University of Massachusetts, Department of Computer Science, Technical Report UM-CS-2000-064

Exploiting Hierarchy for Planning and Scheduling

Aaron G. Cass **Krithi Ramamritham** **Leon J. Osterweil**

Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610
{acass, krithi, ljo}@cs.umass.edu

ABSTRACT

In real-time systems, tasks must satisfy strict timing constraints in order to avoid timing failures. In situations in which task running times are not known *a priori*, i.e., at design time, dynamic scheduling approaches attempt to guarantee, at run time, that tasks will meet their deadlines. Clearly, such guarantees may not always be possible, and so schedulability failures are likely. One option to minimize the effect of such failures is to instead schedule alternative tasks. In this paper, we present and evaluate an approach that exploits hierarchically structured program representations that naturally occur in many application scenarios. Hierarchy allows us to (a) simplify the specification of real-time requirements and (b) specify the alternative tasks to schedule in the event of an impending timing failure. We extend an existing dynamic scheduling algorithm to deal with hierarchical task structures and present the results of our experimentation aimed at determining the efficacy of our approach.

1 INTRODUCTION

In real-time systems, tasks must satisfy strict timing constraints in order to avoid timing failures. In cases in which task running times are not known *a priori*, dynamic scheduling approaches attempt to guarantee, at run time, that all tasks will meet their deadlines. However, this guarantee may not always be possible, and timing failures are likely. In many applications, when this occurs, it might be acceptable to schedule lower-quality alternative versions for certain (sets of) tasks, where the alternatives are designed to consume less processing time. Scheduling such alternatives is one way to reduce the likelihood of (total) timing failures in dynamic systems. Consider, for example, an unexpected emergency situation in a computer controlled process. Resolving this might require obtaining additional information from a set S of sensors within a short time window. Ideally, we will want to obtain all the readings from the sensors in S but it is highly likely that readings from a subset of the sensors will also prove useful.

The specification of these alternatives, as well as the run time approaches used to both choose the best alternatives and produce schedules for those alternatives, can vary. In this paper, we present a flexible, generalized hierarchical approach to specifying the alternatives. In addition to the standard software engineering arguments in favor of hierarchy as a way of providing abstraction and complexity control, we have found that hierarchy has some scheduling benefits. We exploit hierarchical program representations to (a) simplify specification of real-time requirements and (b) specify alternative sets of tasks used to avoid timing failure. We use a scoped exception handling mechanism to limit the likelihood of exposing timing failures. Our approach is very general in the character of possible alternatives and also in the handling of timing failure experienced by local actions. Because our approach is based on a scoped exception handling mechanism, scheduling failure can be handled by any number of remedies, and the remedies themselves might fail to produce a valid schedule and so they might have to be remedied as well. This chaining of possible solutions to scheduling failure allows us not only to specify different remedies, but to specify in what order they should be attempted.

In the next section, we discuss related work on approaches to this and similar classes of problems. We follow with a formal statement of the class of problems in which we are interested, including a description of the hierarchical program structures we wish to schedule. Then, in Section 4 we present our approach, which extends an existing heuristic search based dynamic scheduling algorithm. We then present in Section 5 experiments we have conducted to determine for what types of problems this approach is profitable. We conclude with a discussion of what has been learned in this work, and possible future directions.

2 RELATED WORK

Liestman and Campbell [5] present an off-line approach to a similar problem. They introduce primary and alternate algorithms for computing solutions of differing quality and timing characteristics. They prefer, as in our case, to use the higher quality primary algorithm, but might have to use the faster alternate algorithm if the primary algorithm is predicted to take too long. The approach, however, is static[1] in that it schedules tasks off-line. Because of this, the approach does not handle situations in which the timing of tasks is unknown until run time, as in the problems we are attempting to solve.

Our approach uses exception handling to absorb dynamic scheduling failures. Other scheduling approaches have handled scheduling failures in different ways. For example, in Real-Time Concurrent C [4], the `guarantee` statement can have an `else` clause that indicates what to do in the cases where the run-time system cannot produce a valid schedule or cannot do so quickly enough. This approach allows the specification of local actions to fix scheduling problems, and the `else` clause could even be used to attempt to schedule a faster alternate algorithm. However, this approach requires that scheduling failures be handled locally. Our approach, based on a *scoped exception handling* mechanism, allows a programmer the option of handling the failures at higher scopes instead of requiring handling at the innermost scope.

