# Priority Refinement for Dependent Tasks in Large Embedded Real-Time Software

*Jeffrey R. Merrick, Shige Wang, Kang G. Shin*
*Real-Time Computing Laboratory*
*Department of Electrical Engineering and Computer Science*
*The University of Michigan*
*Ann Arbor, MI 48109-2122*
*email: {jmerrick, wangsg, kgshin}@eecs.umich.edu*

*Jing Song, William Milam*
*Ford Motor Company*
*email: {jsong, wmilam}@ford.com*

## Abstract

*In a large embedded real-time system, priority assignment can greatly affect the timing behavior--which can consequently affect the overall behavior--of the system. Thus, it is crucial for model-based design of a large embedded real-time system to be able to intelligently assign priorities such that tasks can meet their deadlines. In this paper, we propose a priority-refinement method for dependent tasks distributed throughout a heterogeneous multiprocessor environment. In this method, we refine an initial priority assignment iteratively using the simulated annealing technique with tasks' latest completion times (LCT). Our evaluations, based on randomly-generated models, have shown that the refinement method outperforms other priority-assignment schemes and scales well for large, complex, real-time systems. This method has been implemented in the Automatic Integration of Reusable Embedded Software (AIRES) toolkit and has been successfully applied to a vehicle system control application.*

## 1. Introduction

Software for large embedded systems has become complex, containing many software components distributed throughout a multiprocessor environment. At the same time, minimizing the hardware cost is also highly desired. Such a combination of needs creates a situation where careful software design can make a heavily-loaded system function correctly on a resource-limited platform. Recent model-based software development has shown much promise with the ability to abstractly represent the system so that high-level design decisions can be explored and evaluated effectively. In model-based software development, the software architecture is first modeled as interacting components without consideration of the target platform model. It is then translated into a runtime model by strategically allocating the components to the platform and forming OS processes, or tasks, from these software components. As most of today's OSs support only priority-based task scheduling, the thus-formed tasks must be assigned priorities for execution. Once the priorities are assigned for all tasks, the system can be analyzed using real-time scheduling theories to verify the effects of the assignment on meeting the system-level timing constraints.

It has been shown that the problem of allocating software components to computational devices in a platform and the problem of assigning task priorities to meet system-level timing constraints are both NP-hard [1, 3]. Solutions of the component-allocation problem have already been proposed by many researchers [1, 4, 6], and therefore, we assume it has already been solved. In this paper, we focus on the priority assignment after all components have been allocated. The design goal of our priority-assignment method is that it should be able to find a feasible assignment quickly and it should be scalable for a model with a large number of components and interactions. In our approach, we first determine an initial priority assignment for a task based on the latest completion time (*LCT*) of the task with which all of its successors are schedulable. To refine this priority assignment we developed an algorithm based on simulated annealing. The refinement algorithm also uses *LCT* values to constrain the options of assignments and alternates between a random swap of task priorities, which adjusts priorities randomly, and a strategic swap, which adjusts priorities based on analysis. Our evaluations, using a set of randomly-generated software models, show that this method finds a feasible priority assignment quickly and scales well to large systems.

The work reported in this paper was supported in part by DARPA under the US AFRL contracts F30602-01-02-0527 and F3615-00-1706, and by Ford Motor Company under a University Research Partnership grant.

Finally, we apply the method to a vehicle system control used at the Ford Motor Company to demonstrate its effectiveness.

The rest of the paper is organized as follows. Section 2 describes the software architecture and platform models, and formally states the priority-assignment problem. Section 3 defines the *LCT* of a task and discusses the method to calculate and use it in priority assignment. Section 4 presents the simulated annealing algorithm and our heuristics for priority refinement. Section 5 provides an example of applying the method to a vehicle system control. Section 6 shows the evaluation results of our algorithm based on randomly-generated system models. Section 7 discusses related work. The paper concludes in Section 8.

## 2. System Models and Problem Statement

To assign priorities to tasks, the models of both the software architecture and the platform are essential. In this paper we model the software architecture in a *runtime model* and the platform in a *platform model,* which are, respectively, modified versions of the *structural model* and the *platform model* presented in [4]. Informally, a *runtime model* consists of a set of tasks connected by directed links which signify data passing from one task to the next. Each task in the runtime model is allocated to one of the computational devices in the platform for execution. The platform model can informally be defined as a set of computational devices connected by a single shared communication link. The formal definitions of these models are given as follows.

**Definition 1** *A task* $t = (C, I, O, B, e, d, p, pr)$ *is a port-based object where*

- $C$ *is a set of software components [4, 8] which perform certain actions and define the functionality of the task;*
- $I$ *is a set of input ports through which a task receives data and/or is triggered to execute;*
- $O$ *is a set of output ports through which a task sends data and/or triggers other tasks to execute where* $I \cap O = \varnothing$*;*
- $B$ *defines the behavior of the task where* $B \subseteq (E \times I \rightarrow c \times O)$ *where* $E$ *defines a set of events and* $c \subseteq C$*;*
- $e$ *is the worst-case computation resource consumption which is defined by the behavior of* $B$ *and the computation resource consumption of the components in* $C$*;*
- $d$ *is the time by which the task must complete relative to the release time of its input task;*
- $p$ *is the minimum time interval between consecutive invocations of* $t$*;*
- $pr$ *is the priority of* $t$ *and is initially undefined.*

As defined in Definition 1, a task performs a series of computations and actions with the components in *C*. A task's input and output interfaces are defined by the input ports in *I* and output ports in *O,* respectively. The exact behavior of a task, defined by *B*, depends on the inputs received by *I* and the interactions between the components in *C* where this interaction could possibly generate output to *O*. The execution of the components will require a computational resource and the worst case consumption of this resource is defined by *e*. Furthermore, a task must complete its execution by a deadline defined by *d*. In this paper we consider hard real-time systems so we assume the results of missing this deadline could possibly cause a critical failure. We also assume that every task is periodic or is triggered by an outside event that occurs no faster than a certain rate. The period of the task or the minimum time between consecutive invocations is defined by *p*. Finally, we assume that tasks are scheduled based on a fixed priority *pr* which is initially left undefined.

