Distributed Synthesis of Real-Time Computer Systems

Ahmad Abualsamid Raed Alqadi Parameswaran Ramanathan

Department of Electrical and Computer Engineering
University of Wisconsin–Madison
Madison, WI 53706–1691.
parmesh@ece.wisc.edu, (608) 263–0557

Abstract

High-level synthesis has become commonplace in many areas of computing such as VLSI design and digital signal processing. However, it is just beginning to receive attention in the area of real-time systems. Given a real-time application and a design library of components, high-level synthesis involves three main steps: (i) estimation of processors and resources required to meet the constraints of the application, (ii) identifying suitable architectures using the components from the design library, and (iii) scheduling application tasks on the selected architecture. In this paper, we focus on the first and the third steps of this process. Specifically, we identify key issues in parallelizing these two steps. We then discuss approaches to deal with these issues and present results of our distributed implementation. The results of this implementation on a network of workstations show that considerable speedup in overall runtimes can be achieved by using multiple workstations.

1 Introduction

Tasks in a real-time application usually have deadline constraints by which they must complete their execution. Failure to complete a computation within a task's deadline may lead to a catastrophe. Examples of such applications include flight-control systems, life-support systems, nuclear power-plants and process-control systems. Due to the severity of the consequences, a distributed computing system is often dedicated to the tasks in a real-time application. This system must be carefully designed to ensure that all the constraints of the application are satisfied. However, due to the large number of constraints involved and due to the numerous design alternatives, designing such a computer system is a very difficult problem. This problem can be alleviated with the help of computer-aided synthesis (CAS) tools which facilitate the search and evaluation of design alternatives. This approach has been successful in other areas of computing such as VLSI design and digital signal processing. However, the approach is just beginning to receive attention in the area of real-time systems [2, 7, 9].

The main issue in the design of CAS tools is their runtime complexity. Due to the large search space, the time required for synthesis may be unacceptably large for use in "real" applications. To overcome this problem, we have been developing algorithms/heuristics which run in a distributed fashion on a network of workstations and/or parallel computing systems. In this paper, we describe our techniques for such a distributed synthesis of computer systems for real-time applications. We have implemented two key steps in the synthesis process, namely lower bound analysis and scheduling, on a network of workstations. The lower bound analysis step estimates the number of processors and resources required to meet the constraints of the application while the scheduling step verifies whether the application constraints are satisfied on a candidate architecture. The implementation is done using the Parallel Virtual Machine (PVM) message passing system [6]. The results of this implementation demonstrate that the use of a distributed system for computer-aided synthesis has a tremendous potential.

The rest of this paper is organized as follows. Section 2 contains an overview of the synthesis process. Then, Section 3 identifies some of the issues in parallelizing the lower bound analysis step of the synthesis process and then discusses solutions to deal with them. Section 4 similarly deals with the scheduling step of the synthesis process. The results of an empirical evaluation of our distributed implementation are presented in Section 5. The paper concludes with Section 6.

2 Overview of the synthesis tool

The inputs to our synthesis system are a design library of components, and the mission-critical applica-

The work reported here is supported in part by the National Science Foundation grant MIP-9213716.

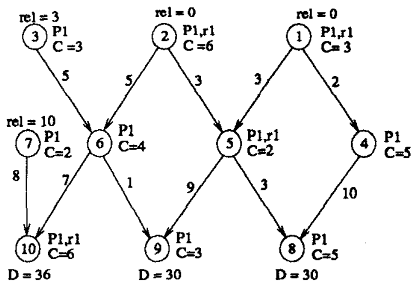


Figure 1: An example real-time application.

tion. The output includes a heterogeneous architecture containing components from the design library, a mapping of the tasks in the application onto the modules in the architecture, and a schedule for the tasks and messages in the application. The design library contains processors, resources, and interconnects from which the distributed system is to be synthesized. Components in the design library are characterized by attributes such as performance and cost. The application is specified as a set of cooperating tasks and the constraints that each task must satisfy, e.g., a task may have deadline constraints, resource requirements, precedence relations, computational needs, etc. The objective is to identify the lowest cost computer system (constructed using the components in the design library) which satisfies all the constraints of the application.

For example, Figure 1 shows a simple application with 10 non-preemptive tasks. The tasks are numbered from 1 to 10. Each task is annotated with its computation time and its resource requirements (e.g., task 10 requires 6 units of computation on processor of type P1 and resource r1). Tasks 1, 2, 3, and 7 are also annotated with their release times. Tasks 8–10 are annotated with their deadlines. The precedence relationships between the tasks is shown by a directed arrow (e.g., tasks 1 and 2 must complete their execution before task 5 can begin its execution). After a task completes its execution it sends information to its successors. A task must receive this information from its predecessors before it can begin its execution. If a task and its successor are assigned to two different processors, then the information is sent in the form of a message. The size of the message is indicated alongside the directed arrow.

