1996

Real-Time Scheduling for Software Prototyping

Luqi; Shing, M.

Journal of Systems Integration, 6, 41-72 (1996)
https://hdl.handle.net/10945/42328

Journal of Systems Integration, 6, 41–72 (1996)

Real-Time Scheduling for Software Prototyping

LUQI
Computer Science Department, Naval Postgraduate School, Monterey, CA 93943

M. SHING
Computer Science Department, Naval Postgraduate School, Monterey, CA 93943

Abstract. This paper presents several real-time scheduling algorithms developed to support rapid prototyping of embedded systems using the Computer Aided Prototyping System (CAPS). The CAPS tools are based on the Prototyping System Description Language (PSDL), which is a high-level language designed specifically to support the conceptual modeling of real-time embedded systems. This paper describes the scheduling algorithms used in CAPS along with the associated timing constraint and hardware models, which include single and multi-processor configurations.

Keywords: Computer-aided software engineering, rapid prototyping, hard real-time systems, real-time scheduling.

1. Introduction

The correctness of a hard real-time system depends not only on the logical result of computation, but also on the time at which the results are produced [28, 26]. One of the major differences between a hard real-time system and a conventional system is that the application software must meet its deadlines even under worst case conditions. Large scale, parallel and distributed, hard real-time systems are important to both civilian and military applications. Examples of hard real-time systems include air traffic control systems, controls for automated factories, telecommunication systems, space shuttle avionics systems and C3I systems. Hard real-time software systems are typically embedded in larger systems, performing critical control functions. These real-time control functions may require the software system to interact with a wide variety of hardware/software subsystems via networks. The design and development of these systems is often plagued with uncertainty, inconsistency, unpredictability, and brittleness.

Rapid prototyping can be used to reduce the risks of producing hard real-time systems that do not meet customer needs [14]. The Computer Aided Prototyping System (CAPS) [18, 21] supports an iterative prototyping process characterized by exploratory design and extensive prototype evolution. It enables the engineers to produce complex systems that match user needs and reduces the need for expensive modifications after delivery by providing automated decision aids for designers and customers. Demonstrations of proposed system behavior can be effective for validating system requirements, especially for new or unfamiliar application areas. Unlike traditional approaches to software development, which produce working code only near the end of the process, rapid prototyping, when utilized during the early stages of the development life cycle, allows validation of the requirements, specification, and initial design before valuable time and effort are expended on implementation software. Prototyping of real-time systems depends on automated real-

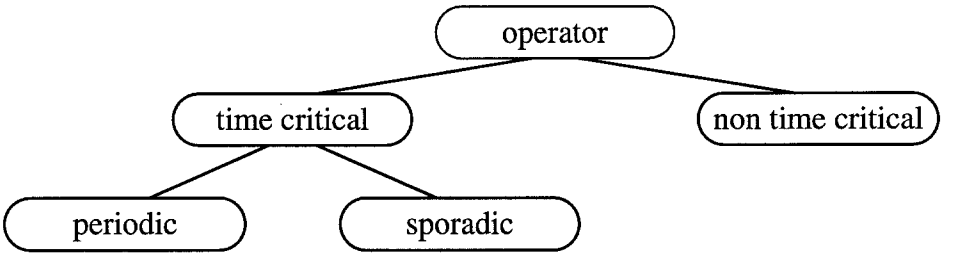

Figure 1.

time scheduling. This paper describes the scheduling methods used by CAPS and outlines directions for improvements.

2. Handling Timing Constraints in Caps

The CAPS tools are based on the Prototyping System Description Language (PSDL) [13], which is a high-level prototyping language designed specifically to support conceptual modeling of real-time embedded systems, including the timing aspects of hard real-time systems in single and multi-processor hardware configurations. The features of PSDL for specifying the real-time behavior of concurrent operations and their formal semantics are given in [12, 15, 20 and 22] and are briefly reviewed in this section.

PSDL models software systems as networks of operators communicating via data streams. This model can be represented as an augmented directed hypergraph whose nodes are operators and whose edges are streams. Edges can have multiple sources (operators writing into the stream) and multiple sinks (operators reading from the stream). The operators are state machines with zero or more private state variables. When an operator fires, it reads one data value from each of its input streams, updates zero or more of its state variables, and writes at most one data value into each of its output streams. The operators can only interact via the streams, which are the only shared resources in the model. The hypergraph is augmented by associating timing and control constraints with the operators and streams. The timing and control constraints determine the conditions under which the operators are activated (i.e. can be fired).

2.1. PSDL Real-Time Constraints

This section focuses on the timing constraints because they determine the scheduling problems that CAPS must solve. PSDL operators can be classified according to their timing constraints as shown in Fig. 1. An operator is time-critical if it has at least one timing constraint associated with it, and is non-time-critical otherwise. A time-critical operator is periodic if it is activated by a periodic temporal event, and it is sporadic if it is activated by the arrival of data. The types of timing constraints associated with PSDL operators and streams are summarized in Table 1.

Table 1. Types of PSDL timing constraints.

Constraint	Abbreviation	Applies to	Constrains	Default
maximum execution time	MET	time critical operators	cpu time	-
period	P	periodic operators	(activation, next activation)	-
finish within	FW	periodic operators	(activation, completion)	P
maximum response time	MRT	sporadic operators	(activation, completion)	heuristic
minimum calling period	MCP	sporadic operators	(activation, next activation)	MRT–MET
latency	L	streams	(write, next read)	0
minimum period	MP	streams	(write, next write)	0

The *maximum execution time* (MET) is the maximum amount of CPU time required to execute an operator under worst-case conditions. PSDL maximum execution times are expressed relative to the host hardware for the CAPS system; these times must be scaled by the scheduler if the target hardware for the prototype has a different execution speed. Every time-critical atomic operator must have a maximum execution time to enable the scheduler to allocate enough CPU time to meet its deadline. The allocated CPU time must start after the operator is activated and must end before the operator is due to be completed. This CPU time needs not be all in one contiguous interval, and it needs not be all on the same processor. However, schedules consisting of several disjoint time intervals and possibly different processors for the same operator must supply additional time within and between the intervals sufficient for context switching and interprocessor communication.

All of the other timing constraints are bounds on durations of time intervals defined by pairs of events (see Table 1). These bounds are specified by constants that have units of physical time. Consequently the representations of all timing constraints other than the maximum execution time are independent of the target hardware.

Periodic operators are activated at regular, predictable intervals: the time between one activation and the next is always exactly equal to the specified period. However, note that there can be a delay between the activation time, when firing is enabled, and the starting time, when firing actually begins. This delay, which is controlled by the scheduler, cannot exceed the bound (FW–MET), and is called the *slack* of the operator. The scheduler is free to choose the starting time for the first firing of a periodic operator o_i (denoted by starting_time($o_{i,1}$)), subject to the dataflow precedence constraints defined in Section 2.2.1 and the following initialization constraints:

$$\begin{aligned} \text{beginning\_time} &\leq \text{activation\_time}(o_{i,1}) = \text{starting\_time}(o_{i,1}) \\ &\leq \text{beginning\_time} + \text{P}(o_i), \end{aligned}$$

where beginning_time denotes the time at which the system begins firing the very first operator in the prototype. Denote the k^{th} instance of an operator o_i by $o_{i,k}$. The absolute times of all the activations of the periodic operator o_i and the corresponding deadlines are determined by the time of the first activation as follows.

$$\begin{aligned} \text{deadline}(o_{i,k}) &= \text{activation\_time}(o_{i,k}) + \text{FW}(o_i) \\ \text{activation\_time}(o_{i,k+1}) &= \text{activation\_time}(o_{i,k}) + \text{P}(o_i) \end{aligned}$$

Sporadic operators are activated (or triggered) by the arrival of new data on the input streams specified in the operator's control constraints. The activation time is the earliest time the triggering data is available for reading by the operator; this is the time the data is written plus any interprocessor communication delays due to a possibly distributed implementation. Scheduling is based on the following constraints.

$$\text{deadline}(o_{i,k}) = \text{activation\_time}(o_{i,k}) + \text{MRT}(o_i)$$
$$\text{activation\_time}(o_{i,k+1}) \geq \text{activation\_time}(o_{i,k}) + \text{MCP}(o_i)$$

Sporadic operators can be realized with finite computational resources only under the assumption that the activation rate is bounded. The required bound is specified by the minimum calling period if it is given, and defaults to the highest activation rate supported by all realizations of the required maximum response time (Table 1). If the required maximum response time is not known, the scheduler helps formulate the requirements by approximately determining the smallest value that can be realized under the assumption that all time critical sporadic operators without specified maximum response times are entitled to equal shares of available CPU time.

PSDL can also model communication delays and bandwidth constraints imposed by fixed allocations of external data sources and software functions to physical nodes of a distributed system. The *latency* of a stream is an upper bound on the time between the instant a data value is written into a stream and the instant that data value can be read from the stream. The *minimum period* is a lower bound on the time between two successive write events on the stream. The latencies and minimum periods declared in PSDL are external requirements that constrain the scheduler and the implementation. Additional constraints on latencies and minimum periods due to hardware constraints and resource allocations made by the scheduler are calculated by the scheduler based on the chosen hardware model, and are provided to the designer as feedback.

2.2. *Scheduling Constraints*

The order in which PSDL operators can be scheduled is influenced by precedence and mutual exclusion constraints, and the times at which operators can be scheduled are influenced by constraints derived from the specified model for the target hardware for the prototype.

2.2.1. *Precedence Constraints*

The dataflow precedence constraint requires the initial firings of all operators with timing constraints to occur in an order consistent with the *dataflow ordering*, which means the operators that write into a stream without a declared initial value must be fired before the first firing of an operator that reads from the stream. Formal definitions of this concept can be found in [22]. Assume that the periodic operator o_1 precedes another periodic operator o_2 in a given prototype. The instances $o_{1,i}$ and $o_{2,j}$ are subject to additional synchronization

precedence constraints if $(i-1) \times P(o_1) = (j-1) \times P(o_2)$. In such a case:

(1) the i^{th} instance of operator o_1 must complete firing before the j^{th} instance of operator o_2 can fire, and

(2) the j^{th} instance of operator o_2 must read its inputs before the $(i+1)^{st}$ instance of operator o_1 can fire.