Liu et al. [6] describe dynamic algorithms for scheduling imprecise calculations in which a calculation is logically broken into a mandatory subtask and an optional subtask. In that work, Liu et al. discuss fairly sophisticated algorithms to handle this class of problems. In our work, we have chosen instead to keep the scheduling algorithm relatively straightforward and build a structure around it that helps to solve the problem at hand in a flexible manner. This structure is the scoped exception handling approach. Also, while our overall goals are similar to Liu's in that we allow the specification of subtasks for a task, we do not limit the description to a single optional subtask.

The work we describe in this paper is also related to the area of intelligent scheduling, surveyed by Zweben and Fox [12]. In this work, there is a common distinction between *planning* and *scheduling*. The real-time systems community and the intelligent scheduling community share a definition of scheduling, namely that scheduling "assigns resources and times for each activity" so as to meet precedence and timing constraints [3]. Planning, on the other hand "selects and sequences activities" to achieve system goals [3]. Informally, planning is deciding which tasks to perform, while scheduling is deciding if they can be performed in time.

As shown by Durfee and Lesser [2], planning and scheduling must work together. They describe an approach that uses a planner to devise a plan of action, and then predicting using a scheduling algorithm how long the plan will take. They re-plan as necessary if the plan will overrun deadlines. Similar to our work and that of Liestman and Campbell [5] referenced earlier, Durfee and Lesser are willing to trade

[1]We take definitions of static and dynamic from Cheng, et al. [1].

quality of result for timeliness of achieving that result. In their paper, they focus on how to make the timing estimates on which a scheduler will predict the running time, something that we have not done in our work.

Instead of using an intelligent planner that designs a plan based on potentially conflicting criteria, such as Wagner and Lesser's Design-to-Criteria Scheduler [9], we have opted to hard-code the plan. This gives our approach a desired simplicity at the expense of lost intelligence. A planner that trades off various criteria to produce the optimal schedulable plan is likely to introduce combinatoric complexity – if removing single tasks from the schedule doesn't make it schedulable, or doesn't produce the highest quality, then the planner must then attempt removing pairs of tasks, then triples, etc. We are interested in providing some of the flexibility of such approaches, while keeping the number of plans to schedule tractable (perhaps linear in the number of tasks). However, it also seems fruitful to explore the use of such powerful planners in future work to see precisely what benefit they provide and at what complexity cost.

3 STATEMENT OF THE PROBLEM

We are interested in providing structures and approaches to help build real-time programs that provide graceful degradation under overloads in dynamic real-time systems.

We can formalize the scheduling problem for the window as follows:[2]

Given $T = \{T_1, T_2, ..., T_n\}$, the tasks that are to be scheduled, each task description $T_i = (t_i, d_i, r_i)$, where:

- t_i, the estimated processing time of the task.
- d_i, the deadline for the task, relative to the beginning of the time window.
- $r_i = \{R_{i_1}, R_{i_2}, ..., R_{i_m}\}$, the resource requirements of the task, where $R_{i_j} = \{RI_{i_{j_1}}, RI_{i_{j_2}}, ..., RI_{i_{j_k}}\}$ is a set of resource instances, such that T_i can be executed only if $\forall R_{i_j} : \exists r' \in R_{i_j}$, and r' is allocated to T_i. In this work, we assume that all resources so allocated are acquired for exclusive use.

In this section, we present a formal description of a class of problems for which schedules are desired. We start with a description of a traditional scheduling problem, which we call the Unconstrained Scheduling Problem, in which the input is a flat set of independent tasks that must be scheduled to complete before their deadlines. We generalize this to the set of problems, the precedence Constrained Scheduled Problem, in which some pairs of tasks are constrained by a precedence relation. We then extend this to address the Alternatives Scheduling Problem, in which some sets of tasks can be replaced in the schedule with alternative sets. Finally, we present the Hierarchical Scheduling Problem, a class of problems for which we have developed a flexible approach. Here, sets of tasks and there precedence relationships and alternatives are described using a hierarchical program representation and a schedule is constructed to exploit this hierarchical representation.