Along with this formal definition a task also has certain characteristics which define properties of a task that are not necessarily captured by its model. The characteristics of a task *t* required in this work include the latest completion time $LCT_t$, the higher relative priority task set $H_t$, and the lower relative priority task set $L_t$. $LCT_t$ is defined as the latest time that *t* can complete so that *t* and all of its successors complete before their deadlines. $H_t$ and $L_t$ are the sets of tasks that have higher and lower relative priorities than *t,* respectively. These three characteristics are detailed in Section 3.

**Definition 2** *A runtime model* $M_R = (T, L_A, L_S)$ *is a weighted directed task graph where*

- $T$ *is a set of nodes for the tasks in* $M_R$*;*
- $L_A \subseteq \cup_{u\in T}O_u \times \cup_{v\in T}I_v$ $(u \neq v)$ *is a set of asynchronous links which may form loops with nodes in* $T$*;*
- $L_S \subseteq \cup_{u\in T}O_u \times \cup_{v\in T}I_v$ $(u \neq v)$ *is a set of synchronous links which may not form loops with nodes in* $T$ *where* $L_S \cap L_A = \varnothing$ *and* $\forall t\in T, \neg(\exists u,v\ (u \neq v)\ u\in(I_t \times L_S) \text{ and } v\in(I_t \times L_S))$.

A *runtime model* defined in Definition 2 is a weighted directed graph where the nodes are tasks and the edges represent data flowing from the output interface of one task to the input interface of another. Data can be passed along an asynchronous link in $L_A$ or a synchronous link in $L_S$. Asynchronous traffic differs from synchronous traffic in that a task is triggered to execute when it receives a synchronous message. However, when an asynchronous message is received it is buffered and then read when the task next executes.

Synchronous links in the *runtime model* also have a few constraints. First, synchronous links should not form cycles in the graph. This is because a cycle in the graph would be viewed as an infinite loop in the software architecture. Cycles in a *runtime model*, describing control functions such as closed-loop feedback and multi-rate control, should be eliminated using techniques such as those described in [8]. Second, each task can have at most one synchronous input link. If a task in the runtime model has more than one synchronous link, it must be converted to one with a single input link. We assume that the relationship of these multiple links is either AND or OR, and thus, can be converted to a single link using techniques for transforming AND [8] and OR [7] relationships. After such transformations a task in the runtime model will either be an input task or will be triggered (directly or indirectly) by a single input task.

A task in this model is either an input task or is triggered by some event that passes along a link in $L_S$. We define an input task as a task that does not have any input synchronous links. Conversely, an output task can be defined as a task that does not have any output synchronous links. An input task is periodically or sporadically released based on a timer or an external event at a rate defined by $p$. For simplicity we assume that all input tasks are initially released at the same time.

In this model a task's successors and predecessors are defined only by the synchronous links. A successor to a task $t$ is then the one that directly or indirectly is triggered for execution by $t$ through synchronous links. Likewise, a predecessor to $t$ is the one that directly or indirectly triggers the execution of $t$ through synchronous links.

**Definition 3** *A platform model $M_P = (P, N, V, R)$ is an undirected graph where*

- *$P$ is a set of computational devices;*
- *$N$ is a single shared communication device for all computational devices in $P$;*
- *$V$ is a mapping of tasks to a single computational device in $P$ given a runtime model $M$ where for all tasks $t \in T_M$, $V \times t = u \in P$;*
- *$R$ is a conversion from communication resource consumption to an equivalent computation resource consumption for $N$.*

A *platform model*, as defined in Definition 3, is an undirected graph where the nodes are the computational devices in $P$ which are all connected by a single link that represents the shared communication device $N$. This definition of the platform model assumes that all tasks have been allocated to a single computational device, which is defined by $V$. $N$ is assumed to pass data at some non-trivial rate $R$ from one computational device to another. With such a rate $R$, we can convert the communication resource consumption for a message from one task to another across $N$ into an equivalent computation resource consumption.

The schedulability test that we run for the task set is a modified version of the classical rate-monotonic scheduling algorithm [9, 11]. The input for the test is a *Timing Analysis Graph* (*TAG*), which is equivalent to the *runtime model* except (i) all tasks are assigned priorities and (ii) there are *concurrent links* between the nodes. A *concurrent link* is an unweighted, directional link. It signifies that the source task has the potential to preempt the destination task. These *concurrent links* are, thus, between two tasks satisfying the following conditions: (i) both are allocated to the same computational device, (ii) one task is not the predecessor (directly or indirectly) of the other, and (iii) the source task has a higher priority than the destination task.

The schedulability test can use the concurrent links along with the synchronous links to accurately determine the response time of each task. This can be done through techniques that consider how many invocations of concurrent tasks could preempt a chain of tasks instead of only considering tasks individually. The test also considers scheduling messages sent from one computational device to another across $N$. This is done by modeling $N$ as a computational device and using $R$ to model each message as a task, as shown in [10].

Such a schedulability test shows several advantages. First, it is less pessimistic on the number of times that a task can preempt another task because it considers the task chains instead of individual tasks. Second, since the tasks' casual relationship is known, if a change is made to the system, a minimal number of affected tasks can be determined according to the casual relationships, and only those need to be rescheduled. A drawback, however, is that this schedulability test incurs overheads. Before the test can be run, a *TAG* must be created. Furthermore, the *TAG* needs to update the concurrent links whenever a task modifies its priority, which can occur frequently.