For this application, the design library must contain several copies of a processor of type P1 and a resource of type r1. The synthesis system will select appropriate number of copies of processor P1 and resource r1 and determine an interconnection structure between them. In this paper, we assume that the selected processors are interconnected through a set of shared buses. Each processor can send or receive on all the buses in the system. Furthermore, we assume that there is a separate interconnection between the processors and the resources. The design of that communication network is not considered in this paper, i.e., we assume that the communication between a processor and a resource takes place instantaneously and there is no need to schedule those communications.

Given a real-time application and a design library of resources, the key steps in the synthesis process are:

1. Lower Bound Analysis: Compute a lower bound on the number of processors, resources, and interconnects required to meet the constraints of the application.

2. Module Selection:

   (a) Choose processors and resources from the design library for inclusion in the architecture.

   (b) Identify a suitable interconnection structure between the selected components.

3. Scheduling: Evaluate the architecture identified in Step 2 by scheduling the application tasks and the resulting messages on the architecture.

4. Terminate if a satisfactory solution is found. Otherwise, go to Step 2 and improve the architecture based on the results from Step 3.

The lower bound analysis step is considerably longer than the other steps, thus it is essential to parallelize it. Steps 2 and 3 will be invoked many times during the synthesis process, thus they also need to be parallelized.

3 Distributed Lower Bound Analysis

Fernandez and Bussell [4] did one of the earliest work on determining a lower bound for the number of processors required to schedule a given application within its critical time. They considered applications in which the tasks have integer execution times and precedence relationships, but, zero communication time with other tasks. Al-Mohummed [1] extended the algorithm in [4] to applications in which the communication requirements between the tasks is non-zero. However, Al-Mohummed's work did not deal with many of the constraints commonly found in real-time applications. Just recently, Alqadi and Ramanathan extended the algorithm in [1] to deal with

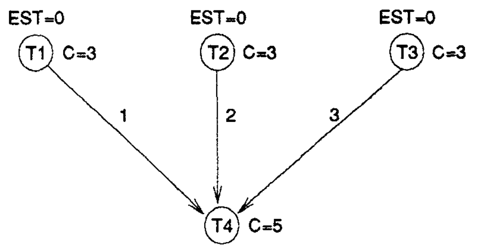


Figure 2: An example application for illustration of EST analysis.

release time, deadline, and resource constraints [3]. The implementation in this paper is based on the theoretical results in [3]. However, several issues had to be resolved in converting the results to an efficient implementation. Before describing these issues, we briefly review the results from [3].

3.1 Informal Overview of Lower Bound Analysis

The lower bound analysis in [3] has four basic steps:

1. EST Analysis: Compute the earliest start time (EST) of each task in the application.
2. LCT Analysis: Compute the latest completion time (LCT) of each task in the application.
3. Partitioning: Partition the application tasks into a sequence of smaller subsets such that each subset can be treated independently with respect to lower bound analysis.
4. Lower Bound Computation: Compute a lower bound on the number of units of each processor/resource required by the application.

EST Analysis. The EST analysis starts with tasks that have no predecessors and recursively proceeds towards tasks which have no successors. It evaluates the EST of a task only after computing the EST of all its predecessors. If there are no communication requirements, then the evaluation of the EST of a task is simple, given the EST of its predecessors. However, if there are communication requirements, then the evaluation of EST is more involved. The basic idea of this analysis is best illustrated by a simple example. Consider the evaluation of the EST of task T4 in Figure 2. In this example, T4 has three predecessors, T1, T2, and T3. The computation times of T1, T2, and T3 are 3 units each. The computation time of T4 is 5 units. Furthermore, after completion of T1, T2, and T3, the time required to send messages to T4 (if they are assigned to a different processor) are 1, 2, and 3 units, respectively. Also, suppose that the EST of T1, T2, and T3 are 0.

If T4 is assigned to a processor different from those of T1, T2, and T3, then T4 cannot start before time 6 since it must receive a message from T3. However, if T3 and T4 are assigned to the same processor which is different from that of T1 and T2, then T4 cannot start earlier than 5. This is because there is no need to send a message from T3 to T4 and T4 must wait for a message from T2. Now, if T2, T3, and T4 are assigned together and T1 is at a different processor, then T4 cannot start earlier than time 6, because T2 and T3 must complete their execution before T4 can begin. That is, the earliest start time of T4 depends on which combination of its predecessors are assigned together with T4. Therefore, to determine the EST of T4, one must find the best subset of its predecessor to be assigned to the same processor as T4. An efficient algorithm to identify such subsets are described in [3]. Using this algorithm, the EST of all tasks in an application can be found in $O(E)$ time where E is the total number of direct precedence relationships between the tasks.

LCT Analysis. The algorithm for LCT analysis is similar to that of EST analysis. The LCT of a task is evaluated only after the LCTs of its successors. As in the case of EST analysis, the LCT of all tasks in the application can be found using an $O(E)$ algorithm.

Partitioning. Using the EST and LCT of the tasks, the third step partitions the application with respect to each resource. The partitioning is done to reduce the runtime complexity of the next step. Basically, let r be a resource used by the application and let ST_r be the set of tasks which require resource r. ST_r is then partitioned into subsets P_{r1}, P_{r2}, …, P_{rm} such that the LCT of every task in P_{rl} is less than or equal to the EST of every task in any P_{rk}, for all l, k, $k > l$. It is then proved in [3] that each of these partitions can be treated independently while computing a lower bound for resource r.