The purpose of these constraints is to ensure the instance $o_{2,j}$ operates on the data produced by the instance $o_{1,i}$. The first constraint is needed to ensure that the output of $o_{1,i}$ has been produced before it is used by $o_{2,j}$, and the second constraint is needed to ensure that the output of $o_{1,i}$ is not over-written by the output of $o_{1,i+1}$ before it can be read by $o_{2,j}$. In distributed architectures, if two instances of periodic operators subject to a synchronization constraint are allocated to different processors, then the scheduler must provide sufficient time between their execution intervals to cover any possible interprocessor communications delays.

2.2.2. *Mutual Exclusion*

Since updates to state variables must be serialized to preserve the integrity of the data, any pair of operators that belong to a common cycle in the expanded dataflow graph[1] must not be scheduled concurrently. If two such operators are allocated to different processors of a distributed target architecture, then the scheduler must provide sufficient time between the completion of one such operator and the start of the next to account for possible interprocessor communication delays.

2.2.3. *Pipelining*

A time-critical operator whose period or minimum period is less than its maximum execution-time can be only be realized if it can be pipelined (i.e. more than one instance of the operator can be firing at the same time). A PSDL operator can be pipelined if and only if the operator does not appear on a cycle in the expanded dataflow graph, and the operator does not have internal states.

2.3. *Hardware Models*

The semantics of PSDL is independent of the hardware model, but scheduling and the feasibility of realizing the declared real-time constraints depend on the architecture and characteristics of the hardware system on which the proposed system will run. In particular, methods for static scheduling are strongly influenced by the hardware model. All of the hardware models associated with PSDL are based on the following assumptions [22]:

(1) The speed of a processor is independent of the type of program it is executing.

(2) The entire capacity of the hardware is available for critical real-time computations.

(3) The capacity of the hardware configuration is known before execution begins and does not change with time.

The hardware models associated with PSDL can be characterized by the number of processors N, a vector of processor speeds S_i, a matrix of interprocessor delays $D_{i,j}$, and a matrix of inverse link speeds (seconds per bit) $T_{i,j}$, where $D_{i,i} = 0$, $T_{i,i} = 0$, and $1 \leq i, j \leq N$. Some useful special cases are a single processor ($N = 1$), identical processors ($S_i = s$), shared memory ($D_{i,j} = 0$), unlimited bandwidth ($T_{i,j} = 0$), and a homogeneous network ($D_{i,j} = d$ for $j \neq i$). The derived latency for the transmission of a data value b bits long from processor i to processor j is $D_{i,j} + b \times T_{i,j}$.

2.4. *Feasibility*

To provide useful diagnostic information, the scheduler checks the following necessary conditions for the existence of a feasible schedule and reports violations to the designer.

(1) Basic CPU time requirements imply that periodic operators must have MET $\leq$ FW and sporadic operators must have MET $\leq$ MRT.

(2) In the absence of pipelining we must also have MET $\leq$ P for periodic operators and MET $\leq$ MCP for sporadic operators.

(3) MET(x) $<$ P(y) for any two operators x and y which are placed on the same processor.

(4) For a set of periodic operators to be schedulable on N processors, the *load factor*, which equals $\sum$ MET(x)/P(x) over all periodic operators x in the prototype, must be $\leq N$.

The scheduler also checks each operator-pair connected by dataflow streams to ensure that the consumer's period is not greater than that of the producer. Stream buffer overflows will result if this constraint is violated. Furthermore, the scheduler in the current version of CAPS does not handle operators which require pipelining. Any prototype that contains operators with MET $>$ P will be considered unschedulable and the scheduler will report the violations to the designer.

3. Real-Time Scheduling Methods in CAPS

One of the major tasks in rapid prototyping is to determine whether the timing constraints of a given specification can be satisfied by some real-time program. The feasibility analysis is usually done either via static timing analyzers [27, 33] or pre-run-time schedulers [34]. One drawback of static timing analyzers is that the analysis works well only if the hard real-time system runs exactly as specified in the high-level description. This can be difficult to achieve in a portable fashion due to operating system dependencies. Hence, CAPS chooses to demonstrate the schedulability of a prototype via the generation of a static run-time schedule that enforces all hard real-time-constraints under the worst case conditions.

Table 2. LCM optimizations.

Operators	MET	Initial Period	New Period
1	20	100	100
2	50	500	500
3	80	600	600
4	100	800	800
5	165	1035	**1000**
LCM		828000	3000
Load Factor		0.718	0.723

Like [23], the scheduler converts all sporadic time critical operators into equivalent periodic operators. As shown in [16, 4], the "equivalent period" of the sporadic operator must be $\leq$ min(MRT–MET, MCP) and the finish within must be set to (MRT − "equivalent period") to catch every set of triggering data and process it within MRT. Since it is desirable to set the "equivalent period" as large as possible in order to minimize the impact on the load factor of the prototype, it seems logical for the scheduler to use min(MRT–MET, MCP) as the default "equivalent period" for the sporadic operators. However, such defaults may result in a set of periods with a very large LCM. (For example, the set of initial periods shown in Table 2 has an LCM of 828,000.) We observed in our early experiments that prototypes with large LCM's are less likely to be schedulable. Furthermore, a very small change in the periods, while only affects the load factor slightly, may be sufficient to get rid of some prime factors of the LCM and reduce the LCM significantly. (For example, changing the period 1035 to 1000 in Table 2 reduces the LCM from 828,000 to 3,000.) Hence, we have developed a heuristic algorithm to minimize the LCM of a prototype [4]. The algorithm allows the user to specify the range of acceptable values for the period of each operator and tries to reduce the LCM by replacing the periods containing the prime factors that was driving up the LCM with other values within the allowable range. An early experiment with the algorithm on 50 randomly generated prototypes shows that the new LCMs represent an average of 47% reduction over the original ones.

We shall assume that all time critical operators are periodic for the rest of the paper. A set O of non-preemptive periodic operators with precedence relationship is *schedulable* if there exists a static schedule such that the starting and completion time of every operator instance satisfy the timing and scheduling constraints in Section 2. It is a well known and accepted result that the least common multiple (LCM) of their periods provides a finite interval of time, for which a cyclic schedule can be calculated, if one exists, and repeated forever [23]. Many interpret the above statement to mean that a cyclic feasible schedule must only exist in the closed interval [0, LCM], meaning that each operator instance that starts within the interval [0, LCM] must complete its execution by time LCM. Such an interpretation is overly restrictive. Consider a set with two operators o_1 and o_2 shown in Fig. 2a, with MET(o_1) = 190, P(o_1) = FW(o_1) = 600, MET(o_2) = 20, and P(o_2) = FW(o_2) = 200. Since o_1 precedes o_2, the first instance of o_2 cannot start before time 190, forcing the third instance of o_2 to start at time 590 and complete at time 610. Hence, no feasible

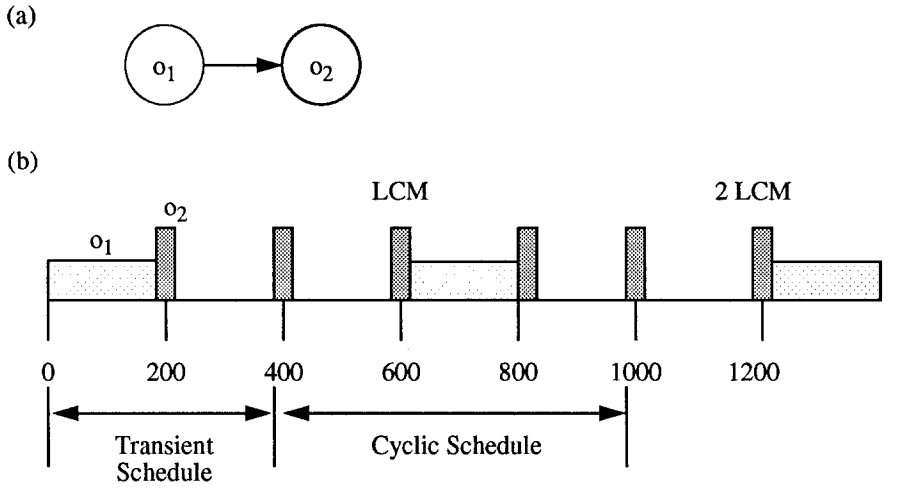

Figure 2.

schedule exists if we require every operator instance that starts within the interval [0, 600] to complete its execution by time 600, even though the schedule shown in Fig. 2b satisfies all the constraints outlined in Section 2.

In [4], Cordeiro proved that

> "If there exists an infinite feasible schedule S without any inserted idle time[2] for a set of periodic operators with precedence constraints, such that the first instance of every operator o_i must start by time $P(o_i)$, then there exists an infinite schedule S' consisting of a transient portion of length at most LCM, followed by a cyclic portion of length LCM that repeats forever."

For example, in the schedule shown in Fig. 2b, the third instance of o_2 has to start at time 590 in order to allow itself and the second instance of o_1 to both meet their deadlines. This forces the second instance of o_1 to delay its actual starting time to 610 and the fourth instance of o_2 to delay its actual starting time to 800, resulting in feasible schedule with a transient portion of length 390 following by a cyclic portion of length 600.

Based on Cordeiro's observation, it suffices to compute the cyclic schedule within [0, 2 × LCM). This is done by considering a scheduling constraint graph CG that contains all task instances that must start in the interval [0, 2 × LCM). The scheduling constraint graph can be constructed using the algorithm in Table 3. (For example, applying the algorithm to the expanded data flow graph shown in Fig. 3a results in the scheduling constraint graph shown in Fig. 3c.) Note that the CAPS scheduler does not construct the scheduling constraint graph CG explicitly. It computes the precedence constraints described by CG dynamically as it builds the static schedule based on the global precedence graph G' of the prototype,

Table 3. Construction of the scheduling constraint graph.