Using the definitions of the *runtime* model, the *platform model*, and the *TAG* schedulability test, the priority assignment problem can be concisely stated as follows:

> Given a *runtime model MR* and a *platform model MP* where $\forall t \in T_{MR}$ $V_{MP}$ maps $t$ to $P_{MP}$, assign a one-to-one mapping $pr_t \rightarrow \{1, 2, \ldots, |T_{MR}|\}$ such that the *TAG* is schedulable.

## 3. LCT-Based Priority Assignment

Our priority assignment scheme is based on the latest completion time (*LCT*) of each task. The *LCT* of a task is defined as the latest time at which the completion of the task allows all of its successors to meet their timing constraints. In our *LCT*-based priority assignment, tasks' priorities are assigned in a reverse order of their *LCTs* (i.e., a task with a smaller *LCT* is assigned a higher priority).

```
inputs: T: a list of tasks where each $t \in T$ only has
           successors that are output tasks.
        P: a list of all computational devices

while(T is not empty) loop
    $t = Pop(T)$
    for all successors $t_s$ of t loop
        $S_t = MergeSchedule(t, t_s)$
    end loop
    $LCT_t = \min(LCT[p_j] \mid \forall p_j \in P)$
    add time interval for t to $S_t$
    for all predecessors $t_p$ of t loop
        if(all successors of $t_p$ are assigned an LCT) then
            $Push(T, t_p)$
    end loop
end loop
```

**Figure 1: LCT Assignment Algorithm**

The first step of the *LCT*-based priority assignment algorithm is to calculate the *LCT* for each task. The algorithm to calculate tasks' *LCTs* is given in Figure 1.

The input to the algorithm is a list of computational devices and a list of tasks whose successors are output tasks. The task list represents the tasks that are ready to have their *LCT* assigned. A task is ready to have its *LCT* assigned only when all of its successors have their *LCTs* assigned. Furthermore, the *LCT* of an output task is trivially assigned as its deadline since it does not have any successors. At each step, the algorithm takes a task ready for schedule, and merges it to the schedule of all its successors. Such a schedule thus determines the execution order of the task and its successors. The *LCT* for the task can then be found by taking the minimum *LCT* for each computational device, where the *LCT* for the task on a computational device, denoted $LCT[p_j]$ for some $p_j \in P$, is defined to be the start time of the first task in the schedule on that computational device. Once the *LCT* has been calculated, the start time for the task can be found and the task can be added to the schedule. The next step is to check if any of the task's predecessors are ready to have their *LCT* calculated. Since the *LCT* of a task depends on the *LCT* and schedule of its successors, only those tasks, whose successors' *LCTs* and schedules have been assigned, can have their *LCT* calculated. This process should be repeated until all tasks have a *LCT* assigned.

The core of the *LCT* calculation is the *MergeSchedule* function given in Figure 2. The inputs to this function are two tasks $t_1$ and $t_2$ which have corresponding schedules $S_1$ and $S_2$. The output of the function is the schedule corresponding to $S_1$ merged with $S_2$. A schedule, *S*, is defined to have individual schedules for each processor denoted by $S[p]$ for some processor *p*. Each individual schedule is an ordered, non-overlapping list of time intervals where a time interval is simply a time where the processor would be busy. A time interval has a start time, *st*, and a completion time, *ct*. The schedule for an output task, *t*, is defined to be a single time interval for the processor which it is allocated to that has a *ct* equal to $d_t$ and an *st* equal to $d_t - e_t$. All other tasks begin with an empty schedule.

```
inputs:  $t_1$, $t_2$: two tasks to have their schedules merged
outputs: S: the merged schedule

for all processors $p_j \in P$ loop
    time block $b_{last} = \varnothing$
    time block $b_1 = PopBack(S_1[p_j])$
    time block $b_2 = PopBack(S_2[p_j])$
    while($S_1[p_j]$ and $S_2[p_j]$ are not empty) loop
        if($S_2[p_j]$ is empty or $b_{1.ct} > b_{2.ct}$) then
            time block $b_{max} = b_1$
            $b_1 = PopBack(S_1[p_j])$
        else
            time block $b_{max} = b_2$
            $b_2 = PopBack(S_2[p_j])$
        end if-else
        if($b_{max.ct} > b_{last.st}$) then
            $b_{max.st} = b_{last.st} - (b_{max.ct} - b_{max.st})$
            $b_{max.ct} = b_{last.st}$
        end if
        $Push(S[p_j], b_{max})$
        $b_{last} = b_{max}$
    end loop
end loop
return S
```

**Figure 2: MergeSchedule Function**

The *MergeSchedule* function begins by looping through all of the processors and merging all of the schedules on each individual processor. This is done by first removing the last time block from each of the schedules. These two time blocks are compared to find which one has a later completion time, $b_{max}$, and removes the last time block from the corresponding schedule so it can be used on the next loop. If $b_{max}$ has a *ct* that is before the *st* of the last block that was added to the schedule, $b_{last}$, then these two time blocks do not overlap and $b_{max}$ can be added to the schedule. However, if they do overlap $b_{max}$ needs to be shifted such that $b_{max.ct}$ is equal to $b_{last.st}$ before it can be added to the schedule. This process is then repeated until all of the time blocks from the two schedules have been added to the merged schedule.

One observation of the *LCT*-based priority assignment is that if all tasks are independent, then it is equivalent to deadline monotonic scheduling and, thus, is optimal under the conditions that make deadline monotonic scheduling optimal. However, it is also true that under certain conditions *LCT* priority assignment is optimal for a task set that includes dependent tasks.