Lower Bound Computation. Consider a partition P_{rl} and let EST_l be the smallest EST among all tasks in P_{rl}. Likewise, let LCT_l be the largest LCT among all tasks in P_{rl}. Now consider an interval $[t_1, t_2]$ such that $\text{EST}_l \leq t_1 < t_2 \leq \text{LCT}_l$. Let $\Psi_r(\tau, t_1, t_2)$ be the minimum computation time which must be completed by task τ in the interval $[t_1, t_2]$ on resource r in order for all tasks to complete by their respective LCTs. Let ST_r be the set of tasks which require resource r. Then $\Theta_r(t_1, t_2) = \sum_{\tau \in ST_r} \Psi_r(\tau, t_1, t_2)$ is the total minimum computation time which must be completed on resource r in $[t_1, t_2]$. Therefore, in order to ensure that all tasks complete by their respective LCTs, we need at least $\left\lceil \frac{\Theta_r(t_1, t_2)}{t_2 - t_1} \right\rceil$ copies of resource r. That is,

$$\mathrm{LB}_r = \max_{1 \leq l \leq m} \left\{ \max_{\mathrm{EST}_l \leq t_1 < t_2 \leq \mathrm{LCT}_l} \left\lceil \frac{\Theta_r(t_1, t_2)}{t_2 - t_1} \right\rceil \right\}. \tag{3.1}$$

The correctness of the above equation is proved in [3]. In this equation, the outer maximum is over all partitions and for each partition P_{rl}, the inner maximum is taken over all possible intervals $[t_1, t_2] \subseteq [\mathrm{EST}_l, \mathrm{LCT}_l]$.

3.2 Implementation Issues

There are two issues which have to be resolved in converting the theoretical results in the previous section to an efficient distributed implementation. These two issues are: (i) selection of intervals, and (ii) the parallelism approach. The first issue arises because for each partition the maximum operation is to be taken over uncountably many intervals. As a result, a direct implementation of Equation 3.1 is computationally intractable. On the other hand, if the maximum operation is taken over fewer intervals, then the resulting lower bound may be smaller (hence, weaker) than the lower bound as given by Equation 3.1. The challenge is to select a suitable set of intervals such that the resulting bound in reasonably close to the bound given by Equation 3.1 while being computationally tractable.

The second issue arises because the way in which the workload is distributed affects the runtime efficiency of the distributed implementation. In the following two subsections we discuss these two issues in more detail and present our approach for tackling them.

3.2.1 Selection of intervals

If all the computation and the communication times in the application are integers, then the bound in Equation 3.1 can be computed by considering all possible integer intervals $[a, b] \subseteq [0, D_{max}]$, where D_{max} is the largest deadline in the application. This idea was suggested by Fernandez and Bussell [4], although their work did not deal with many of the constraints found in real-time applications. Since there are $O(D_{max}^2)$ such intervals and since D_{max} is often quite large, the runtime complexity of this approach may be unacceptable for many real-time applications.

An alternate approach is to only consider all intervals of the form $[e_\tau, l_\tau]$, where τ is a task in the application and e_τ (l_τ) is the EST (LCT) of task τ. This approach was suggested in [1]. The advantage is that there are exactly N intervals, where N is the total number of tasks in the application. However, our experience indicates that using only N intervals gives a weak lower bound.

Our approach is to choose a set of random intervals within each partition. For each partition, the total number of intervals considered is equal to $R \cdot N$, where R is a design parameter and N is the number of tasks in the application. In our implementation, $R = 40$.

3.2.2 Parallelism Issue

There are two possible ways of parallelizing the lower bound analysis, Application Parallel and Computation Parallel. In the discussion below, we describe these two ways and argue that Computation Parallel approach is better than the Application Parallel approach for lower bound analysis.

Application Parallel Approach. In this approach, the computational load is distributed among the workstations by partitioning the application and assigning a subset of tasks to each workstation. Each workstation then performs all computations in the lower bound analysis for the set of tasks that has been assigned to it. The workload is distributed because each workstation has to deal with only a subset of the tasks in the application.

However, due to the nature of the computations in the lower bound analysis, each workstation will have to communicate extensively with other workstations to perform its computation. For example, consider the computations in the EST analysis. To compute the EST of a task, the corresponding workstation needs the EST of the predecessors of the task. If a predecessor of the task is assigned to a different workstation, then its EST value must be obtained from the other workstation, i.e., a communication overhead is incurred in the EST analysis. A similar type of communication overhead is necessary for the LCT analysis.

There is also a need for communication in the final lower bound computation step, and when performing the max operations.

Computation Parallel Approach. In this approach, each workstation has the entire application. However, for each task, it performs only part of the computations needed in the lower bound analysis. In particular, each workstation independently computes the EST and the LCT of all tasks in the application. Each workstation then independently identifies the partitions for each resource. The workstations then independently choose to work on mutually disjoint set of intervals, i.e., the workload here is distributed by dividing the set of intervals among the workstations. Each workstation independently computes a lower bound for each resource based on its set of intervals. The workstations perform a reduction operation to compute an overall lower bound for each resource.