Given the expanded dataflow graph G, define the scheduling constraint graph $CG = (V, E)$ as follows:
(1) First, obtain a global precedence graph G' by removing all edges in G which represent state variables and then taking the transitive closure of the resultant graph.
(2) For each operator o_i in G', create a vertex for each instance of o_i that must appear in the static schedule. Denote the vertices created by $\{o_{i,1}, o_{i,2}, \ldots, o_{i,n_i}\}$, where $n_i = 2 \times \text{LCM}/\text{P}(o_i)$, the number of instances of o_i that can be activated within a interval of $2 \times$ LCM.
(3) For $1 \leq k \leq n_i - 1$, add an edge $o_{i,k} \rightarrow o_{i,k+1}$ of zero latency to CG.
(4) For any two vertices $o_{i,p}$ and $o_{j,q}$ obtained in Step (1), add the edges $o_{i,p} \rightarrow o_{j,q}$ of latency $\text{L}(o_i \rightarrow o_j)$ and $o_{j,q} \rightarrow o_{i,p+1}$ of latency $\text{L}(o_j \rightarrow o_i)$ if o_i precedes o_j in G' and synchronization is needed. ($\text{L}(o_j \rightarrow o_i) = 0$ if the edge $o_j \rightarrow o_i$ is not present in G.)
(5) For any two vertices o_i and o_j in G', if o_i precedes o_j and mutual exclusion exists between the two vertices, then, for $1 \leq k \leq n_i$, add the edges $o_{i,k} \rightarrow o_{j,k}$ with latency $\text{L}(o_i \rightarrow o_j)$ and $o_{j,k} \rightarrow o_{i,k+1}$ with latency $\text{L}(o_j \rightarrow o_i)$ to CG.
(6) Create a dummy vertex DUMMY with MET = 0 and P = $2 \times$ LCM.
(7) For each vertex $o_{i,1}$ that has no incoming edges after Steps (1) and (2), connect DUMMY to $o_{i,1}$ with the edge DUMMY $\rightarrow o_{i,1}$ and set the latency of the edge to zero.

which can be obtained from the expanded dataflow graph G by removing all edges in G which represent state variables and then taking the transitive closure of the resultant graph. For brevity, the notations shown in Table 4 will be used throughout the remaining paper.

3.1. The PSDL Scheduling Problem

Given a scheduling constraint graph CG and a set of N identical processors with a common shared memory, a *static schedule*, is function that maps each instance of the operators that must start within [0, $2 \times$ LCM) to a triple *(pid, st, ct)* where *pid* is the label of the processor that executes the operator instance, st is the exact execution start time for the operator instance and $ct = st +$ MET, the time by which the operator instance must complete its execution. A static schedule is said to be *legal* if the relative ordering of the operator-instances (i.e. vertices of CG) in the schedule satisfies the precedence constraints imposed by CG. A static schedule is said to be *feasible* if the schedule is legal and every operator-instance when executed according to the schedule meets its deadline. The *cost* of a schedule is defined to be maximum tardiness over all operator-instances in CG. Hence, any legal schedule with zero cost is a feasible schedule.

The static scheduling problem is to decide if there is a feasible schedule for the given scheduling constraint graph CG on a set of N identical processors. (See [29] for a survey of the complexities of various real-time scheduling problems.) Since the static scheduling problem is NP-hard [30, 31, 35], it not likely to have efficient algorithms for solving the general static scheduling problem. Hence, both exponential-time optimal scheduling algorithms and fast heuristic scheduling algorithms are considered in the CAPS system.

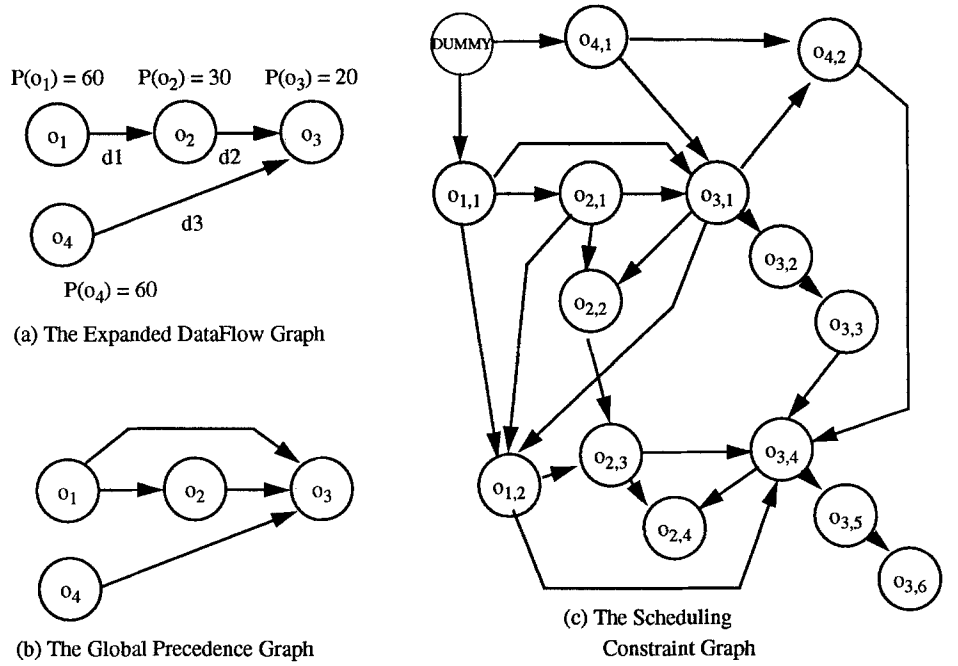

Figure 3.

3.2. Single Processor Scheduling

The static scheduling algorithms currently available in CAPS for the uniprocessor configuration fall into three categories: the exponential-time optimal algorithms, the fast heuristic algorithms, and the parameterized search algorithms [2, 5, 11, 17]. Algorithms in the first category (*exhaustive-enumeration* and *branch-and-bound*) guarantee finding a feasible schedule if one exists, but their long running times limit their usefulness to small problems. Algorithms in the second category (*earliest-starting-time-first*, and *earliest-deadline-first*) on the other hand, are very efficient. Given a scheduling constraint graph, each of these algorithms only tests one legal schedule for feasibility, and can fail to find the feasible schedule even if one exists. In order to increase the chance of finding a feasible schedule, CAPS provides two parameterized search algorithms (*simulated annealing* and *limited backtrack*) which allow users to control the trade off between computational time and the number of legal schedules tested with a set of parameters.

Table 4. Summary of notations.

	Notation	Meaning
	$\text{act}(o_{i,k})$	the beginning of the k^{th} period of the operator o_i
	$\text{st}(o_{i,k})$	the actual starting time of $o_{i,k}$
	$\text{ct}(o_{i,k})$	the completion time of $o_{i,k}$
	$\text{d}(o_{i,k})$	denote the deadline (i.e. latest completion time) of $o_{i,k}$
	$\text{tardiness}(o_{i,k})$	the amount of time by which $o_{i,k}$ misses its deadline
	$\text{ready}(o_{i,k})$	the earliest time when o_i can actually fires in the k^{th} period
	$\text{parent}(o_{i,k})$	the set of parents of $o_{i,k}$ in CG
	$\text{children}(o_{i,k})$	the set of children of $o_{i,k}$ in CG
where		
	ready(DUMMY)	$= \text{st(DUMMY)} = \text{ct(DUMMY)} = 0$,
	$\text{ready}(o_{i,1})$	$= \max\{\text{ct}(u) + \text{L}(u \rightarrow o_{i,1}) \mid \text{for all } u \in \text{parent}(o_i)\}$, $0 \leq \text{ready}(o_{i,1}) \leq \text{st}(o_{i,1}) \leq \text{P}(o_i)$,
	$d(o_{i,1})$	$= \min\{\text{P}(o_i) + \text{MET}(o_i), d(o_{j,1}) - \text{MET}(o_j) - \text{L}(o_{i,1} \rightarrow o_{j,1}) \mid \text{for all } o_{j,1} \in \text{children}(o_{i,1})\}$, i.e., $\text{d}(o_{i,1})$ is the latest time o_i has to complete its first instance so that all other first instances $o_{j,1}$ following $o_{i,1}$ in CG can also complete their firing by $\text{P}(o_j) + \text{MET}(o_j)$
	$\text{ct}(o_{i,1})$	$= \text{st}(o_{i,1}) + \text{MET}(o_i)$,
	$\text{tardiness}(o_{i,1})$	$= \max\{\text{ct}(o_{i,1}) - \text{d}(o_{i,1}), 0\}$.
And for $k > 1$,		
	$\text{act}(o_{i,k})$	$= \text{st}(o_{i,1}) + (k-1) \times \text{P}(o_i)$
	$\text{ready}(o_{i,k})$	$= \max\{\text{act}(o_{i,k}), \text{ct}(u) + \text{L}(u \rightarrow o_{i,k}) \mid \text{for all } u \in \text{parent}(o_{i,k})\}$, $\text{act}(o_{i,k}) \leq \text{ready}(o_{i,k}) \leq \text{st}(o_{i,k})$,
	$\text{ct}(o_{i,k})$	$= \text{st}(o_{i,k}) + \text{MET}(o_i)$,
	$\text{d}(o_{i,k})$	$= \text{act}(o_{i,k}) + \text{FW}(o_i)$,
	$\text{tardiness}(o_{i,k})$	$= \max\{\text{ct}(o_{i,k}) - \text{d}(o_{i,k}), 0\}$.

3.2.1. *Exhaustive-Enumeration and Branch-And-Bound*

Exhaustive-enumeration (Table 5) is a very simple algorithm that inspects all legal schedules one by one and returns the first feasible schedule it finds.