We define the Unconstrained Scheduling Problem, $\mathrm{USP}(T) \mapsto S$, where:

- $T = \{T_1, T_2, ..., T_n\}$ is a set of tasks.
- S is a valid, feasible schedule. $S = \{ST_1, ST_2, ..., ST_n\}$, a set of n scheduled tasks $ST_i = (T_i, S_i, RA_i)$, where:
  - T_i is the task that is scheduled.
  - S_i is the start time assigned to the task. $S_i + t_i \leq d_i$.
  - $RA_i = \{RI_{i_1}, RI_{i_2}, ..., RI_{i_m}\}$ is the resource assignment for T_i. $\forall RI_{i_j} \in RA_i : RI_{i_j} \in R_{i_j}$ and $\forall R_{i_j} \in r_i : \exists (r' \in RA_i) \in R_{i_j}$. Because the resources are acquired for exclusive use, no other

[2]We use notation similar to that found in [6].

task executing during the interval $(S_i, S_i + t_i)$ can be assigned any of the resource instances in RA_i.

The Constrained Scheduling Problem, $\mathrm{CSP}(T, P) \mapsto S$, adds a precedence relation, P as input:

- $P \subset T \times T$, a partial order.
- S is constrained such that $ST_j \in S \wedge (T_i, T_j) \in P \rightarrow ST_i = (T_i, S_i, RA_i) \in S \wedge S_i + t_i \leq S_j$.

We are interested in generalizations of this problem that allow more flexibility in what tasks are scheduled. For example, if the application uses pairs of redundant sensors to improve fault tolerance, it may be acceptable to schedule only one sensor of each pair under overloaded situations. While it is desirable to read all the sensors present, this may not be strictly required.

Let $A = \langle A_1, A_2, ..., A_p \rangle$ be a collection of alternatives ordered by preference. Each alternative $A_k \subseteq T$ is a set of tasks to be scheduled. The goal for this problem is then to schedule the most desirable alternative to meet the deadlines, which involves both choosing the best alternative and producing a valid, feasible schedule for it. We call this problem the Alternatives Scheduling Problem, $\mathrm{ASP}(T, P, A) \mapsto (A', S)$ where:

- T, P are defined as above.
- $A = \langle A_1, A_2, ..., A_p \rangle$, where $A_k \subseteq T$.
- $A' \in A$
- $\forall T_i : T_i \in A' \leftrightarrow T_i \in S$.

The decisions regarding which alternatives are schedulable may have to be taken dynamically if the expected durations of the tasks depend on run-time conditions (e.g., because of uncertainties in measurement, network load, sensor noise, mobility, etc...).

Many traditional scheduling approaches (e.g., [8]), are designed to determine only whether tasks are schedulable or not. In this paper, we present a dynamic approach to choosing and scheduling alternatives, and thus provide a more flexible solution to this class of problem. We give particular attention to the subclass of problems in which the alternatives $A_2, A_3, ..., A_p \subset A_1$. In the sensor reading example, an alternative to reading all the sensors is to read some subset of the sensors.

Under the hypothesis that having more tasks successfully scheduled is better than having fewer, we propose to measure the effectiveness of solutions to the Alternatives Scheduling Problem by counting the total number of tasks scheduled in a given time interval. The goal is thus to maximize the amount of work scheduled to meet the given deadlines.

3.1 An Example

Consider, for example, a control application that reads two pairs of redundant sensors. The application uses four tasks, each reading one sensor: $T = \{T_{1a}, T_{1b}, T_{2a}, T_{2b}\}$, where each of the tasks has an estimated processing time, $t_i = 150$ time units, and a deadline, $d_i = 300$. Each task has a resource specification $r_i = \{R1\}$, where $R1$ is a set of resource instances. The precedence relation, P, contains no pairs and therefore the tasks can be scheduled concurrently.

If $R1 = \{RI_1, RI_2\}$, then two of the leaf tasks can be scheduled concurrently, and all tasks are schedulable. A possible valid, feasible schedule is $S = \{ST_{1a}, ST_{1b}, ST_{2a}, ST_{2b}\}$ in which $S_{1a} = S_{1b} = 0$ and $S_{2a} = S_{2b} = 150$.