One case that the *LCT*-based priority assignment is optimal for a dependent task set *T* is when all $t \in T$ are allocated to the same computation device. This is because the task precedence constraints can be ignored in this case since such constraints will be enforced by the task priorities. Since the *LCT* of a task is dependent on the schedules and *LCTs* of its successors, the *LCT* of a task must be less than that of its successors. Thus, the priority of a task will always be higher than that of its successors. Therefore, the task set can be viewed as an independent task set and since a task must complete by its *LCT* for all of its successors to be able to meet their deadlines the *LCT* can be viewed as the task's deadline. Under these assumptions there is no difference between *LCT* priority assignment and deadline monotonic scheduling. Furthermore, since deadline monotonic scheduling has been proven to be optimal for independent tasks, *LCT* priority assignment must also be optimal.

Another observation is that if all tasks have the same period *LCT*-based priority assignment is very likely to be the optimal priority assignment for the task set. This is intuitive because tasks can directly compare *LCT* values. The reason that two tasks with the same period can compare *LCT* values directly is because a task's *LCT* is relative to the release time of the task's input task. Since we assume that all input tasks have the same phasing all input tasks with the same period will always be released at the same times. However, if two tasks have different periods their *LCTs* most likely will not be based on the same release time so it may appear that one task can complete later than the other when this might not actually be the case.

However, if all tasks do not have the same period or not all tasks are located on the same computational device, there may be another priority assignment algorithm that will find a solution better than that of the *LCT* priority assignment. We propose that in such cases it is advantageous to find groups of tasks where all of the tasks in each of these groups satisfy one or both of these conditions. Groups can be formed between tasks that (i) all have the same period or (ii) where all of the tasks in the group and all of their predecessors (direct or indirect) are allocated to the same computational device. The tasks in group (ii) satisfy the condition that all tasks are on the same computational device and it can be viewed that all tasks in this group are released at the same time.

Since these groups satisfy one of the conditions, we can assign priorities to these tasks relative to the other tasks in the group based on their *LCTs*. These relative *LCT* priorities can be used as constraints when assigning the global priorities, thus greatly reducing the number of possible priority assignments. We use these constraints to define $H$ and $L$ in the task model in Section 2. $H$ can be defined for task $t$ to be the set of all tasks with a higher relative *LCT* priority than $t$. Similarly, $L$ can be defined for task $t$ to be the set of all tasks with a lower relative *LCT* priority than $t$.

## 4. Priority Refinement Using Simulated Annealing

Our priority-refinement algorithm uses the relative *LCT* constraints in simulated annealing to quickly find a feasible schedule. Simulated annealing is a general global optimization algorithm. It attempts to find the lowest point of energy in an energy landscape, where this point corresponds to the best solution for the given problem. This technique has been used successfully for many different optimization problems that have been shown to be NP-complete including task allocation and priority assignment by Tindell *et al* [1]. Differently, we use simulated annealing only for priority assignment since we have assumed that all tasks have already been allocated to processors.

The general simulated annealing algorithm is fairly simple. An initial starting point, $p$, is chosen as a possible solution and the energy at this point, $E_p$, is evaluated where the lower the energy the better the solution. A neighbor, $t$, of $p$ is chosen as an alternative solution and the energy at this point, $E_t$, is evaluated. If $E_t < E_p$ then $t$ is chosen as the next $p$ otherwise $t$ is chosen as the next $p$ with a probability of $e^x$ where $x = (E_p - E_t) / c$ and $c$ is some control temperature. This algorithm repeats periodically decreasing $c$ until a stopping criterion is met.

In our refinement algorithm, a point, $p$, corresponds to a priority assignment to all tasks. A neighbor to $p$ is a different priority assignment where two tasks have swapped priorities. The energy of a point is defined to be:

$$E = \sum_{\forall t \in T} \max(0, resp_t - d_t)$$

where $resp_t$ is the response time of $t$ for the given priority assignment. This definition of $E$ directs the simulated annealing algorithm search for an assignment that reduces the total time that the task set misses its deadlines by. The initial value of the control temperature $c$ can be chosen in many different ways. Our selection is based on the energy of the original point $p$. This results in almost all neighboring points originally being accepted. $c$ is periodically reduced after a number of steps so that after each reduction a neighboring point corresponding to a worse priority assignment has less of a probability of being accepted.

Incorporating the *LCT* relative priority constraints in the simulated annealing algorithm restricts which tasks are allowed to swap priorities. If a task $t$ would have its priority raised then the swap is allowed only if the new priority is not higher than that of the lowest priority task in $H_t$. Likewise, if a task would have its priority lowered then the swap is allowed only if the new priority is not less than that of the highest priority task in $L_t$. This ensures that a task cannot raise its priority above another

```
inputs:  $T_{MD}$: the set of tasks that do not meet their
         deadlines
outputs: a pair of tasks to swap priorities with or $\varnothing$ if no
         valid pair is found

t = random task from $T_{MD}$
P = list of all direct predecessors of t
while(P is not empty) loop
    $i = pr_t + 1$
    while($i < pr_{tl}$, tl = lowest priority task in $H_t$) loop
        if(t and task with priority i is a valid swap) then
            return t and task with priority i
        $i = i + 1$
    end loop
    for(all tasks tp that preempt t) loop
        $i = pr_{tp} - 1$
        while($i > pr_{th}$, th = highest priority task in $L_{tp}$) loop
            if(tp and task with priority i is a valid swap)then
                return tp and task with priority i
            $i = i + 1$
        end loop
    end loop
 t = Pop(P)
PushBack(P, all direct predecessors of t)
end loop
return $\varnothing$
```

**Figure 3: StrategicSwap Function**

task that has a higher relative priority and a task cannot lower its priority below another task with a lower relative priority. In other words, a task can only swap priorities with another task where there is no relative priority relationship between the two and the priorities of each task are within the correct ranges.