Since we are only interested in feasible schedules, the algorithm will cut off any partial schedule that has an operator-instance with a positive tardiness. One way to further reduce the running time of the exhaustive-enumeration method is by modifying Line (8) of the BackTrack procedure to cut off a partial schedule based on an estimated cost, resulting in the Branch-and-Bound procedure shown in Table 6. The estimated cost is a lower bound on the cost of all the legal schedules generated from the common partial schedule, and is computed by the function Estimate_Cost shown in Table 7.

3.2.2. *Earliest-Starting-Time-First and Earliest-Deadline-First*

Both the earliest-starting-time-first and the earliest-deadline-first algorithms follow the logic of the topological-ordering algorithm (shown in Table 8), which produces a legal schedule by sorting the vertices in CG topologically. They only differ in the way in which vertices

Table 5. The Exhaustive-Enumeration Algorithm.

```
The Exhaustive_Enumeration Algorithm:
Begin
(1)   Last_Stop_Time := 0;
(2)   Partial_Schedule := empty;
(3)   Ready_Set := DUMMY;
(4)   Best_Schedule := empty;
(5)   BackTrack(Last_Stop_Time, Partial_Schedule, Ready_Set, Best_Schedule);
(6)   if Best_Schedule /= empty then
(7)     Output "Schedule Found";
(8)     Output Best_Schedule;
(9)   else
(10)    Output "Schedule Not Found";
(11)  end if;
End.

Procedure BackTrack(Last_Stop_Time, Partial_Schedule, Ready_Set, Best_Schedule):
Begin
(1)   Working_Ready_Set := Ready_Set;
(2)   Found := false;
(3)   while not Found and Not_Empty(Working_Ready_Set) loop
(4)     Temp_Schedule := Partial_Schedule;
(5)     v := Remove_Item_From_Set(Working_Ready_Set);
(6)     st(v) := max {Last_Stop_Time, ready(v)};
(7)     ct(v) := st(v) + MET(v);
(8)     if ct(v) ≤ d(v) then
(9)       Add_Item_To_Schedule(v, Temp_Schedule)
(10)      Temp_Ready_Set := Ready_Set − {v};
(11)      For each child u of v in CG loop
(12)        if all parents of u are in Temp_Schedule then
(13)          Add_Item_To_Set(u, Temp_Ready_Set);
(14)        end if;
(15)      end loop;
(16)      if Not_Empty(Temp_Ready_Set) then
(17)        BackTrack(ct(v), Temp_Schedule, Temp_Ready_Set, Best_Schedule);
(18)        Found := Non_Empty(Best_Schedule);
(19)      else -- no unscheduled vertex
(20)        Best_Schedule := Temp_Schedule;
(21)        Found := true;
(22)      end if;
(23)    end if;
(24)  end loop;
End.
```

Table 6. The Branch-and-Bound Procedure.

```
Procedure Branch_And_Bound(Last_Stop_Time, Partial_Schedule, Ready_Set, Best_Schedule):
Begin
(1)   Working_Ready_Set := Ready_Set;
(2)   Found := false;
(3)   while not Found and Not_Empty(Working_Ready_Set) loop
(4)     Temp_Schedule := Partial_Schedule;
(5)     v := Remove_Item_From_Set(Working_Ready_Set);
(6)     st(v) := max {Last_Stop_Time, ready(v)};
(7)     ct(v) := st(v) + MET(v);
(8)     Add_Item_To_Schedule(v, Temp_Schedule)
(9)     if Estimate_Cost(Temp_Schedule) ≤ 0 then
(10)      Temp_Ready_Set := Ready_Set −{v};
(11)      For each child u of v in CG loop
(12)        if all parents of u are in Temp_Schedule then
(13)          Add_Item_To_Set(u, Temp_Ready_Set);
(14)        end if;
(15)      end loop;
(16)      if Not_Empty(Temp_Ready_Set) then
(17)        Branch_And_Bound(ct(v), Temp_Schedule, Temp_Ready_Set, Best_Schedule);
(18)        Found := Non_Empty(Best_Schedule);
(19)      else -- no unscheduled vertex
(20)        Best_Schedule := Temp_Schedule;
(21)        Found := true;
(22)      end if;
(23)    end if;
(24)  end loop;
End.
```

Table 7. Estimating the lower bounding cost of a partial schedule.

```
Function Estimate_Cost(Partial_Schedule):
Begin
(1)  Lower_Bound := max{0, ct(v) − d(v) | for all vertex v in Partial_Schedule};
(2)  Last_Stop_Time := ct(w) where w is the last scheduled vertex in Partial_Schedule;
(3)  For each unscheduled vertex o_{i,k} in CG loop
(4)    Lower_Bound := max{Lower_Bound, est(o_{i,k}) + MET(o_i) − ed(o_{i,k})}
         where
           est(o_{i,k}), lower bound on the starting time of o_{i,k}, equals
             Last_Stop_Time + Σ MET(u) over all unscheduled ancestors u of o_{i,k} in CG,
           ed(o_{i,k}), upper bound on the deadline of o_{i,k}, equals
             st(o_{i,1}) + (k − 1) × P(o_i) + FW(o_i)      if k > 1,
             and equals P(o_i) + MET(o_i)                  if k = 1.
(5)  end loop;
(6)  return(Lower_Bound);
End.
```

Table 8. The Topological-Ordering Algorithm.

```
The Topological_Ordering Algorithm:
Begin
(1)   Ready_Set := {Dummy};
(2)   Schedule := empty;
(3)   Last_Stop_Time := 0;
(4)   While Not_Empty(Ready_Set) loop
(5)     v := Remove_Item_From_Set(Ready_Set);
(6)     st(v) := max {Last_Stop_Time, ready(v)};
(7)     ct(v) := st(v) + MET(v);
(8)     Add_Item_To_Schedule(v, Schedule)
(9)     Last_Stop_Time := ct(v);
(10)    For each child u of v in CG loop
(11)      if all parents of u are in Schedule then
(12)        Add_Item_To_Set(u, Ready_Set);
(13)      end if;
(14)    end loop;
(15)  end loop;
(16)  if Cost(Schedule) = 0 then
(17)    Output "Schedule Found";
(18)  else
(19)    Output "Schedule Not Found";
(20)  end if;
(21)  Output Schedule;
End.
```

are removed from the Ready_Set in Line (5) of the topological-ordering algorithm.

The earliest-starting-time-first algorithm works like the topological-ordering algorithm, except that it always removes the vertex with the earliest ready time among all the vertices in the Ready_Set (Table 9).

The earliest-deadline-first algorithm, on the other hand, always removes the vertex with the earliest deadline among all the vertices v in the Ready_Set with ready(v) $\leq$ Last_Stop_Time (Table 10). If every vertex v in the Ready_Set has ready(v) $>$ Last_Stop_Time, then the one with the earliest starting time will be chosen to minimize the CPU idle time.

3.2.3. Simulated-Annealing

The major drawback of the previous two algorithms is that they both take a hit-or-miss attitude, since they only test one legal schedule for feasibility. One way to increase the chance of finding a feasible schedule without spending exponential execution time is the use of stochastic search. CAPS provides a fifth algorithm that finds feasible schedules using simulated annealing (Table 11). Simulated annealing is a search technique based upon the Metropolis Algorithm, which simulates a complex system of particles (molecules) in a heat bath [24]. Recognizing concepts similar to optimization, Kirkpatrick *et al.* [10] and Cerny [1] independently developed simulated annealing. Since then, many researchers have used it to solve a variety of combinatorial optimization problems [8, 9, 25, 32].

Table 9. The Earliest_Starting_Time_First Algorithm.

```
The Earliest_Starting_Time_First Algorithm:
Begin
(1)   Ready_Set := {Dummy};
(2)   Schedule := empty;
(3)   Last_Stop_Time := 0;
(4)   While Not_Empty(Ready_Set) loop
(5)     v := Remove_Item_With_Earliest_Start_Time(Ready_Set);
(6)     st(v) := max{Last_Stop_Time, ready(v)};
(7)     ct(v) := st(v) + MET(v);
(8)     Add_Item_To_Schedule(v, Schedule)
(9)     Last_Stop_Time := ct(v);
(10)    For each child u of v in CG loop
(11)      if all parents of u are in Schedule then
(12)        Add_Item_To_Set(u, Ready_Set);
(13)      end if;
(14)    end loop;
(15)  end loop;
(16)  if Cost(Schedule) = 0 then
(17)    Output "Schedule Found";
(18)  else
(19)    Output "Schedule Not Found";
(20)  end if;
(21)  Output Schedule;
End.
```

Starting from the initial infeasible legal schedule S, the algorithm randomly perturbs this schedule to obtain a new legal schedule Temp_S. As in the standard local iterative improvement approach, the algorithm always replaces S with Temp_S if the change in cost, ΔC, is non-positive. However, unlike the local iterative improvement approach, Temp_S is accepted with probability $= exp(-\Delta C/T)$ if ΔC is positive.

