Completeness*", W. H. Freeman and Company, New York, 1979.

[Gup93] R. K. Gupta and G. De Micheli, "Hardware-software cosynthesis for digital systems," *IEEE Design & Test of Computers*, Vol. 10, No. 3, pp. 29-41, 1993.

[IEE91] IEEE, *"Real-Time extensions to POSIX"*, IEEE Std. P. 1003.4, IEEE, New York, NY, 1991.

[IEE93] IEEE, *"Futurebus+ Recommended Practice"*, IEEE Std. 896.3, IEEE, New York, NY, 1993.

[Ism94] T. B. Ismail, K. O'Brien, and A. Jerraya, "Interactive System-Level Partitioning with PARTIF," n *Proceedings, Euro-DAC '94*, pp. 464-468, 1994.

[Law81] E.L. Lawler, C.U. Martel, "Scheduling periodically occurring tasks on multiple processors", Information Processing Letters, Vol. 12, No. 1, pp. 9-12, 1982..

[Leh89] J.P. Lehoczky, L. Sha, Y. Ding, "The Rate Monotonic Scheduling Algorithms - Exact Characterization and Average Case Behavior" *IEEE Real-Time System Symp.*, pp. 181-191, 1986.

[Lee95] E.A. Lee, T.M. Parks, "Dataflow Process Networks", *Proc. of the IEEE,* Vol. 83, No. 5, pp. 773-799, 1995.

[Liu73] C.L. Liu, J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment", *Journal of ACM*, Vol. 20, No. 1, pp. 46-61, 1973.

[McF90] M.C. McFarland, A.C. Parker, R. Camposano: "The High-Level Synthesis of Digital Systems", *Proceedings of the IEEE*, Vol. 78, No. 2, pp. 301-317, February 1990.

[Nag92] R. Nagarajan, C. Vogt, "Guaranteed Performance of Multimedia Traffic over the Token Ring", Tech. Report No. 439210, IBM-ENC, Heidelberg, Germany, 1992.

[Pat90] D.A. Patterson, J.L. Hennessy, *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[Pau89] P.G. Paulin, J.P. Knight, "Force-directed scheduling for the behavioral synthesis of ASICS", *IEEE Transactions on CAD,* Vol. 8, No. 6, pp. 661-679, June 1989.

[Pot95] M. Potkonjak, W.H. Wolf, "Cost Optimization in ASIC implementation of Periodic Hard-Real Time Systems using Behavioral Synthesis Techniques", *ICCAD95 International Conference on Computer-Aided Design*, pp. 446-451, 1995.

[Pra94] S. Prakash, A. C. Parker: "Synthesis of application-specific multiprocessor systems including memory components", *Journal of VLSI Signal Processing*, Vol.8, No.2, pp. 97-116, Oct. 1994

[Ram94] K. Ramamritham, J.A. Stankovic: "Scheduling Algorithms and Operating System Support for Real-Time Systems", *Proc. of the IEEE*, Vol. 82, No. 1, pp. 55-67, January 1994.

[Rus95] S. Russel, P. Norvig, "*Artificial Intelligence: A Modern Approach*", Prentice-Hall, Englewood Cliffs, NJ, 1995.

[Sha90a] L. Sha, J.B. Goodenough: "Real-Time Scheduling Theory and Ada", *IEEE Computer*, Vol. 23, No. 4, pp. 53-62, April 1990.

[Sha90b] L. Sha, R. Rajkumar, J. Lehoczky, "Real Time Scheduling support in Futurebus+", *IEEE 11th Real-Time Systems Symposium*, pp. 331-340, 1990.

[Sha94] L. Sha, R. Rajkumar, S.S. Sathaye: "Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems", *Proc. of the IEEE*, Vol. 82, No. 1, pp. 68-82, January

1994.

[Sou81] J. Soukup, "Circuits Layout", *Proc. of the IEEE*, Vol. 69, No. 10, pp. 1281-1304, October 1981.

[Sta95] J.A. Stankovic, M. Spuri, M. Di Natale, G.C. Buttazzo, "Implications of Classical Scheduling Results for Real-Time Systems", *IEEE Computer*, Vol. 28, No. 6, pp. 16-25, June 1995.

[Ste95] R. Steinmetz, "Analyzing the Multimedia Operating Systems", *IEEE Multimedia,* Vol. 2, No. 1, pp. 68-84, 1995.

[Wal91] R.A. Walker, R. Camposano: "*A Survey of high-level synthesis systems*", Kluwer Academic, Norwell, MA, 1991.

[Wol94] W.H. Wolf: "Hardware-Software Co-Design of Embedded Systems", *Proc. of IEEE,* Vol. 82, No. 7, pp. 967-989, 1994.

[Yen95] T.-Y. Yen, W. Wolf, "Communication Synthesis for Distributed Embedded Systems", *ICCAD95 International Conference on Computer-Aided Design*, pp. 288-294, November 1995.