Since it chooses a neighboring point to evaluate randomly, the classical simulated annealing algorithm does not predict what might be a ‘good’ neighbor. Although such randomness helps to keep the algorithm from getting stuck at local minimums, it may also cause the algorithm to take a long time to find a solution due to blindly searching through the many possible neighboring points. We would like to take advantage of the randomness of the classical simulated annealing algorithm while also accelerating it by guiding it to ‘good’ neighboring points. This is achieved by alternating between a random swap of priorities and a strategic swap of priorities.

The strategic swap that we develop is based on the fact that a task *t* has a better probability of completing before its deadline if either: (i) its priority is raised, (ii) a task that is preempting it has its priority lowered, or (iii) one of the task’s predecessors has its priority raised. Although this does not guarantee that the system as a whole will yield a better schedule, it gives *t* a better chance at meeting its deadline. The strategic swap function that

```
inputs:  T: task set
outputs: Success or Failure depending if a valid priority
         assignment was found

Calculate LCTs for all tasks in T
Assign LCT priorities
Determine H and L for all tasks in T
Choose a starting temperature C
Assign initial priorities to all tasks in T
if(Schedulability test returns Success) then
    return Success
loop
    loop
        E = energy for current priority assignment
        if(random swap was performed last) then
            $t_i$, $t_j$ = StrategicSwap()
        if(strategic swap was performed last pass or
        StrategicSwap() just returned $\varnothing$) then
            $t_i$, $t_j$ = RandomSwap()
        Swap the priorities of $t_i$ and $t_j$
        if(Schedulability test returns Success) then
            return Success
        $E_{swap}$ = energy for the swapped priority assignment
        if($E_{swap} \geq E$) then
            $x = (E - E_{swap}) / C$
            if($e^x < random(0, 1)$) then
                Swap back the priorities of $t_i$ and $t_j$
        end if
    while(thermal equilibrium has not been reached)
    C = update C
while(some stopping criterion has been met)
return Failure
```

**Figure 4: Simulated Annealing Priority Assignment**

attempts to perform one of these actions is given in Figure 3.

The entire priority-assignment algorithm using simulated annealing, *LCT* relative priority constraints, and alternating between the random priority swap and the strategic priority swap is given in Figure 4. This algorithm begins by calculating *LCT* values for all of the tasks and using these values to assign *LCT* priorities for each task. Tasks are formed into groups where relative priorities can be assigned and *H* and *L* are determined for each task. An initial control temperature is chosen and each task is assigned an initial priority assignment (e.g., by deadline monotonic or *LCT* priority assignment). If this priority assignment yields a successful schedule then a success can be returned right away. Otherwise, the energy, *E*, is calculated for the current priority assignment. The priorities of two tasks are then swapped either randomly or through the strategic swap function depending on what was performed in the previous step. The schedulability test is run for this priority assignment at each step. If a successful schedule has been found, the function returns a success. Otherwise, the energy of the new priority assignment, $E_{swap}$, is computed and compared with $E$. If $E_{swap}$ is less than $E$, it indicates a better schedule has been found, and the priority swap is accepted. Otherwise, the swap is accepted with a probability of $e^x$ where x = ($E$ - $E_{swap}$) / $C$ and if the swap is not accepted then the priorities are swapped back to their original values. This process repeats until it is determined that thermal equilibrium is reached where $C$ is updated accordingly. The algorithm continues until some stopping criterion has been met (e.g., the number of steps without lowering $E$ has crossed a threshold) where it can be said that the algorithm failed to find a solution.

## 5. Example: Vehicle System Control

To show the effectiveness of our priority refinement algorithm, we applied it to a vehicle system control (VSC) application used by the Ford Motor Company. Vehicle system control is the high-level control that coordinates different subsystems in a vehicle. For example, in a hybrid gas/electric vehicle one function of the VSC may determine the operating mode of the gas- powered engine or the battery-powered motor. The structure of the VSC application is given in Figure 5 where the black links indicate synchronous messages and the gray links indicate asynchronous messages. The model contains components for sensor system control (Ssc), actuator system control for the engine module (AscEm), actuator system control for the transmission module (AscTm), vehicle system control for driver / traffic / vehicle / coordination input fusion (VscDif, VscTif, VscVif, and VscCif respectively), and vehicle system control for driver / traffic / vehicle / coordination layer (VscDl, VscTl, VscVl, and VscCl, respectively). This is a simplified version of the application where the actual model had over 40 components in multiple layers of hierarchy. The platform consists of two computational devices $P_1$ and $P_2$ connected by a shared link $N$. Using the allocation algorithm given in [4] Ssc, AscEm, AscTm, and VscCl are allocated to $P_1$ and VscDif, VscTif, VscVif, VscCif, VscDl, VscTl, and VscVl are allocated to $P_2$. For simplicity we do not consider link scheduling in this example.

In this example the input tasks are Ssc and VscCif which are triggered by timers at 100 ms and 110 ms respectively (these times and those in Figure 6 have been slightly modified to highlight the different features of the priority refinement algorithm). The successors to these two tasks form two different groups where the relative *LCT* priorities can be assigned since each member of these groups has the same rate of invocation. The algorithm begins by determining *LCT*, *H*, *L*, and *pr* for

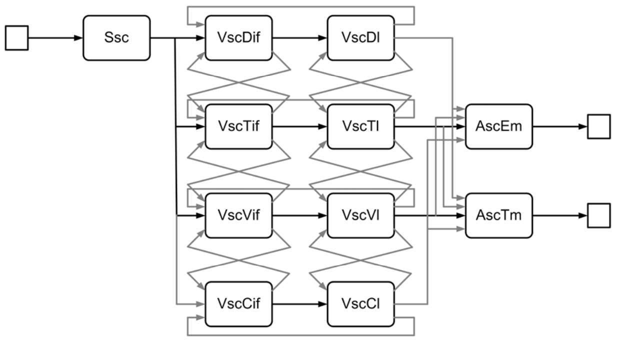

**Figure 5: Structure of vehicle system control application.**

each task and determining if this priority assignment passes the schedulability test. These results are given in Figure 6 where it can be seen that VscDl misses its deadline by 5 ms.