The control temperature, T, is a value in the same units as the cost function. It regulates the probability distribution that defines the acceptance criteria of the new schedules with degrading costs. At each temperature, T, the procedure attempts up to either a total of L trials or L_a acceptance moves. The temperature is then reduced by a cooling factor R. The resulting behavior is a downward-biased random walk through the solution space, with the ability to escape local minima. Gradually decreasing control temperature changes the exponential probability distribution. This tightens the acceptance criteria against larger degradations. The value of T for which no degradations are reasonably expected and no more improvements can be found is T_f, the freezing temperature. At this stage the algorithm returns the best solution, and halts. Since we are only interested in finding a feasible schedule, i.e. a legal schedule with zero cost, we have modified the simulated annealing algorithm to halt as soon as it encounters such a schedule. To utilize the simulated annealing algorithm, we must provide efficient and effective ways to

(1) obtain the initial legal schedule,

(2) perturb the existing schedule to obtain new legal schedules, and

Table 10. The Earliest_Deadline_First Algorithm.

```
The Earliest_Deadline_First Algorithm:
Begin
(1)   Ready_Set := {Dummy};
(2)   Schedule := empty;
(3)   Last_Stop_Time := 0;
(4)   While Not_Empty(Ready_Set) loop
(5)     Earliest_Deadline_Set := {v | v ∈ Ready_Set and ready(v) <= Last_Stop_Time};
(6)     if Not_Empty(Earliest_Deadline_Set) then
(7)       v := Remove_Item_With_Earliest_Deadline(Earliest_Deadline_Set);
(8)       Ready_Set := Ready_Set − {v};
(9)     else
(10)      v := Remove_Item_With_Earliest_Start_Time(Ready_Set);
(11)    end if;
(12)    st(v) := max{Last_Stop_Time, ready(v)};
(13)    ct(v) := st(v) + MET(v);
(14)    Add_Item_To_Schedule(v, Schedule)
(15)    Last_Stop_Time := ct(v);
(16)    For each child u of v in CG loop
(17)      if all parents of u are in Schedule then
(18)        Add_Item_To_Set(u, Ready_Set);
(19)      end if;
(20)    end loop;
(21)  end loop;
(22)  if Cost(Schedule) = 0 then
(23)    Output "Schedule Found";
(24)  else
(25)    Output "Schedule Not Found";
(26)  end if;
(27)  Output Schedule;
End.
```

(3) controls the number of legal schedules being examined at each temperature T and the rate at which T is lowered.

Although annealing can begin from any solution in the search space, empirical evidence suggested that reasonably good initial solutions can often provide better final results [8, 25]. Hence, we always run the earliest-deadline-first algorithm before the annealing algorithm. The result generated by the earliest-deadline-first algorithm will be used as the starting solution of the annealing process if it is not a feasible schedule.

The method for adjusting a given schedule to generate new schedules must maintain the precedence relationships between the tasks as defined by the constraint graph CG. The Adjust_Schedule routine (Table 12) produces a new schedule either by (1) constructing a brand new schedule from scratch using a randomized version of the topological-ordering algorithm or (2) local re-arrangement of the operator instances in the current schedule. Although local re-arrangement is a much faster operation than generating a brand new schedule, our empirical data show that local re-arrangement often causes the search process to be trapped at local minima. Hence, the Adjust_Schedule routine is designed to bias towards generating brand new schedules.

Table 11. The Simulated-Annealing Algorithm.

The Simulated_Annealing Algorithm:

```
Begin
(1)   Use Earliest_Deadline_First algorithm to find a legal schedule S;
(2)   if Cost(S) = 0 then
(3)     Output "Schedule Found";
(4)     Output S;
(5)   else
(6)     Found := false;
(7)     Best_Schedule := S;
(8)     T0 := 2 × Cost(S);          -- set initial temperature
(9)     Tf := 1;                    -- set freezing temperature
(10)    L := 100;                   -- set maximum number of schedules sampled at each temperature
(11)    La := 35;                   -- set maximum number of schedules accepted at each temperature
(12)    R := 0.85;                  -- set cooling factor
(13)    while (T > Tf and not Found) loop
(14)      N := 0;                   -- keep track the number of schedules sampled at current T
(15)      Na := 0;                  -- keep track the number of schedules accepted at current T
(16)      while (N < L and Na < La and not Found) loop
(17)        Temp_S := Adjust_Schedule(S);
(18)        N := N + 1;
(19)        if Cost(Temp_S) = 0 then
(20)          BEST_SCHEDULE := Temp_S;
(21)          Found := true;
(22)        else
(23)          if Cost(Temp_S) < Cost(Best_Schedule) then
(24)            Best_Schedule := Temp_S;
(25)          end if;
(26)          ΔC := Cost(Temp_S) − Cost(S);
(27)          if (ΔC < 0 or else random( ) < exp(−ΔC/T)) then
(28)            S := Temp_S;
(29)            Na := Na + 1;
(30)          end if;
(31)        end if;
(32)      end loop;
(33)      T := T × R;
(34)    end loop;
(35)    if not Found then
(36)      Output "Schedule Not Found"
(37)    else
(38)      Output "Schedule Found";
(39)    end if;
(40)    Output Best_Schedule;
(41)  end if;
End.
```

Table 12. The Adjust_Schedule Routine.

```
Function Random_Schedule():
Begin
      generates a new schedule from scratch by randomly removing vertices among
      all the vertices in the Ready_Set in Line (5) of the topological-ordering algorithm.
End.

Function Adjust_Schedule(S):
Begin
(1)   if random() ≤ 0.6 then
(2)     return Random_Schedule();
(3)   else
(4)     start randomly at some point in the schedule S, traverse up the
          schedule and find the first task, say v, with a positive tardiness.
(5)     If no task with a positive tardiness is found, then
          return Random_Schedule();
(6)     else -- move v as far up the schedule as possible
(7)       Let u be the task immediately before v in the schedule S.
(8)       if u is a parent of v or ready(v) > st(u) then -- cannot move v at all
(9)         return Random_Schedule();
(10)      else
(11)        While u is not a parent of v and ready(v) ≤ st(u) loop
(12)          interchange(u, v);
(13)        end loop;
(14)        Update st(w) and ct(w) for each vertex w affected by the move.
(15)      end if;
(16)    end if;
(17)  end if;
End.
```

The choice for T_o, T_f, R, L provides the trade-off between the running time and the effectiveness of the annealing algorithm. The higher the initial temperature T_o, the larger the cooling factor R, and the larger the number of trials L at each temperature will result in a more thorough search of the solution space. To avoid excessive sampling at a particular temperature, the annealing algorithm keeps track of the number of schedules accepted at each temperature and forces the annealing process to reduce its temperature when a total of L_a schedules have been accepted. These parameters are normally established from trial and error experimentation [6]. The goal in choosing these parameters is to ensure that a sufficient, but not excessive, number of solutions are examined. The following parameters used in our experiment:

T_o = twice the cost of the starting solution,
T_f = 1.0,
R = 0.85,
L = 100,
L_a = 35.

3.2.4. *Limited-Backtrack*

Early experimentation with the earliest-deadline-first algorithm showed that the algorithm is very fast and very effective for prototypes with load factor below 0.6. A closer inspection of the infeasible schedules showed that, in many cases, either no feasible exists, or a simple interchanging of the relative ordering among the operator-instances with the three earliest deadlines will result in a feasible schedule. Hence, a sixth algorithm, limited-backtrack, was developed to take advantage of this observation.

The limited-backtrack algorithm (Table 13) enumerates the legal schedules like the exhaustive-enumeration algorithm. However, it differs from the exhaustive-enumeration algorithm in the way in which vertices are removed from the Working_Ready_Set in Line (5) of the BackTrack procedure. The vertices are removed from the Working_Ready_Set in the order of non-increasing deadlines and at most Backtrack_Limit vertices will be expanded at each backtrack level.

3.2.5. *Performance Evaluation*

All the algorithms described in the previous subsections have been implemented in Ada and tested on several of prototypes [19]. Early tests show that the exponential-time algorithms (exhaustive-enumeration and branch-and-bound) take too much time to run except for very small problems, although they always guarantee finding a feasible solution if one exists. Furthermore, due to the fact that the Estimate_Cost is a very time consuming operation, the branch-and-bound algorithm actually ran slower than the simple exhaustive-search algorithm in most of the cases tested.

The earliest-deadline-first and the earliest-starting-time-first algorithms are very efficient and perform equally well for most of the prototypes we tested. In order to better judge the performance of these two algorithms under different load factors and scheduling constraint graph complexity, we conducted a second empirical study where we applied the CAPS scheduler to a total of 2700 prototypes generated by the PSDL random graph generator developed by Cordeiro [4]. The 2700 prototypes are made up of 9 groups of expanded dataflow graphs. The prototypes in each groups are generated based on a unique combination of dataflow graph size and edge density.[3] (See Table 14 for a summary of the random prototypes.)

Among the 2700 random prototypes, 356 prototypes have load factors exceeding 1.0 and are rejected by the scheduler. The remaining 2344 prototypes are used to test the efficiency and effectiveness of the earliest-deadline-first and the earliest-starting-time-first algorithms.

Efficiency of the algorithms are measured by the elapsed time taken by the algorithms to produce a schedule from a global precedence graph. Since the elapsed time is based on the real-time clock on a Sun SPARCstation and may vary significantly depending on the system load, it only provides a rough measure of the efficiency of the algorithms. The average running times shown in Table 15 clearly indicate that both algorithms are very efficient, with the earliest-starting-time-first algorithm slightly faster than the earliest-deadline-first algorithm in most of the cases.