The main disadvantage of this approach is that some of the computations are redundantly performed by all workstations. The advantage, of course, is that there is no need to convey this value to other workstations. The only communication occurs at the end; a maximum operation on one value for every resource. Since communication is very expensive in a network of workstations, the reduction in communication is more significant than the increase in computation. Thus this approach works better than the Application Parallel approach.

4 Distributed Static Scheduling

Due to its importance, the scheduling problem has received considerable attention from researchers [8, 11, 12]. For the kind of applications considered here, the problem is known to be NP-complete, but several good heuristics have been proposed [5, 11]. However, to the best of our knowledge, none of the existing work have addressed the problem of distributing a static scheduler to reduce its runtime. Existing work on distributed schedulers usually focus on inserting a dynamically arriving task in the schedule of tasks already present in the system [10]. The incoming tasks are typically assumed to be independent of the tasks already present in the system. The main issue is to determine whether the constraints of an incoming task can be met without jeopardizing the promises made to other tasks.

In contrast, in this paper, we discuss an approach for distributing a static scheduler. A static scheduler has all the necessary information about the tasks it has to schedule. However, the number of tasks is typically fairly large and the tasks are usually not independent of each other. The main issue here is how to distribute and coordinate the computations in a scheduler in order to ensure that all the constraints of the application are satisfied.

As in the case of lower bound analysis, there are two ways of distributing the scheduling workload among a network of workstations. One approach is to partition the set of processors and resources required by the application and assign a subset to each workstation. The workstation is then responsible for scheduling on its subset of processors and resources. Each workstation may have to perform some computation for every task in the application. An alternate approach is to partition the application tasks and assign a subset of tasks to each workstation. A workstation is then responsible for scheduling only the tasks in its assigned subset. However, each workstation may have to schedule on all the processors and resources required by the application.

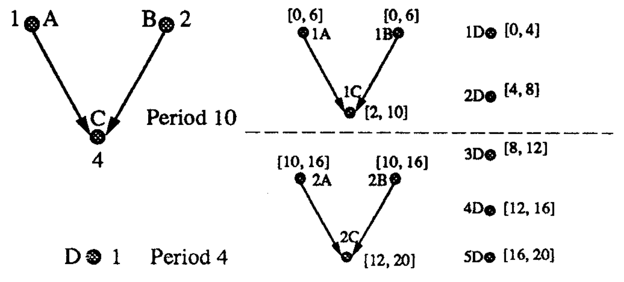
The two approaches differ in the nature and the amount of interaction needed between the workstations. In the first approach, when scheduling a task, the workstations need to know where and when the task's predecessors have been scheduled. To obtain this information, a communication overhead is incurred after scheduling each task in the application. This approach has significant communication overhead and thus lower runtime efficiency. In contrast, as shown later in this section, the amount of communication between the workstations in the second approach can be minimal if the application is partitioned carefully. Consequently, the second approach has better runtimes. We pursue this approach in this paper. We first present the partitioning scheme and then give an overview of the scheduler used in our implementation.

4.1 Partitioning Scheme

Let Γ be the set of tasks in the application and let m be the number of workstations participating in the distributed scheduler. Then, the objective of the partitioning strategy is to identify m disjoint subsets Γ_1, $\ldots$, Γ_m such that: (i) $\cup_{i=1}^{m}\Gamma_i = \Gamma$, and (ii) the LCT of all tasks in Γ_i is less than or equal to the EST of all tasks in Γ_j, for all i, j, $j > i$. The rationale for this objective is that each of these sets can be scheduled independently by a workstation without any communication overhead. This is because the time window in which the tasks in Γ_i must execute is disjoint from the time window for the tasks in Γ_j, $j \neq i$.

However, it may not always be possible to partition Γ in this fashion without some additional constraints. To make this possible, the partitioning strategy imposes additional release time and/or deadlines constraints on some tasks. Theoretically, these additional constraints may make the application impossible to schedule. However, our experience indicates that this is not the case (see Section 5). The question then is how do we select the additional constraints to be imposed on the application?

Our approach for determining the additional constraints is very simple. This simplicity is of at most importance because this partitioning step is an overhead which is not present in a non-distributed scheduler and this overhead affects the runtime efficiency of the distributed implementation. Our approach is to first partition the interval $[0, D_{max}]$ into m equal subintervals, where D_{max} is the largest hard deadline in the application. For tasks whose EST and LCT lie within the same subinterval, we do not impose any additional constraints. For other tasks, we either add a



(a) Periodic jobs (b) Invocations in a superperiod

Figure 3: An example application.

release time greater than the task's EST or a deadline less than its LCT so that its modified EST and LCT lie within one subinterval. Note that, because of the way in which the constraint is generated, any feasible schedule of the modified application is also a feasible schedule of the application. However, as stated earlier, it is theoretically possible for the modified application to be infeasible even though the application is feasible. If this happens, the distributed scheduler will not be able to identify a feasible schedule whereas a non-distributed scheduler may have succeeded.