The priority refinement algorithm begins by randomly selecting two tasks to swap priorities which turn out to be VscCif and VscDif. This swap is a legal swap since the new priorities for both VscCif and VscDif are still in the proper ranges given their respective *H* and *L*. As shown in Figure 7, this priority swap does not affect the response times of any tasks positively or negatively. This results in an energy value, $E_{swap}$, that is equivalent to the previous energy value, $E$. In the case that the two energy levels are the same the simulated annealing algorithm always accepts the swap so the tasks switch priorities.

The second step is to choose two tasks to swap priorities using the strategic swap. First, the algorithm tries to increase the priority of VscDl, but this cannot be done since this would raise the priority to be greater than VscDif which would violate the relative *LCT* priority constraints. Next, the algorithm tries to increase the priority of the predecessor of VscDl, VscDif. This would swap the priorities of VscDif and VscCif, which is a legal swap, and since this is a reversal of the first step we are back to the original priority assignment.

The third step in the algorithm chooses a valid random swap between VscCl and AscTm. This swap again does not affect the response times of the tasks and results in an equivalent energy value.

The fourth step of the algorithm again tries to increase the priority of VscDl through the strategic swap function. However, since the previous strategic swap increased the priority of VscDif VscDl can now increase its priority without violating its relative *LCT* constraints. The increase in priority for VscDl results in it finishing by 90 ms and increases the response time of VscCl to be 110 ms, both of which are at or below their respective deadlines. At this point the algorithm terminates since all tasks have a worst case response time that is earlier than their respective deadlines.

| task | $d/p_t$ | $e_t$ | $LCT_t$ | $pr_t$ | $resp_t$ | $H_t / L_t$ |
|---|---|---|---|---|---|---|
| Ssc | 100 | 10 | 20 | 11 | 10 | {-} / { VscTif, VscVif, VscTl, VscVl, VscDif, VscDl, AscEm, AscTm} |
| VscDif | 100 | 10 | 90 | 6 | 80 | {Ssc, VscTif, VscVif, VscTl, VscVl} / {AscTm, AscEm, VscDl} |
| VscDl | 100 | 10 | 100 | 4 | 105 | {Ssc, VscTif, VscVif, VscTl, VscVl, VscDif} / {AscTm, AscEm} |
| VscTif | 100 | 10 | 60 | 10 | 20 | {Ssc} / {AscTm, AscEm, VscDl, VscDif, VscVl, VscTl, VscVif} |
| VscTl | 100 | 20 | 80 | 8 | 50 | {Ssc, VscTif, VscVif} / {AscTm, AscEm, VscDl, VscDif, VscVl} |
| VscVif | 100 | 10 | 60 | 9 | 30 | {Ssc, VscTif} / {AscTm, AscEm, VscDl, VscDif, VscVl, VscTl} |
| VscVl | 100 | 20 | 80 | 7 | 70 | {Ssc, VscTif, VscVif, VscTl}/ {AscTm, AscEm, VscDl, VscDif} |
| VscCif | 110 | 25 | 90 | 5 | 90 | {-} / {VscCl} |
| VscCl | 110 | 10 | 110 | 1 | 100 | {VscCif} / {-} |
| AscEm | 100 | 20 | 100 | 3 | 70 | {Ssc, VscTif, VscVif, VscTl, VscVl, VscDif, VscDl} / {AscTm} |
| AscTm | 100 | 20 | 100 | 2 | 95 | {Ssc, VscTif, VscVif, VscTl, VscVl, VscDif, VscDl, AscEm} / {-} |

**Figure 6: Initial priority assignment configuration for Vehicle System Control.**

| step | $E$ | swap tasks | $E_{swap}$ | $C$ | $e^x$ | rand(0,1) | $T_{MD}$ |
|---|---|---|---|---|---|---|---|
| 1 | 5 | VscCif, VscDif - *Random* | 5 | 10 | 1 | ..676 | VscDl |
| 2 | 5 | VscDif, VscCif - *Strategic* | 5 | 10 | 1 | .197 | VscDl |
| 3 | 5 | VscCl, AscTm - *Random* | 5 | 10 | 1 | .231 | VscDl |
| 4 | 5 | VscDl, VscCif – *Strategic* | 0 | - | - | - | - |

**Figure 7: Steps for refining priority for Vehicle System Control.**

## 6. Evaluation

Our evaluation focused on how well the priority assignment algorithm performed for varying the average utilizations of the computational devices. The performance metrics that we used to evaluate this were the average sum of excessive execution time after deadline (i.e. the average final value of *E* in the simulated annealing algorithm) and the failure rate. In the experiments, we terminated the simulated annealing algorithm if there was no improvement on the overall schedulability of the task set over 150 consecutive steps. This prevents the algorithm from repeating infinitely for task sets where no feasible priority assignment exists. The failure rate is then simply the number of experiments where a solution was not found over the total number of experiments performed. The average sum of excessive execution time after deadline can be used to determine which priority assignment schemes yielded a better schedule among those tests that fail. Thus, we can determine which algorithm was closer to finding a solution and, thus, more likely to find a solution given another task set with the same attributes.