If these tasks are executed repeatedly, and if the time estimates, t_i, vary from iteration to iteration and may exceed 150 in some, it is possible that in some iterations not all four readings are schedulable. In a

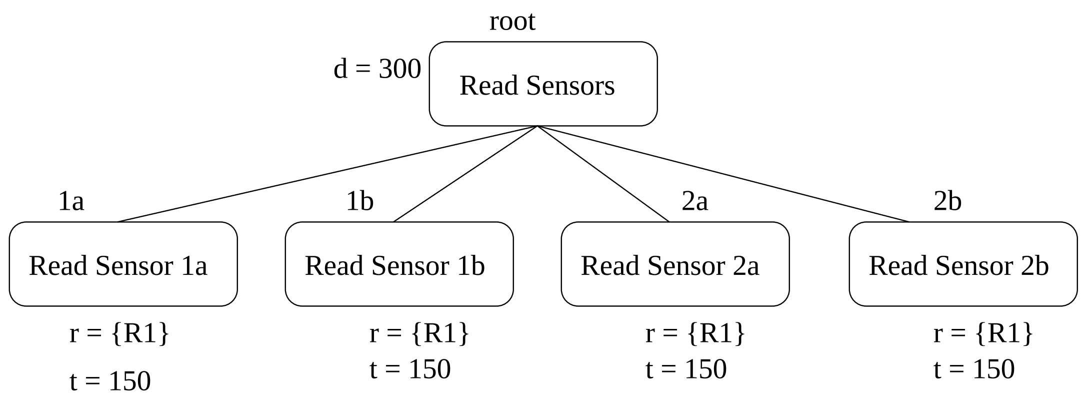


Figure 1: An example hierarchical program

traditional dynamic scheduling algorithm [8], this failure to produce a schedule for all four readings will result in none of the tasks being scheduled. Therefore the total number of tasks scheduled is not increased on that iteration.

In this class of problems it is desirable to schedule some instead of no tasks on each iteration, if possible. In the example, we can afford to schedule T_{1a}, T_{1b}, and T_{2a} in those cases where the time estimates are too high to schedule all four tasks. Therefore, a programmer might define two acceptable task sets, $A_1 = \{T_{1a}, T_{1b}, T_{2a}, T_{2b}\}$ and $A_2 = \{T_{1a}, T_{1b}, T_{2a}\}$ for the Alternatives Scheduling Problem.

3.2 Hierarchical Program Structures

In the descriptions of the problem presented above, T is a set of tasks, T_i, each of which must be defined individually. However, we have found that the introduction of a hierarchical notation allows for simpler specification of the time, resource, and precedence constraints of the tasks, and so will use hierarchical program structures in the rest of this paper.

The introduction of hierarchy changes the Alternatives Scheduling Problem slightly in that T is redefined and an additional function, p is added:

- p is the parent function, defining a tree. $p(i) = j \leftrightarrow T_j$ is the parent of T_i in the tree. $p(x)$ is not defined if x is the root of the tree.
- We define the root of the tree: $T_{root} : \nexists i : p(root) = i$.
- $T = \{T_1, T_2, ..., T_n\}$, where $T_i = (t_i, d_i, r_i)$, where:
  - t_i returns the duration of T_i. If T_i is not a leaf, i.e. $\exists x | p(x) = i$, then $t_i = 0$. Otherwise, t_i is equal to the duration specified for T_i. We define t_i this way because, in this hierarchical structure, the leaf tasks are those tasks that need to do work, while the non-leaf tasks are used for structuring to ease specification.
  - d_i returns the deadline of T_i. If there is a deadline specified for T_i directly, d_i returns it. Otherwise, d_i returns $d_{p(i)}$.
  - r_i returns the resource specification for T_i. If there is a resource specification specified for T_i directly, r_i returns it. Otherwise, r_i returns $r_{p(i)}$.

Instead of requiring the specification of a deadline, d_i for each task, the programmer can specify a deadline for a parent task. This is shown in the hierarchical program description of the sensor reading example problem in Figure 1. Each of the leaf steps in the example has a processing time, t_i, specified but the deadline, d_i, is derived from the root step.