To illustrate this approach consider the simple application in Figure 3(a). The application has two periodic jobs with periods 4 and 10, respectively. The first job is comprised of three non-preemptive tasks (namely A, B, and C) whereas the second periodic job has only one non-preemptive task (namely D). The precedence relations between the tasks in the first job are shown as directed arrows. The execution time of the tasks is shown as weights near the corresponding vertices. For simplicity, the communication times between the tasks are assumed to be zero. Figure 3(b) shows the invocations of these two jobs in a superperiod (i.e., least common multiple of the periods). The scheduler must find a feasible schedule for all the tasks in this superperiod and repeat the schedule for the subsequent superperiods. The numbers enclosed in [ ] are the EST and the LCT of the corresponding task.

Now suppose that this application is to be scheduled using two workstations. Then, we must partition the tasks in Figure 3(b) into two groups and assign them to the two workstations. To identify this partition, our approach divides the $[0, 20]$ into two subintervals $[0, 10]$ and $[10, 20]$. For all tasks whose EST and LCT lie within one of these intervals no additional constraints are imposed. In this example, this condition is true for all tasks except task 3D. The EST of task 3D lies in $[0, 10]$ whereas its LCT lies in $[10, 20]$. We, therefore, need to impose an additional constraint on this task. Since the subinterval boundary (i.e., 10) is midway between the EST and the LCT of this task, we can add either a release time or a deadline constraint. For example, we can impose a deadline of 10 on this task and thus make its modified LCT equal to 10. With this one additional constraint, tasks 1A, 1B, 1C, 1D, 2D, and 3D will belong to a partition while the remaining tasks will belong to the other partition. One workstation will independently schedule the tasks in the first partition in the interval $[0, 10]$ while the other workstation will schedule the remaining tasks in the interval $[10, 20]$. Since these intervals are disjoint, no communication is required between these workstations after they have identified their respective partitions. In fact, they can each use any appropriate scheduling heuristic to schedule their partition on the processors and resources identified at the end of the lower bound analysis step.

Described below is an informal overview of the scheduler we used in obtaining results presented in Section 5.

4.2 Overview of the scheduler

The scheduler is provided with the number of copies of each resource needed by the application. It is responsible for assigning the tasks to the resources and then determining a start time for each task such that all constraints of the application are satisfied.

The scheduler starts by ordering the tasks in the increasing order of their latest start times[1]. Initially, the latest start times are as obtained during the lower bound analysis. The tasks are considered for scheduling one at a time.

A task is scheduled on the processor on which it can complete the earliest. To identify this earliest completion time, the scheduler first picks the least utilized copy of each required resource. It then considers all possible processor assignments for the task. Each possible processor assignment generates a different set of predecessor messages that have to be scheduled on the communication network. This is because only predecessors which are assigned to a different processor need to send a message through the communication network. For each possible assignment, the scheduler first determines the earliest completion time of all the predecessor messages of the task under consideration. The least schedulable time (i.e., when the processor and the resources under consideration are free to execute the task) after all predecessor messages have ar-

[1]The latest start time of a task is its LCT minus its computation time.

rived plus the corresponding execution time for the task is the earliest completion time of the task on a given processor. The task is scheduled on the processor in which it has the minimum earliest completion time. After a task is scheduled, the ready list is updated to possibly include the immediate successors of the just scheduled task.

The scheduler continues in this fashion until all tasks have been tentatively scheduled. If some tasks do not meet their deadlines, then the whole process is repeated after recomputing the latest start time of all tasks based on the assignment just generated. Note that, a new assignment results in a different communication pattern for some tasks. Consequently, there is a change in the latest completion time of some tasks, which in turn, changes the priority order among the tasks in the next iteration. The scheduler terminates either when a feasible schedule is identified or when a pre-specified iteration limit is exceeded.

5 Evaluation

In this section, we present results of an empirical evaluation of the distributed synthesis tools. The goal of this evaluation is to demonstrate that multiple workstations can be effectively used to synthesize computer systems for real-time applications. To demonstrate this fact, we show that: (i) there is a significant reduction in the overall runtime as a result of using multiple workstations, and (ii) the likelihood of a distributed scheduler being able to find a feasible schedule is comparable to that of a sequential scheduler. It is necessary to demonstrate this second aspect because in distributing the scheduler, additional constraints are imposed on the application. We need to demonstrate that these additional constraints do not have much impact on the likelihood of finding a feasible schedule.

The evaluation is carried out by running the lower bound analysis and the scheduler on several synthetic applications. Each synthetic application is comprised of a number of periodic jobs with different periods, release time and deadline constraints. The periodic jobs have between 5–15 non-preemptive tasks with precedence, resource, and communication constraints. The lower bound and scheduling analysis are performed by considering all the task activations in the interval [0, LCM], where LCM is the least common multiple of the periods of the jobs in that application.