Table 13. The Limited-Backtrack Algorithm.

```
Procedure Limited_BackTrack(Backtrack_Limit, Last_Stop_Time,
                            Partial_Schedule, Ready_Set, Best_Schedule):
Begin
(1)   Working_Ready_Set := Ready_Set;
(2)   Found := false;
(3)   Count := 0;
(4)   while not Found and Not_Empty(Working_Ready_Set) and Count < Backtrack_Limit loop
(5)     Count := Count + 1;
(6)     Temp_Schedule := Partial_Schedule;
(7)     Earliest_Deadline_Set := {v | v in Working_Ready_Set and ready(v) <= Last_Stop_Time};
(8)     if Not_Empty(Earliest_Deadline_Set) then
(9)       v := Remove_Item_With_Earliest_Deadline(Earliest_Deadline_Set);
(10)      Working_Ready_Set := Working_Ready_Set − {v};
(11)    else
(12)      v := Remove_Item_With_Earliest_Start_Time(Working_Ready_Set);
(13)    end if;
(14)    st(v) := max{Last_Stop_Time, ready(v)};
(15)    ct(v) := st(v) + MET(v);
(16)    if ct(v) ≤ d(v) then
(17)      Add_Item_To_Schedule(v, Temp_Schedule)
(18)      Temp_Ready_Set := Ready_Set −{v};
(19)      For each child u of v in CG loop
(20)        if all parents of u are in Temp_Schedule then
(21)          Add_Item_To_Set(u, Temp_Ready_Set);
(22)        end if;
(23)      end loop;
(24)      if Not_Empty(Temp_Ready_Set) then
(25)        Limited_BackTrack(Backtrack_Limit, ct(v), Temp_Schedule, Temp_Ready_Set, Best_Schedule);
(26)        Found := Non_Empty(Best_Schedule);
(27)      else -- no unscheduled vertex
(28)        Best_Schedule := Temp_Schedule;
(29)        Found := true;
(30)      end if;
(31)    end if;
(32)  end loop;
End.
```

Ideally, the effectiveness of a heuristic algorithm should be measured by the ratio

$$\frac{\text{success-rate of the heuristic algorithm}}{\text{success-rate of an optimal algorithm}}$$

where success-rate of an algorithm is defined as the ratio

$$\frac{\text{number of feasible schedule found}}{\text{number of prototypes tested}}.$$

Unfortunately, it is impractical, if not impossible, to run the exhaustive-enumeration algorithm on all 2344 prototypes, so we shall only use the success-rate of the heuristic algorithm as a relative measure on the effectiveness of the algorithms. Since the success-rate varies

Table 14. Summary of the prototypes for the uni-processor algorithms.

Group ID	prototypes generated	prototypes accepted	operators count per prototype	edge density	average edge count per prototype	vertices count per constraint graph	average edge count per constraint graph
1	200	169	8	0.1	2.94	24	38.90
2	200	173	8	0.3	8.10	24	82.72
3	200	179	8	0.5	14.07	24	279.73
4	300	286	16	0.1	12.06	48	136.28
5	300	274	16	0.3	36.22	48	390.33
6	300	276	16	0.5	60.32	48	538.41
7	400	324	32	0.1	49.23	96	680.33
8	400	324	32	0.3	148.40	96	1916.45
9	400	324	32	0.5	246.91	96	2307.28

Table 15. Average running time of the uni-processor EDF and ESF algorithms.

	Average Running Time (sec.)								
Group ID	1	2	3	4	5	6	7	8	9
Earliest-Deadline-First	0.16	0.39	0.86	5.02	5.14	9.32	44.48	97.35	169.34
Earlist-Starting-Time-First	0.12	0.33	0.73	4.19	4.93	8.71	40.48	98.87	170.57

significantly for prototypes with large load factors, we further subdivide the prototypes in each group into the five subgroups based on the following load factor ranges: [0.0, 0.6], (0.6, 0.7], (0.7, 0.8], (0.8, 0.9] and (0.9, 1.0]. The success-rate of the two algorithms over the 9 groups of prototypes are shown in Table 16.

The earliest-deadline-first algorithm is very effective in finding feasible solutions for prototypes with load factor 0.7 or below. For prototypes with load factor above 0.7, its success-rate decreases significantly as the complexity of the scheduling constraint graphs increases. While the earliest-deadline-first algorithm may have difficulties in finding the feasible schedules in more complicated graphs, the number of graphs which have feasible schedules also decreases as the graphs become more complicated. Hence, the actual performance of the earliest-deadline-first algorithm could be much better than what the success rate indicates.

The earliest-starting-time-first algorithm also performs quite well for prototypes with load factor 0.7 or below, but its overall performance is worse than that of the earliest-deadline-first algorithm algorithm for large load factors.

Since the schedule produced by the earliest-deadline-first algorithm is also the first schedule examined by both the simulated-annealing algorithm and the limited-backtrack algorithm, the latter two algorithms are at least as effective as the earliest-deadline-first algorithm. In an attempt to find out whether they can really out-perform the earliest-deadline-first algorithm, we tested the two algorithms with the 179 prototypes which the earliest-deadline-first algorithm failed to find feasible solutions in test groups 4, 5 and 6. Each algorithm was allowed to spend up to one hour on each prototype and the Limited-Backtrack algorithm

Table 16. Success-rate of the uni-processor EDF and ESF algorithms.

Group ID	Earliest-Deadline-First Load Factors					Earliest-Starting-Time-First Load Factors				
	[0, 0.6]	(0.6, 0.7]	(0.7, 0.8]	(0.8, 0.9]	(0.9, 1.0]	[0, 0.6]	(0.6, 0.7]	(0.7, 0.8]	(0.8, 0.9]	(0.9, 1.0]
1	**1.00**	**1.00**	**1.00**	0.86	0.94	**1.00**	0.94	0.68	0.41	0.16
2	**1.00**	**1.00**	0.97	0.80	0.39	**1.00**	0.94	0.76	0.52	0.09
3	**1.00**	0.94	0.71	0.58	0.29	**1.00**	0.94	0.55	0.31	0.38
4	**1.00**	**1.00**	**1.00**	0.93	0.77	**1.00**	0.96	0.29	0.07	0.03
5	**1.00**	**1.00**	0.70	0.40	0.00	**1.00**	0.94	0.40	0.00	0.00
6	**1.00**	0.88	0.50	0.15	0.03	**1.00**	0.81	0.26	0.05	0.00
7	**1.00**	**1.00**	0.97	0.68	0.32	**1.00**	0.90	0.17	0.00	0.00
8	**1.00**	0.89	0.37	0.02	0.00	**1.00**	0.70	0.07	0.00	0.00
9	**1.00**	0.71	0.13	0.00	0.00	**1.00**	0.65	0.05	0.00	0.00

was allowed to expand up to four vertices in each backtrack level. As shown in Table 17, both algorithms take a long time to run. Among the 179 prototypes tested, the simulated-annealing algorithm found 7 feasible schedules and was timed out for the remaining 172 prototypes. The limited-backtrack found 6 feasible schedules, was timed out in 84 times, and stopped and returned no feasible schedule for the remaining 87 prototypes.

Table 17. Simulated-annealing and limited-backtrack results.

		Simulated-annealing		Limited-backtrack	
Group ID	prototypes tested	prototypes timed out	feasible schedules	prototypes timed out	feasible schedules
4	12	11	1	8	3
5	75	71	4	52	3
6	92	90	2	24	0

Assuming that a backtrack limit of 4 is sufficient to locate all feasible schedules for prototypes with 16 operators, then we can eliminate the 87 prototypes which the limited-backtrack algorithm reported to have no feasible schedules from the test groups 4, 5 and 6, resulting in the improved success-rates for the earliest-deadline-first algorithm and the earliest-starting-time-first algorithm shown in Table 18, a further indication that the earliest-deadline-first algorithm is very effective for CAPS uni-processor scheduling.

3.3. Multiple Processor Scheduling

Since the next generation of CAPS will run in a multi-processor configuration, we have also extended the earliest-deadline-first, the earliest-starting-time-first, and the limited-backtrack algorithms (Table 19, 20 and 21) to handle multi-processor scheduling for hard real-time systems [3, 7]. The reason for choosing these three algorithms is because they are the most practical ones (in terms of efficiency/effectiveness trade-offs) for the rapid prototyping environment.

The algorithms assume a shared memory, multi-processor configuration (i.e. $S_i = s$, $D_{i,j} = 0$, and $T_{i,j} = 0$). The major difference between single processor scheduling and multiple processor scheduling is that, in addition to deciding which task is to be executed next, the multiple processor scheduling algorithms must decide which processor the task should run on. Given a constraint graph CG and N identical processors, the N-processor schedule S is a N-tuple of linear tables $[S_1, S_2, \ldots, S_N]$ that partitions the vertices in CG into N disjoint sets.

All three algorithms uses two arrays SCHEDULE_ARRAY[1..N] and LAST_STOP_TIME_ARRAY[1..N] to keep track of the vertices assigned to each of the N processors, and the completion time of the last scheduled vertex in each of the processors respectively. They follow the same logic as their uni-processor counter parts in removing vertices from the Ready_Set and the Working_Ready_Set, and always assign the vertices to the processor with the smallest Last_Stop_Time value to minimize the CPU idle time.

Table 18. Adjusted success-rate of the uni-processor EDF and ESF algorithms.

Group ID	Earliest-Deadline-First Load Factors [0, 0.6]	(0.6, 0.7]	(0.7, 0.8]	(0.8, 0.9]	(0.9, 1.0]	Earliest-Starting-Time-First Load Factors [0, 0.6]	(0.6, 0.7]	(0.7, 0.8]	(0.8, 0.9]	(0.9, 1.0]
4	**1.00**	**1.00**	**1.00**	0.93	0.77	**1.00**	0.96	0.29	0.07	0.03
5	**1.00**	**1.00**	0.72*	0.40*	0.00	**1.00**	0.94	0.41*	0.00	0.00
6	**1.00**	0.93*	0.70*	0.60*	0.33*	**1.00**	0.85*	0.34*	0.20*	0.00

* - improved success-rates

Table 19. The Multi-Processor Earliest-Starting-Time-First Algorithm.

```
The Multi_Processor_Earliest_Starting_Time_First Algorithm:
Begin
(1)   Ready_Set := {Dummy};
(2)   Schedule_Array[1..N] := [empty..empty];
(3)   Last_Stop_Time_Array[1..N] := [0..0];
(4)   While Not_Empty(Ready_Set) loop
(5)     i := Id of the processor with smallest Last_Stop_Time value;
(6)     v := Remove_Item_With_Earliest_Start_Time(Ready_Set);
(7)     st(v) := max {Last_Stop_Time_Array[i], ready(v)};
(8)     ct(v) := st(v) + MET(v);
(9)     Add_Item_To_Schedule(v, Schedule_Array[i])
(10)    Last_Stop_Time_Array[i] := ct(v);
(11)    For each child u of v in CG loop
(12)      if all parents of u are in Schedule_Array then
(13)        Add_Item_To_Set(u, Ready_Set);
(14)      end if;
(15)    end loop;
(16)  end loop;
(17)  if Cost(Schedule_Array) = 0 then
(18)    Output "Schedule Found";
(19)  else
(20)    Output "Schedule Not Found";
(21)  end if;
(22)  Output Schedule_Array;
End.
```

3.3.1. Performance Evaluation

All three algorithms have been implemented in Ada. To evaluate their performance, we set N, the number of processors, to 4 and applied the algorithms to a total of 3900 prototypes generated by the PSDL random graph generator developed by Cordeiro [4]. The 3900 prototypes are again made up of 9 groups of expanded dataflow graphs shown in Table 22, and none of the 3900 random prototypes is rejected by the scheduler since they all have load factors less than 4.0.