To perform our experiments we used a set of randomly generated models. In the task graphs 100 tasks were generated each of which had a link output degree of 1 to 5. Each of the links that connected the tasks could be either synchronous or asynchronous. The number of bytes that was passed along each of these links varied randomly from 10 to 200 bytes. The computational resource consumption for each task was set so that the average utilization of each computational device varied between .5 and 1 in increments of .05. The platform model consisted of 5 computational devices all connected by a single communication device. Each of the tasks is randomly allocated to one of these computational devices with an attempt to balance the computational loads.

To evaluate our modified simulated annealing priority assignment algorithm, we compared it with a selected set of other priority assignment algorithms. The baseline algorithm that we tested had an initial priority assignment using deadline monotonic (with deadlines assigned using a deadline distribution algorithm) and standard simulated annealing was used for adjusting the priority (DM/SA). The next priority assignment algorithm tested used the LCT priority assignment for the initial priority assignment along with standard simulated annealing for refinement (LCT/SA). We then added to the previous test the LCT priority constraints to determine valid priority swaps (LCT/SA+LCT). The final test is our proposed algorithm which adds the strategic swap alternated with the random swap (LCT/SA+LCT+SS).

In our experiments, we varied the average utilization of the computational devices from .5 to 1 with increments of 0.05. This range was chosen because initial experiments showed that if the average utilization was below 0.5 almost all tests found a solution often right away with the initial priority assignment. Likewise, once the average utilization gets close to 1 almost all tests fail. However, we can still get meaningful data from these tests by seeing how close they were to finding a solution.

Figure 8 shows the failure rate for the different priority assignment algorithms. The DM/SA and the LCT/SA algorithms performed very similarly and had the highest failure rates. The similarity between these two tests is not

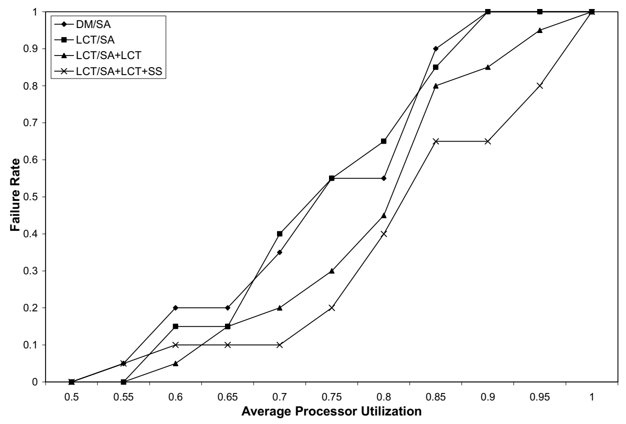

Figure 8: Failure rate for different average processor utilizations.

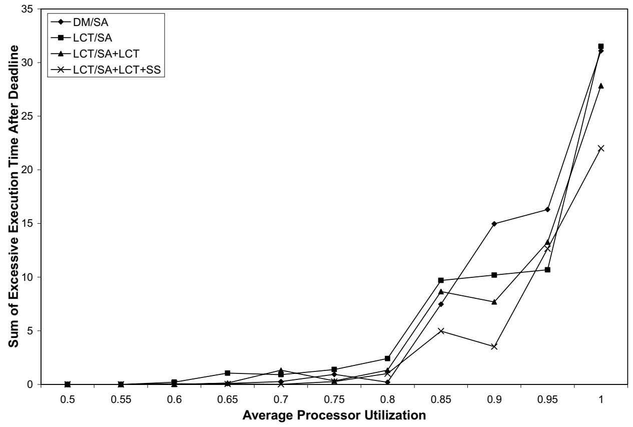

Figure 9: The average sum of excessive execution time after deadline for different average processor utilizations.

surprising since the only difference between the two is the initial priority assignment. The LCT/SA+LCT priority assignment algorithm had the next lowest failure rate showing that the *LCT* relative priority constraints help to move towards a better solution. Among all the experiments, our priority assignment algorithm (LCT/SA+LCT+SS) resulted in the lowest failure ratio. When compared with the DM/SA priority assignment algorithm, LCT/SA+LCT+SS on average found a successful solution 1.8 times as often.

Figure 9 shows the average sum of excessive execution time after deadline. The DM/SA and LCT/SA algorithms performed similarly and missed the tasks' deadlines by the most. Again, this is due to the only difference being their initial priority assignment. The LCT/SA+LCT priority assignment algorithm missed its deadlines by the next lowest amount followed by LCT/SA+LCT+SS. It can also be seen that DM/SA misses its deadlines on average by 3.4 times as much as our LCT/SA+LCT+SS.

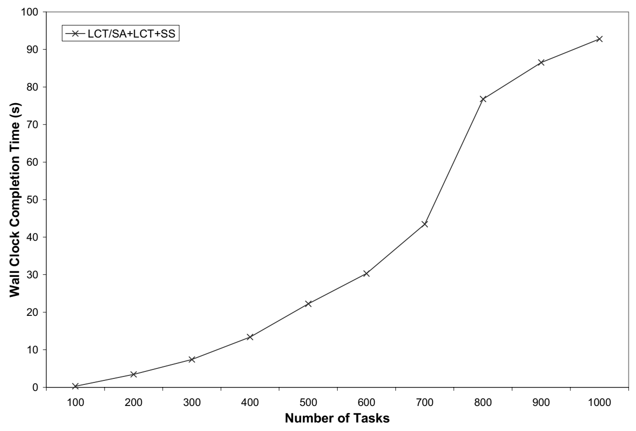

Figure 10: Wall clock time for algorithm to complete.