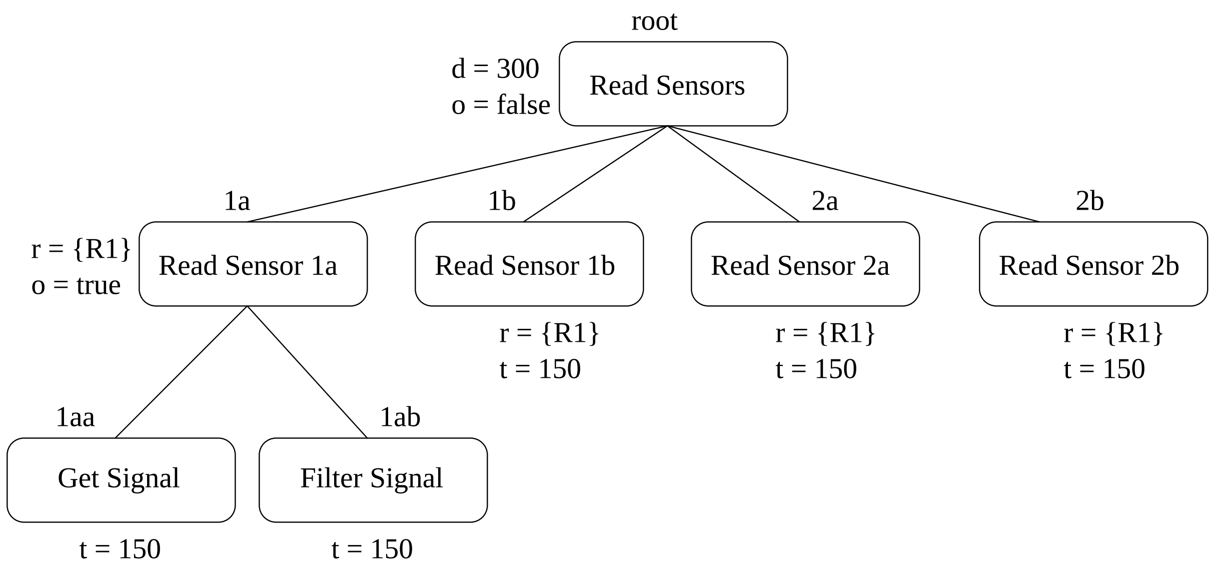


Figure 2: A more complex hierarchical program

Similarly, task T_{1a} in Figure 2 is defined to consist of the two tasks, T_{1aa} and T_{1ab}. The programmer does not have to indicate a deadline on T_{1a}, in which case the deadline for the two subtasks will be derived from the deadline for the root. Using a hierarchical description also allows the specification of resource requirements to be simpler. In Figure 2, T_{1aa} and T_{1ab} use the same resource as is specified in T_{1a}, namely a resource from the set $R1$. In this way, the hierarchical representation provides a flexibility to the programmer to specify only the real timing requirements, instead of being forced to specify deadlines at all levels just to make the scheduling algorithm simpler.

A further redefinition of non-leaf tasks, $T_i = (t_i, d_i, r_i, o_i)$, simplifies the specification of the precedence relation, P:

- $o_i \in \{true, false\}$ provides ordering information for subtasks. If o_i is true, the subtasks of T_i are said to be *ordered* from left to right. If o_i is false, the tasks are *unordered*.
- We now define P':
  - $\forall T_i : (T_i, T_{p(i)}) \in P'$.
  - Let $s_i = \langle T_{i_1}, T_{i_2}, ..., T_{i_n} \rangle$ be the subtasks of T_i, $T_{i_j} | p(i_j) = i$, ordered in the same order as the left-to-right order of the task descriptions in the graphical description.
  - $o_i = true \rightarrow \forall_{i \le j \le n-1} T_{i_j} : (T_{i_j}, T_{i_{j+1}}) \in P'$).
- P can now be defined based on the transitive closure of P', P'^*. Since we are only interested in the precedence relationships between leaf tasks, $P = P'^* - \{(T_i, T_j) | \exists T_x : p(x) = i \vee p(x) = j\}$.

In the example show in Figure 2, because $o_{a1} = true$, $P' = \{(T_{1a}, T_{root}), (T_{1b}, T_{root}), (T_{2a}, T_{root}), (T_{2b}, T_{root}),$ $(T_{1aa}, T_{1a}), (T_{1ab}, T_{1a}), (T_{1aa}, T_{1ab})\}$. Therefore, $P = \{(T_{1aa}, T_{1ab})\}$.