Table 1 shows the summary of the results obtained from our implementation on a network of Hewlett Packard workstations model HP 735. Each workstation has 80 MBytes of memory and runs HPUX 9.0 operating system. The workstations are interconnected

Table 1: Summary of the results from our distributed implementation.

Appln.	Lower	Timing (secs)		
(Size)	Bounds	1 wk	2 wk	4 wk
G1 (106)	(4 1 1 1)	6.6(S)	4.7(S)	3.2(S)
G2 (503)	(5 3 3 3)	142.0(S)	83.9(F)	47.0(F)
G3 (88)	(4 1 1 1)	7.7(F)	4.0(S)	2.9(F)
G4 (214)	(4 2 2 2)	26.5(S)	15.8(S)	10.1(S)
G5 (115)	(2 1 1 1)	8.9(S)	6.0(S)	4.5(S)
G6 (49)	(2 1 1 1)	2.1(S)	1.7(S)	1.3(F)
G7 (61)	(2 1 1 1)	4.1(F)	2.5(F)	1.8(F)
G8 (540)	(5 3 2 3)	159.6(S)	90.9(S)	50.4(S)
G9 (922)	(4 2 2 3)	477.8(S)	254.1(S)	137.7(S)
G10 (71)	(3 1 1 1)	4.1(S)	2.6(S)	2.0(S)
G11 (106)	(4 1 1 1)	9.4(F)	5.3(S)	3.4(F)
G12 (186)	(5 2 2 2)	21.3(S)	13.0(S)	8.3(S)
G13 (91)	(2 1 1 1)	5.4(S)	3.5(S)	2.9(S)
G14 (1220)	(4 3 2 2)	909.6(S)	477.0(S)	241.2(S)
G15 (67)	(2 1 1 1)	1.3(S)	0.9(S)	0.7(S)
G16 (324)	(3 2 2 2)	18.4(S)	10.2(S)	6.2(S)
G17 (277)	(2 2 1 2)	14.0(S)	7.9(S)	4.9(S)
G18 (306)	(2 1 1 1)	19.7(F)	11.2(F)	7.1(F)
G19 (65)	(1 1 1 1)	1.3(F)	0.8(S)	0.6(F)
G20 (215)	(4 2 2 2)	9.1(S)	5.3(S)	3.3(F)
G21 (694)	(4 2 2 3)	83.1(S)	43.7(S)	24.2(S)
G22 (150)	(4 2 2 2)	5.4(S)	3.2(S)	2.2(F)
G23 (315)	(3 2 1 2)	18.7(S)	10.4(S)	6.2(S)
G24 (142)	(3 1 2 1)	4.2(S)	2.7(S)	1.9(S)
G25 (1109)	(3 2 1 1)	198.8(S)	110.3(S)	66.1(S)
G26 (839)	(2 1 2 1)	113.0(S)	62.7(S)	37.5(S)
G27 (2933)	(2 1 1 1)	1594.6(F)	858.9(F)	510.2(F)

using a Fiber Data Distributed Interface (FDDI) network. The distributed implementations are based on the Parallel Virtual Machine (PVM) message passing system version 3.3.5. In this table, the first column contains the number of task instances in the interval [0, LCM] of the corresponding application. The second column contains the results of the lower bound analysis; it shows the lower bound on the number of copies of the processor and resources R1,R2, and R3, respectively. The final three columns show the overall runtimes in seconds for executions with 1, 2, and 4 workstations, respectively. In these results, the scheduler is given two additional copies of the processor than the lower bound given in the second column. Alongside the runtimes, labels S and F are included to indicate whether the scheduler Succeeded or Failed in finding a feasible schedule for the application.

Figure 4 shows the fraction of applications which are successfully scheduled when 1, 2, 3, and 4 workstations are used. The four curves in this figure corre-

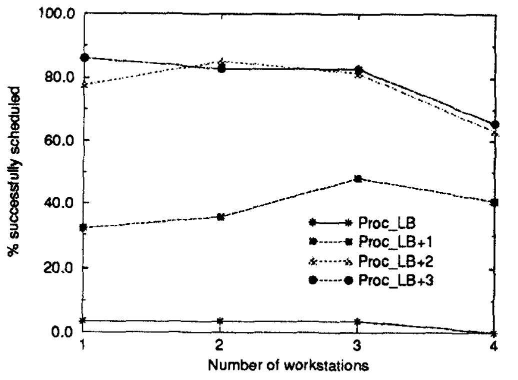


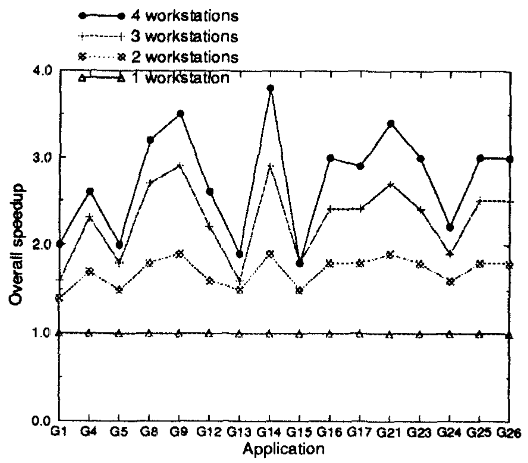
Figure 4: % successfully scheduled applications versus number of workstations.