Finally, we wanted to show that our algorithm is scalable for large systems consisting of a high number of tasks. For this we randomly generated systems varying the number of tasks from 100 to 1000 and allocated them to 5 computational devices each with an average utilization of 0.7. The LCT/SA+LCT+SS algorithm was run 20 times for each case and the average wall clock time that the algorithm took to complete was recorded. The results of this test are given in Figure 10 where the tests were run on a PC using a 1.5 GHz Intel Pentium 4 processor with 256 MB of RAM using the Windows 2000 Professional Operating System. This graph shows that the completion time of the LCT/SA+LCT+SS algorithm increases approximately linearly with an increase in the number of tasks and, thus, scales well to large task sets.

## 6. Related Work

Priority assignment is a classic problem that has been the focus of much research. Liu and Layland introduced rate monotonic scheduling [9] and this was extended to deadline monotonic scheduling by Leung and Whitehead [12]. These well-known approaches have been shown to be optimal under certain assumptions one of which is that the task set is independent. However, if the task set is comprised of dependent tasks the problem becomes NP-complete [6]. Heuristic based algorithms to solve this problem have been given in [2, 3] but these approaches only consider *simple transactions* (i.e., the transactions do not revisit a processor or create subtransactions). Wu *et al* present another priority assignment algorithm in [5], but it enforces the precedence constraints by setting deadlines for each "row" of tasks which introduces extra unnecessary constraints. The problem of task allocation combined with priority assignment has also been considered [1, 6]. However, this approach combines two NP-complete problems into one [1, 3] and in doing so makes assumptions that oversimplify one or both of the problems.

Our approach differs from all of these previous approaches in that it considers the precedence constraints of tasks while minimizing the number of assumptions that need to be made. Furthermore, we consider the task allocation and priority assignment separately so that each problem can be considered with as much detail as is necessary.

## 7. Conclusions and Future Work

In model-based embedded software design and automation, priority assignment is a crucial step in finding a solution that can meet all of system timing constraints. If the priorities are assigned improperly, the software may not behave as desired, which could lead to a complete system failure. To solve the priority-assignment problem, we have developed a method to iteratively refine the task priorities. The method first finds the latest time at which a task can complete such that all of its successors meet their deadlines. The *LCT* is then used to assign initial priorities and assign relative priorities to task groups. We then use simulated annealing to adjust task priorities for a system that is not schedulable. The adjustment uses the relative priority constraints that can be found between tasks that have certain similar characteristics. Furthermore, since we know how to modify the priority of a task to increase likelihood of it being schedulable, we introduce a strategic swap method. To find a global minimum instead of a local one, we alternate between the strategic swap and the original random swap in the simulated annealing algorithm. Our evaluation results show that this algorithm performs better not only by increasing the number of feasible assignments found but also by decreasing the overall time past deadlines. Furthermore, the approach also scales well in the systems with many tasks and can find solutions to the systems with very high workloads.

There are many interesting avenues that can be taken for future work. First, we would like to explore other constraints that may be put on the priority of a task so that the search space may be reduced even further. Second, we would like to apply the priority constraints to other incremental priority-assignment algorithms and compare their results with those generated by simulated annealing. Finally, we would like to incorporate other design refinement decisions, such as task formation and component allocation, to construct a fully-automated design process.

## References

[1] K.W. Tindell, A. Burns, and A. J. Wellings, “Allocating Real-Time Tasks. An NP-Hard Problem Made Easy”, In *Real-Time Systems Journal*, Vol. 4, No. 2, May 1992.

[2] J. G. Garcia and M. G. Harbour, “Optimized Priority Assignment for Tasks and Messages in Distributed Hard Real-Time Systems”, In *3rd Workshop on Parallel and Distributed Systems*, IEEE Computer Society Press, April 1995, pp. 124-131.

[3] R. Etemadi, G. Karam, and S. Majumdar, “Heuristic Algorithms for Priority Assignment in Flow Shops”, In *Performance, Computing and Communications*, IEEE International, February 16-18, 1998, pp. 15-22.

[4] S. Wang, J. Merrick, and K. G. Shin, “Component Allocation with Multiple Resource Constraints for Large Real-Time Embedded Software Design”, In *Proceedings of the Tenth IEEE Real-Time and Embedded Technology and Applications (RTAS 2004)*, Toronto, Canada, May 25-28, 2004, pp. 219-226.

[5] B. Wu, C. Peng, W. Qiu, and X. Sun, “Component Priority Assignment in the Data Flow Dominated Embedded Systems with Timing Constraints”, In *The 7th International Conference on Computer Supported Cooperative Work In Design*, 2002, pp. 385-388.

[6] D. A. L. Piriyakumar and C. S. R. Murthy, “Optimal Scheduling of Parallel Tasks of Tracking Problem Onto Multiprocessors”, In *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 32, Iss. 2, April 1996, 722-731.

[7] C. M. Krishna and K. G. Shin, *Real-Time Systems*, McGraw-Hill Companies, Inc., New York, 1997.

[8] S. Wang and K. G. Shin, “Early-Stage Performance Modeling and Its Application for Integrated Embedded Control Software Design”, In *Proceedings of the Fourth International Workshop on Software and Performance*, Redwood Shores, California, January 14 - 16, 2004, pp. 110-114.

[9] C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment”, In *Journal of the ACM*, Vol. 20, Iss. 1, 1973, pp 46-61.

[10] K. W. Tindell, H. Hansson, and A. J. Wellings, “Analysing Real-Time Communications: Controller Area Network (CAN)”, In *Proceedings of the 15th Real-Time Systems Symposium*, San Juan, Puerto Rico, December 7-9, 1994, pp 259-263.

[11] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, “Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling”, *Software Engineering Journal*, Vol. 8, Iss. 5, September 1993, pp 284-292.

[12] J. Y. T. Leung and J. Whitehead, “On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks”, In *Performance Evaluation*, Vol. 2, No. 4, pp. 46-61, December 1982.