However, if $o_{root} = true$ also, then $P' = \{(T_{1a}, T_{root}), (T_{1b}, T_{root}), (T_{2a}, T_{root}), (T_{2b}, T_{root}),$ $(T_{1aa}, T_{1a}), (T_{1ab}, T_{1a}), (T_{1a}, T_{1b}), (T_{1b}, T_{2a}), (T_{2a}, T_{2b}), (T_{1aa}, T_{1ab})\}$ and therefore $P =$ the transitive closure of $\{(T_{1aa}, T_{1ab}), (T_{1ab}, T_{1b}), (T_{1b}, T_{2a}), (T_{2a}, T_{2b})\}$.

Clearly, in order to enjoy these specification simplifications, the scheduling algorithm must deal with the hierarchical representation of tasks. We extend a heuristic search scheduling algorithm, originally designed for non-hierarchically defined sets of tasks, to schedule hierarchical structures and use the hierarchy to help provide the needed flexibility. We further use exception handling to provide planning capabilities.

4 OUR APPROACH

In this section we first describe a simple dynamic scheduling algorithm for solving the Constrained Scheduling Problem, that we then extend to deal with hierarchical task structures. Then we show how to deal with scheduling failures by capitalizing on the specifications of scoped exception handling, thus solving the Alternatives Scheduling Problem. The approach is evaluated in the next section.

4.1 A dynamic real-time scheduling algorithm

Since we extend an existing dynamic scheduling algorithm [8] to suit our needs, we first give a short summary of this basic algorithm which solves the Unconstrained Scheduling Problem. At the end of this subsection, we show how to extend it to solve the Constrained Scheduling Problem.

The basic approach uses a working list, L_w and a scheduled list, L_s, which represents the current partial schedule. Initially, $L_w = T$. There is no precedence relation, P, and therefore all tasks on L_w can be scheduled to execute in parallel. From this starting point, the algorithm iterates until $L_w = \varnothing$, on each iteration choosing the heuristically determined "best" task to move from L_w to L_s. An additional data structure is used to keep track of the availability of resources, such that we can use $EAT(r)$ to determine the earliest available time that a resource is available. Initially $\forall r : EAT(r) = 0$. As a task T_i is added to L_s,

- The resource assignment, $RA_i \leftarrow \{RI_{i_1}, RI_{i_2}, ..., RI_{i_m}\}$, where $\forall RI_{i_k} \in RA_i : \nexists r' \in R_{i_k} | EAT(r') < EAT(RI_{i_k})$.
- The starting time $S_i \leftarrow MAX_{1 \le k \le m} EAT(RI_{i_k})$.
- The EAT structure is updated: $RI_{i_k} \in RA_i \rightarrow (EAT(RI_{i_k}) \leftarrow S_i + t_i)$.

This iteration continues as long as the schedule represented by L_s is feasible (all tasks complete before their deadlines). If the schedule is infeasible, the algorithm backtracks. If $L_w = \varnothing$ and the schedule created in feasible, then the algorithm succeeds because the partial schedule is now a complete feasible schedule.

This basic approach can be modified to solve the Constrained Scheduling Problem by ensuring that the heuristic that chooses a task from L_w only chooses tasks from the set, $T' = \{T_j | \forall T_i : (T_i, T_j) \in P \rightarrow T_i \in L_s\}$, of all tasks whose preceding tasks are already in L_s.

4.2 Modifications to support hierarchy

In the next extension we develop the hierarchical algorithm. To this end, we adapt the above-described algorithm to take the hierarchy of our program structures into account. We again use a working list L_w and a scheduled list L_s. Initially, $L_w = \{T_{root}\}$. The hierarchical algorithm then iterates until $L_w = \varnothing$, just as in the basic algorithm, adding a task from L_w to L_s based on a heuristic application as long as the schedule is still feasible. However, in order to effectively enforce the hierarchical precedence relationship, P, in the hierarchical algorithm, L_w can grow. In particular, when a task T_i is added to L_s, a set of *following* tasks can be added to L_w: $L_w \leftarrow L_w \cup F_i$, where F_i is defined as follows:

- $\nexists T_j \in L_w : p(j) = i$:
  - $o_{p(i)} = true \wedge (T_i = T_{i_j} \in s_{p(i)}, j < |s_{p(i)}|) \rightarrow (F_i \leftarrow \{T_{i_{j+1}}\})$, the next substep after T_i.
  - $o_{p(i)} = true \wedge (T_i = T_{i_j} \in s_{p(i)}, j = |s_{p(i)}|) \rightarrow (F_i \leftarrow F_{p(i)})$.