spond to the number of processors given to the scheduler after the lower bound analysis. For instance, for the curve labeled Proc_LB, the number of processors given to the scheduler is equal to the value computed from the lower bound analysis. Likewise, for the curve labeled Proc_LB+2, the number of processors given to the scheduler is two more than the value computed from the lower bound analysis. Observe that, the percentage of success does not vary significantly with the number of workstations used. This means that the additional constraints imposed on some tasks to reduce the communication time in the distributed implementation (see Section 4) does not significantly affect the likelihood of successfully scheduling an application. Also note that, the percentage of successfully scheduled applications improves in some cases when multiple workstations are used. This is due to the fact the scheduler is based on heuristics.

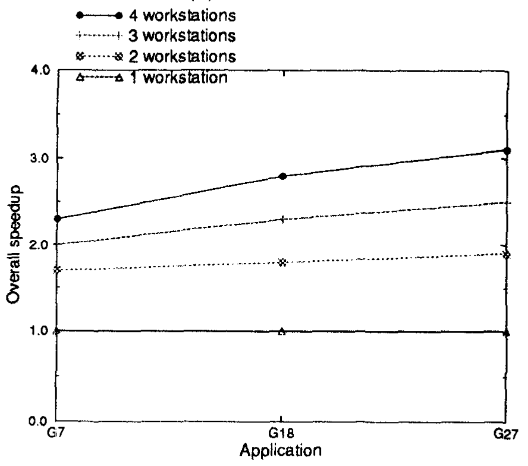
Figures 5, 6, 7 show the various speedups achieved as a result of using multiple workstations for few selected applications. Specifically, Figure 5 shows the overall speedup whereas Figures 6 and 7 show the speedups achieved in the lower bound analysis and the scheduling steps, respectively.

In these figures, the subfigure (a) shows the results only for those applications for which a feasible schedule was found in each of the four executions (i.e., executions with 1, 2, 3, and 4 workstations). Similarly, the subfigure (b) shows the results only for those applications in which no feasible schedule was found in any of the four executions. Also, for the results in these figures, the scheduler is given two more processors than the value computed from lower bound analysis.

It follows from Figure 5, that considerable speedups are achieved in the overall time required for lower



(a) All success



(b) All failure

Figure 5: Speedup in the overall time.

bound and scheduling analysis as a result of using multiple workstations. For example, the speedups when using four workstations range from 2.0 to 4.0. The speedups vary with the applications; larger speedups are achieved in applications with more tasks. This is encouraging because most applications are likely to be much larger than the synthetic ones considered here. By comparing the subfigures and Table 1, we observe that even though the runtimes are much larger when a scheduler fails to find a feasible schedule, the speedups achieved are comparable to those when the scheduler finds a feasible schedule.

By comparing Figures 5 and Figure 6, we observe that speedups achieved in the lower bound analysis are very similar to those corresponding to the overall execution time. This is because 80–90% of the overall execution time is due to the lower bound analysis. Although, scheduling forms a small fraction of the overall time in this paper, it is important to have a very fast scheduling step. This is because in the overall synthesis process, the scheduler will be invoked thousands of times to evaluate different candidate architectures. In this paper, we are presenting results from only one invocation of the scheduler because only one candidate architecture is considered.

In Figure 7, we also notice that the speedups achieved by the scheduler vary more drastically than in the lower bound analysis. Further, we observe that in some cases the speedups are much greater than four even when only four workstations are used. This is because, when multiple workstations are used, we partition the application and impose some additional constraints on few tasks to reduce the amount of communication. As a result, the search space investigated is different for different number of workstations.

6 Conclusions

The paper focused on a distributed implementation of two key steps of a synthesis systems. The first step, namely lower bound analysis, determines a lower bound on the number of processors and resources required to meet the constraints of the application. The second step, namely scheduling analysis, determines where and when the application tasks will execute. We discussed several alternative techniques for parallelizing these two steps. The promising techniques were implemented using the Parallel Virtual Machine message passing system. The results of this implementation were presented in this paper. The results show that the use of a network of workstations for computer-aided synthesis of real-time computer systems is very promising.

Acknowledgements

We would like to recognize the support we received from the Model Advanced Facility at the College of Engineering, University of Wisconsin–Madison.

References

[1] M. A. Al-Mohummed, "Lower bound on the number of processors and time for scheduling precedence graphs with communication costs," *IEEE Transactions on Software Engineering*, vol. 16, no. 12, pp. 1390–1401, December 1990.

[2] R. Alqadi and P. Ramanathan, "Architectural synthesis of mission-critical computing systems," in *Proceedings Complex Systems Engineering Synthesis and Assessment Technology Workshop*, pp. 185–192. Naval Surface Warfare Center, Silver Spring, Maryland, July 1993.

[3] R. Alqadi and P. Ramanathan, "Analysis of resource lower bounds in real-time applications," in *International Conference on Distributed Computing Systems*, May 1995.