Like their uni-processor counter parts, both earliest-deadline-first and earliest-starting-time-first algorithms are very efficient, as indicated by the average running time shown in Table 23. The average running time of the Multi-processor EDF and ESF algorithms is actually less than their uni-processor counter-parts. This abnormality can be explained by the fact that the two experiments were conducted under different system loads.

Table 24 shows the success-rate of the two algorithms under different load factors. The earliest-deadline-first algorithm was able to locate 3200 feasible schedules out of the 3900 prototypes tested. Both algorithms perform very well for prototypes with load factors up to 1.6, and then deteriorate as load factors increase above 1.6.

In order to find out whether the decrease in success-rate is caused by the inability of the algorithms in finding feasible schedules or the infeasibility of the prototypes themselves, we apply the limited-backtrack algorithm to the 700 prototypes which the earliest-deadline-

Table 20. The Multi-Processor Earliest_Deadline_First Algorithm.

```
The Multi_Processor_Earliest_Deadline_First Algorithm:
Begin
(1)   Ready_Set := {Dummy};
(2)   Schedule_Array[1..N] := [empty..empty];
(3)   Last_Stop_Time_Array[1 ... N] := [0 ... 0];
(4)   While Not_Empty(Ready_Set) loop
(5)     i := Id of Processor with smallest Last_Stop_Time value;
(6)     Earliest_Deadline_Set := {v | v in Ready_Set and ready(v) <= Last_Stop_Time[i]};
(7)     if Not_Empty(Earliest_Deadline_Set) then
(8)       v := Remove_Item_With_Earliest_Deadline(Earliest_Deadline_Set);
(9)       Ready_Set := Ready_Set − {v};
(10)    else
(11)      v := Remove_Item_With_Earliest_Start_Time(Ready_Set);
(12)    end if;
(13)    st(v) := max {Last_Stop_Time_Array[i], ready(v)};
(14)    ct(v) := st(v) + MET(v);
(15)    Add_Item_To_Schedule(v, Schedule_Array[i])
(16)    Last_Stop_Time_Array[i] := ct(v);
(17)    For each child u of v in CG loop
(18)      if all parents of u are in Schedule_Array then
(19)        Add_Item_To_Set(u, Ready_Set);
(20)      end if;
(21)    end loop;
(22)  end loop;
(23)  if Cost(Schedule_Array) = 0 then
(24)    Output "Schedule Found";
(25)  else
(26)    Output "Schedule Not Found";
(27)  end if;
(28)  Output Schedule_Array;
End.
```

first algorithm failed to find feasible solutions. With a backtrack limit of 4 and a time-out limit of one hour, the limited-backtrack algorithm found 6 feasible schedules, was timed out once, and stopped and returned no feasible schedule for the remaining 693 prototypes.

Again, assuming that a backtrack limit of 4 is sufficient to locate all feasible schedules for prototypes with up to 32 operators, we can eliminate the 693 prototypes which the limited-backtrack algorithm reported to have no feasible schedules from the 3900 prototypes, resulting in the improved success-rates for the earliest-deadline-first algorithm and the earliest-starting-time-first algorithm in Table 25, which shows that the earliest-deadline-first algorithm is very effective for both uni-processor and multi-processor real-time scheduling.

4. Conclusions

This paper presents a collection of algorithms for generating static schedules for the time critical operators in a software prototype. These algorithms solve the general problem of

Table 21. The Multi-Processor Limited-Backtrack Algorithm.

```
Procedure Limited_BackTrack(Backtrack_Limit, Last_Stop_Time_Array,
                            Partial_Schedule_Array, Ready_Set, Best_Schedule_Array):
Begin
(1)   Working_Ready_Set := Ready_Set;
(2)   Found := false;
(3)   Count := 0;
(4)   while not Found and Not_Empty(Working_Ready_Set) and Count < Backtrack_Limit loop
(5)     Count := Count + 1;
(6)     Temp_Schedule_Array := Partial_Schedule_Array;
(7)     i := Id of the processor with smallest Last_Stop_Time value;
(8)     Earliest_Deadline_Set := {v | v in Working_Ready_Set and ready(v) <= Last_Stop_Time_Array[i]};
(9)     if Not_Empty(Earliest_Deadline_Set) then
(10)      v := Remove_Item_With_Earliest_Deadline(Earliest_Deadline_Set);
(11)      Working_Ready_Set := Working_Ready_Set − {v};
(12)    else
(13)      v := Remove_Item_With_Earliest_Start_Time(Working_Ready_Set);
(14)    end if;
(15)    st(v) := max {Last_Stop_Time_Array[i], ready(v)};
(16)    ct(v) := st(v) + MET(v);
(17)    if ct(v) ≤ d(v) then
(18)      Add_Item_To_Schedule(v, Temp_Schedule_Array[i])
(19)      Temp_Ready_Set := Ready_Set − {v};
(20)      Temp_Stop_Time_Array := Last_Stop_Time_Array;
(21)      Temp_Stop_Time_Array[i] := cv(t);
(22)      For each child u of v in CG loop
(23)        if all parents of u are in Temp_Schedule_Array then
(24)          Add_Item_To_Set(u, Temp_Ready_Set);
(25)        end if;
(26)      end loop;
(27)      if Not_Empty(Temp_Ready_Set) then
(28)        Limited_BackTrack(Backtrack_Limit, Temp_Stop_Time_Array,
                              Temp_Schedule_Array, Temp_Ready_Set, Best_Schedule_Array);
(29)        Found := Non_Empty(Best_Schedule_Array);
(30)      else -- no unscheduled vertex
(31)        Best_Schedule_Array := Temp_Schedule_Array;
(32)        Found := true;
(33)      end if;
(34)    end if;
(35)  end loop;
End.
```

Table 22. Summary of the prototypes for the multi-processor algorithms.

Group ID	prototypes generated	prototypes accepted	operators count per prototype	edge density	average edge count per prototype	vertices count per constraint graph	average edge count per constraint graph
1	200	200	8	0.1	2.73	24	39.52
2	200	200	8	0.3	8.50	24	82.33
3	200	200	8	0.5	13.85	24	122.84
4	500	500	16	0.1	12.16	48	137.16
5	500	500	16	0.3	35.78	48	396.96
6	500	500	16	0.5	60.11	48	540.89
7	600	600	32	0.1	49.48	96	793.51
8	600	600	32	0.3	148.78	96	1944.89
9	600	600	32	0.5	248.68	96	2328.68

Table 23. Average running time of the multi-processor EDF and ESF algorithms.

	Average Running Time (sec.)								
Group ID	1	2	3	4	5	6	7	8	9
Earliest-Deadline-First	0.04	0.08	0.14	0.38	2.41	3.61	15.22	73.36	117.07
Earlist-Starting-Time_First	0.03	0.07	0.12	0.33	2.30	3.32	14.43	74.19	116.46

Table 24. Success-rate of the multi-processor EDF and ESF algorithms.

	Earliest-Deadline-First				Earliest-Starting-Time-First			
Group ID	Load Factors [0, 1.2]	(1.2, 1.6]	(1.6, 2.0]	(2.0, 2.4]	Load Factors [0, 1.2]	(1.2, 1.6]	(1.6, 2.0]	(2.0, 2.4]
1	**1.00**	**1.00**	x	x	**1.00**	**1.00**	x	x
2	0.99	0.67	x	x	0.99	0.67	x	x
3	0.91	0.25	x	x	0.91	0.25	x	x
4	**1.00**	**1.00**	0.96	0.83	**1.00**	**1.00**	0.97	0.83
5	0.98	0.79	0.46	0.20	0.98	0.79	0.43	0.20
6	0.89	0.17	0.00	0.00	0.89	0.17	0.00	0.00
7	**1.00**	**1.00**	0.99	0.90	**1.00**	**1.00**	0.99	0.90
8	0.99	0.80	0.32	0.07	0.99	0.78	0.28	0.07
9	0.80	0.06	0.00	0.00	0.80	0.06	0.00	0.00

x - no prototype generated for this case

Table 25. Adjusted success-rate of the multi-processor EDF and ESF algorithms.

	Earliest-Deadline-First				Earliest-Starting-Time-First			
Group ID	Load Factors [0, 1.2]	(1.2, 1.6]	(1.6, 2.0]	(2.0, 2.4]	Load Factors [0, 1.2]	(1.2, 1.6]	(1.6, 2.0]	(2.0, 2.4]
1	**1.00**	**1.00**	x	x	**1.00**	**1.00**	x	x
2	**1.00***	**1.00***	x	x	**1.00***	**1.00***	x	x
3	**1.00***	**1.00***	x	x	**1.00***	**1.00***	x	x
4	**1.00**	**1.00**	0.97*	0.83	**1.00**	**1.00**	0.99*	0.83
5	**1.00***	**1.00***	0.96*	**1.00***	**1.00***	**1.00***	0.91*	**1.00***
6	**1.00***	**1.00***	y	y	**1.00***	**1.00***	y	y
7	**1.00**	**1.00**	0.99	0.90	**1.00**	**1.00**	0.99	0.90
8	**1.00***	**1.00***	**1.00***	**1.00***	**1.00***	0.98*	0.90*	**1.00***
9	**1.00***	**1.00***	y	y	**1.00***	**1.00***	y	y

x - no prototype generated for this case
y - all prototypes are eliminated by the limited-backtrack algorithm
* - improved success-rates

automated pre-run-time scheduling of processes with arbitrary release times, deadlines and precedence relations in hard real-time systems as defined by the PSDL specification. Empirical studies show that the earliest_deadline_first algorithm is very efficient and effective enough to support the rapid prototyping environment provided by CAPS. The limited-backtrack algorithm, on the other hand, is a good complement to the earliest_deadline_first algorithm since it allows user to trade-off between efficiency and effectiveness through a simple backtrack-limit parameter. We have applied the earliest-deadline-first algorithm to several prototypes with 300 time-critical operators (and scheduling constraint graphs with 900 vertices) and it took at most 2 hours to process each prototype. Since the current implementation neither generates the scheduling constraint graph explicitly nor uses efficient data structures to keep track of the parents and children of each operator-instance in the constraint graph, the algorithms spend a lot of time checking the parents of each operator-instance for *no-unscheduled-parent* condition. The algorithms will be able to handle even larger prototypes once we improve their efficiency with the help of additional data structures.