    - $o_{p(i)} = false \wedge (\forall T_{i_j} \in s_{p(i)} : T_{i_j} \in L_s) \rightarrow (F_i \leftarrow F_{p(i)})$.

- $\exists T_j \in L_w : p(j) = i$:
    - $o_i = true \rightarrow (F_i \leftarrow \{T_{i_1}\})$, the first element of the subtasks set.
    - $o_i = false \rightarrow (F_i \leftarrow s_i)$, all of the subtasks.

By growing the work list in this way, at any time during execution, L_w contains all those tasks that can be added to the partial schedule represented by L_s. By construction, L_w is such that $\forall T_i \in L_w : \nexists T_j \neq T_i | (T_j, T_i) \in P \wedge T_j \notin L_s$. Therefore, at every iteration of the algorithm any of the tasks on the work list can be added to the schedule.

We should note that this is not the only way to adapt the algorithm to work with hierarchical structures. We could have, for example, allowed L_w to contain all tasks T as in the basic approach, and constrained the selection of tasks from L_w as in the basic approach. However, this would have made the implementation more difficult to write and made it harder to reason about what tasks, at any point, can be added to L_s. Our approach keeps L_w simple and makes it easy to reason about what tasks are allowed on the schedule. The main loop of the algorithm does not have to keep track of the precedence relationship – it is only after a task is added to the schedule that we use the precedence semantics to update L_w. A smaller number of tasks in L_w also implies lower scheduling overheads.

We also believe that our approach keeps L_w sufficiently short so as to reduce the number of heuristic applications during the execution of the scheduler to a tractable number. We propose to attempt to verify this hypothesis in future work.

4.3 Planning via exception handling

The scheduling algorithm above solves the Constrained Scheduling Problem (CSP) for hierarchical structures. However, the scheduling algorithm alone does not solve the Alternatives Scheduling Problem, ASP. To solve ASP, we introduce scoped exception handling to the hierarchical program structures, as exemplified by Little-JIL [10, 11] in which hierarchical program structures can be specified with exception handling. Little-JIL provides a graphical representation for the specification of the hierarchical structures, very similar to those shown in Figures 1 and 2. The sequencing of subtasks is defined by either an arrow or two parallel lines, representing the state when $o_i = true$ or $o_i = false$, respectively.

Figure 3 shows the sensor reading example problem, as implemented in this language. This program represents a controlled approach to solving two instances of the CSP: We first attempt the scheduling algorithm on the task called Take All Readings, because there is a deadline specified on the task, which makes it the root of a CSP problem. If the hierarchical algorithm described above fails to produce a schedule for this CSP, a SchedulingFailure exception is thrown, which is handled by the handler attached to the Read Sensors task. The handler, called Perform Three Readings is the root of another CSP problem in which we attempt to schedule three of the tasks. This approach solves the ASP problem that consists of the two alternatives $A = \{A_1 = \{T_{1a}, T_{1b}, T_{2a}, T_{2b}\}, A_2 = \{T_{1a}, T_{1b}, T_{2a}\}\}$, by first attempting to schedule A_1 and then scheduling A_2 if that is unsuccessful.

This approach represents a hard-coded plan: it provides an answer to the question of which tasks to attempt in the case when not all can be scheduled. As we will see in the next section, this seemingly rudimentary, hard-coded plan is surprisingly well-suited in improving the schedulability of tasks.

While we have, in this example, defined the alternatives A as a sequence of subsets of the first alternative, A_1, our approach can be used to provide different planning solutions. For example, we could handle the scheduling failure by attempting a completely different set of tasks that gives desirable timeliness. The flexibility of the approach allows us the ability to execute many different types of remedies.