[4] E. B. Fernandez and B. Bussell, "Bounds on the number of processors and time for multiprocessor optimal schedules," *IEEE Transactions on Computers*, vol. C-22, no. 8, pp. 745–751, August 1973.

[5] M. R. Garey and D. S. Johnson, "Complexity results for multiprocessor scheduling under resource constraints," *SIAM Journal of Computing*, vol. 4, pp. 397–411, 1975.

[6] A. Geist, A. Baguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderom, *PVM: Parallel Virtual Machine, a user's guide and tutorial for networked parallel computing*, MIT Press, 1994.

[7] S. Howell, C. M. Nguyen, and P. Q. Hwang, "System design structuring and allocation optimization," in *Proceedings of the 1991 Systems Design Synthesis Technology Workshop*, pp. 117–128, September 1991.

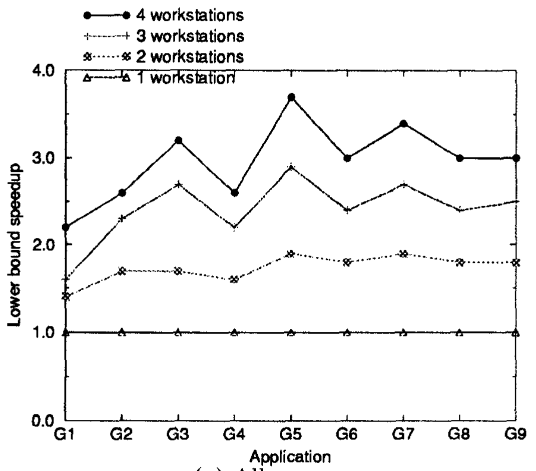
[8] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the Association for Computing Machinery*, vol. 20, no. 1, pp. 46–61, January 1973.

[9] J. W. S. Liu et al., "PERTS: A prototyping environment for real-time systems," Technical report UIUCDCS-R-93-1802, University of Illinois, Urbana, Illinois, May 1993.

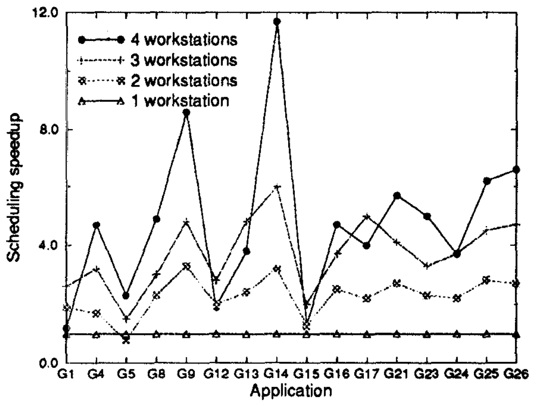
[10] K. Ramamritham and J. A. Stankovic, "Distributed scheduling of tasks with deadlines and resource requirements," *IEEE Transactions on Computers*, vol. 38, no. 8, pp. 1110–1123, August 1989.

[11] K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 55–82, January 1994.

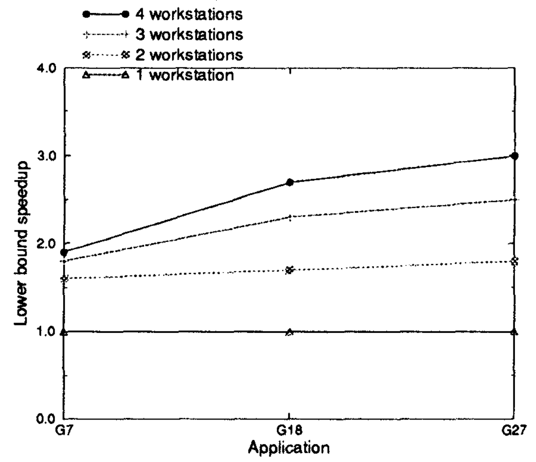
[12] W. Zhao, K. Ramamritham, and J. A. Stankovic, "Scheduling tasks with resource requirements in hard real-time systems," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 5, pp. 564–577, May 1987.



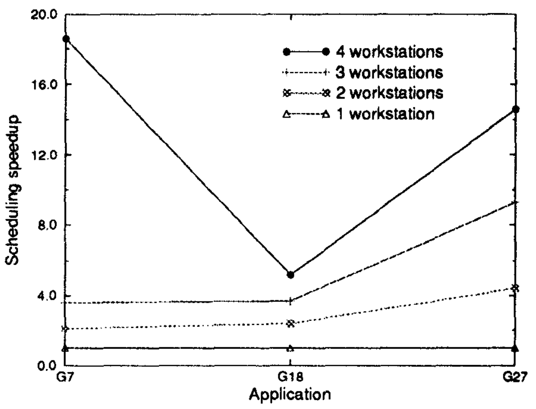
(a) All success



(a) All success



(b) All failure



(b) All failure

Figure 6: Speedup in the time required for the lower bound analysis.

Figure 7: Speedup in the time required for the scheduling.