The ability to generate executable schedules automatically is a valuable asset of a rapid prototyping environment. Constructing and fine tuning a static schedule manually is a slow, labor intensive, and error prone process. An alternative manual approach, centralized implementation through an interrupt driven prioritization scheme, produces a dynamic schedule whose effects are difficult to predict and control. The timing requirements are difficult for the user to provide and for the analysts to determine. As the software is modified, various aspects of its execution behavior change, including maximum execution times and execution precedences for the subfunctions. Without automated schedule generation, these changes are often observed only after the fact: the system crashes during testing, or required functions don't get processed when needed. The availability of a non-preemptive static schedule, though conservative, guarantees that all the specified timing requirements will be met even under the worst case situation. Such information is particularly useful at the design level, where many of the timing requirements are being firmed up through prototype simulation.

The results reported in this article also identify several weaknesses and areas which requires improvement within CAPS and PSDL.

(1) More efficient implementation of existing algorithms:

As mentioned earlier, existing algorithms spend a lot of time checking the parents of each operator-instance for *no-unscheduled-parent condition.* The algorithm can be speeded up tremendously if the above operation can be made more efficient with the help of additional data structures.

(2) Interactive Timing Analysis:

Most of the feasibility checks for the prototypes are currently enforced by the scheduler. Such an approach requires the engineers to go through the "edit, save file, then schedule" cycle in order to find out if the timing constraints violate any feasibility constraint. The prototyping process can be made much more efficient and user-friendly if these checks are enforced by the CAPS PSDL syntax-directed editor, where users can detect and receive warnings as they enter the design.

(3) More intelligent execution profiler:

Prototype execution can reveal a lot of information about the dynamic behavior of the design. Current CAPS has a very simple run-time executive which only checks for the violation of deadlines. It will be very beneficial to the designers if the run-time executive can also collect information like how often each operator fires, how often an operator misses its deadline, the average and worst-case tardiness of an operator.

(4) Operators with soft deadlines:

The timing model described in this paper only allows two kinds of operators, time-critical (TC) operators which have hard deadlines and non-time-critical (NTC) operators which have no deadlines. In many real-time systems, there is often a third kind of operators, those with a "soft deadline". Operators with soft deadlines (STC) are of lower priority than those with hard deadlines, but of higher priority than those with no deadlines [22]. Under the current timing model, a NTC operator can starve for a long time before its execution in prototypes with high load factors. The purpose of a soft deadline is to allow the designers to request the system to allocate enough time for the STC operators over a period of time. It is allowable for the STC operators to miss their deadline once a while. But the CAPS run-time executive should issue a warning if the frequency of missing deadlines by a STC operator exceeds some specified threshold.

Acknowledgment

The authors would like to thank Valdis Berzins for helpful suggestions in revising the manuscript.

This research was supported in part by the National Science Foundation under grant number CCR-9058453 and the Army Research Office under grant number ARO 111-95.

Notes

1. An expanded dataflow graph of a PSDL program can be obtained from the top-level dataflow graph that contains the single root node by successfully replacing all composite operators with their decomposition graphs. See [22] for details.
2. That is, we only allow the processors to idle if no operator is available for execution.
3. The edge density is the probability of having an edge between any two vertices in the random graph

References

1. V. Cerny, "A thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm." *Journal of Optimization Theory and Application* 45, pp. 41–45, 1985.
2. J. Cervantes, "An optimal static scheduling algorithm for hard real-time systems specified in a prototyping language," Master's Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, Dec. 1989.
3. T. Chang, "Static scheduler for hard real-time tasks on multiprocessor systems," Master's Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, Dec. 1989.
4. M. Cordeiro, "Distributed hard real-time scheduling for a software prototyping environment," Doctoral Dissertation, Computer Science, Naval Postgraduate School, Monterey, CA, March 1995.
5. B. Fan, "Evaluations of some scheduling algorithms for hard real-time systems," Master's Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, September 1992.
6. B. Flannery, 1984. *Numerical Recipes in C—The Art of Scientific Computing*. Cambridge University Press: New York, NY.
7. L. Hsu, "Multiprocessor scheduling for hard real-time software," Master's Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, June 1990.
8. D. Johnson, C. Aragon, L. McGeoch, and C. Schevon, "Optimization by simulated annealing: An experimental evaluation; Part I, Graph partitioning." *Operations Research* 37, pp. 865–892, 1989.
9. D. Johnson, C. Aragon, L. McGeoch, and C. Schevon, "Optimization by simulated annealing: An experimental evaluation; Part II, Graph coloring and number partitioning." *Operations Research* 38, 1990.
10. S. Kirkpatrick, C. Gelatt, Jr., and M. Vecchi, "Optimization by simulated annealing." *Science* 220, pp. 671–680, 1983.
11. M. Kilic, "Static schedulers for embedded and hard real-time systems," Master's Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, June 1990.
12. B. Kraemer, Luqi, and V. Berzins, "Compositional semantics of a real-time prototyping language." *IEEE Trans. on Software Engineering* SE-19, pp. 453–477, 1993.
13. Luqi, V. Berzins, and R. T. Yeh, "Prototyping language for real-time software." *IEEE Trans. on Software Engineering* SE-14, pp. 1409–1423, 1988.
14. Luqi and V. Berzins, "Rapidly prototyping real-time systems." *IEEE Software* 5, pp. 25–36, 1988.
15. Luqi and V. Berzins, "Semantics of real-time languages," in *Proceedings of the Real-Time Systems Symposium*, Huntsville, AL, 1988, pp. 106–110.
16. Luqi and V. Berzins, "Execution of a high-level real-time language," in *Proceedings of the Real-Time Systems Symposium*, Huntsville, AL, 1988, pp. 69–76.
17. J. Levine, "Efficient static schedulers for the CAPS systems," Master's Thesis, Computer Science Department, Naval Postgraduate School, Monterey, CA, June 1991.
18. Luqi and M. Ketabchi, "A computer aided prototyping system." *IEEE Software* 5, pp. 66–72, 1988.
19. Luqi, M. Shing, P. Barnes, and G. Hughes, "Prototyping hard real-time Ada systems in a slassroom environment," in *Proc. of the Seventh Annual Ada Software Engineering Education and Training (ASEET) Symposium*, Monterey, CA, 1993, pp. 103–117.
20. Luqi, "Handling timing constraints in rapid prototyping," in *Proceedings of the 22nd Annual Hawaii International Conference on System Science*, Kailua-Kona, Hawaii, 1989, pp. 417–424.
21. Luqi, "Computer-aided prototyping for a command-and-control system using CAPS." *IEEE Software* 9, pp. 56–67, 1992.
22. Luqi, "Real-time constraints in a rapid prototyping language." *Computer Language* 18, pp. 77–103, 1993.

23. A. Mok, "Fundamental design problems of distributed systems for the hard real-time environment," Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1983.
24. N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and A. E. Teller, "Equation of state calculations by fast computing machines." *Journal of Chemical Physics* 21, pp. 1087–1092, 1953.
25. S. Nahar, S. Sahni, and E. Shragowitz, "Simulated annealing and combinatorial optimization," in *Proc. 23rd Design Automation Conference*, 1986, pp. 293–299.
26. L. Sha and J. Goodenough, "Real-time scheduling theory and Ada." *IEEE Computer* 23(4), pp. 53–62, April, 1990.
27. A. Stoyenko, V. Hamacher, and R. Holt, "Analyzing hard real-time programs for guaranteed schedulability." *IEEE Trans. on Software Engineering* SE-17, pp. 737–750, 1991.
28. J. Stankovic and K. Ramamritham, 1988. *Tutorial on Hard Real-Time Systems.* IEEE Computer Society Press: Washington, D.C., 1988.
29. J. Stankovic, M. Sprui, M. Di Natale, and G. Buttazzo, "Implications of classical scheduling results for real-time systems." *IEEE Computer* 28, pp. 16–25, 1995.
30. J. Ullman, "NP-complete scheduling problem." *Journal of Computer and System Sciences* 10, pp. 384–393, 1975.
31. J. Ullman, "Complexity of sequence problem," in *Computer and Job-Shop Scheduling Theory.* John Wiley & Sons: NY, 1976.
32. M. Vecchi and S. Kirkpatrick, "Global wiring by simulated annealing." *IEEE Trans. on Computer-Aided Design* CAD-2, pp. 215–222, 1983.
33. H. Wedde, B. Korel, and D. Huizinga, "Static analysis of timing properties for distributed real-time programs." *Real-Time Systems Newsletter* 7, pp. 88–95, 1991.
34. J. Xu and D. Parnas, "Scheduling processes with release times, deadlines, precedence, and exclusion relations." *IEEE Trans. on Software Engineering* 16, pp. 360–369, 1990.
35. S. Zdrzalka, "Scheduling jobs on a single machine with periodic release date/deadline intervals." *European Journal of Operations Research* 40, pp. 243–251, 1989.