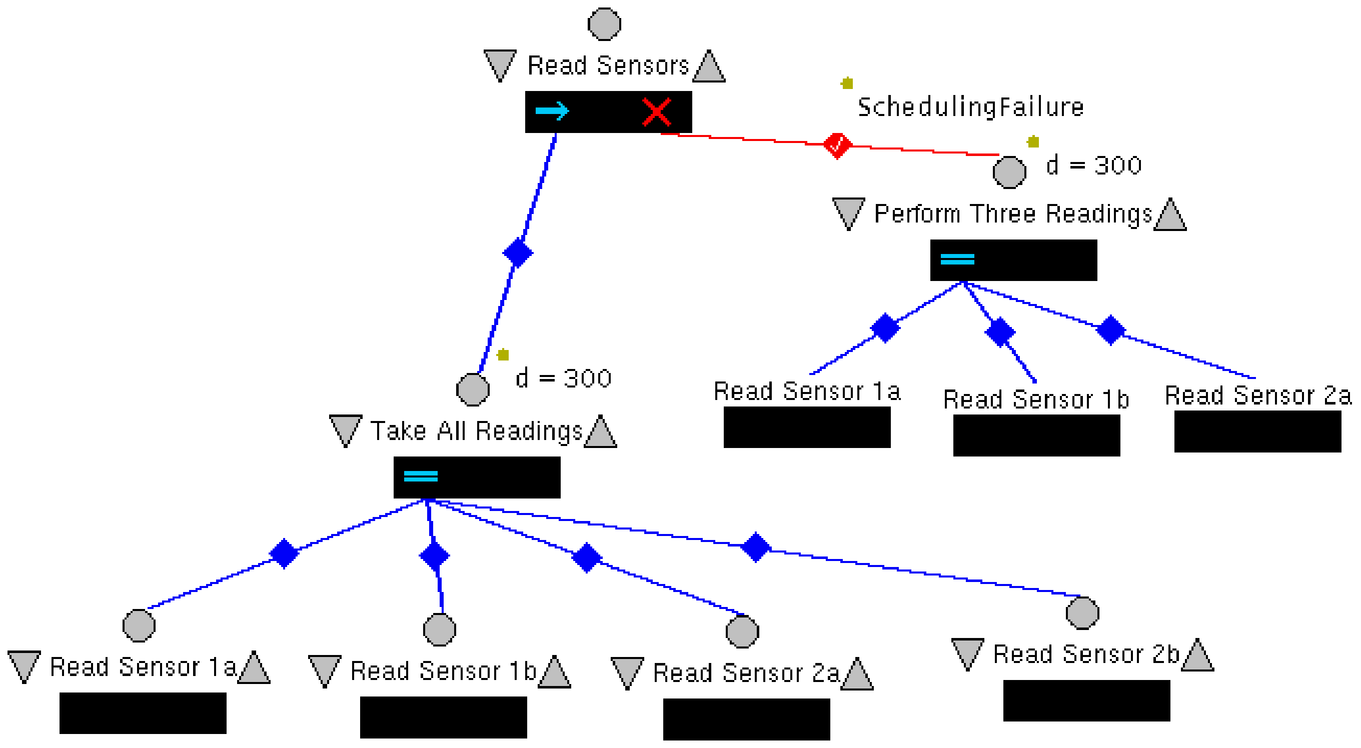


Figure 3: An example hierarchical program with exception handling for scheduling failure

Also, because the exception handlers are themselves tasks that fit into a larger hierarchy, we can specify at even higher levels what to do in the event of a failure to schedule the exception tasks. For example, in Figure 4, a first scheduling failure causes the system to attempt to schedule three tasks. However, if the scheduler is unable to do so, a second scheduling failure occurs, which is handled by attempting to schedule only two tasks. This chaining of possible solutions to scheduling failure allows us not only to specify different remedies, but also to specify in what order they should be attempted. So, unlike the approach taken by Liu, et al. [6], the task is not simply divided into two subtasks.

Also note that the approach using exception handling allows the specification of handling at higher levels. So, if multiple CSP problems are being attempted in a hierarchical program, failures from all of them can all be handled by the same response, as shown in Figure 5. In this example, a scheduling failure can occur at any level deep in any of the three main subtasks of the root, but they are all handled by the same response. This provides more flexibility than the local specification required by other approaches (e.g., as in Real-Time Concurrent C [4]).

While the flexibility, simplicity, and quality of output the approach provides is desirable, we seek in future work to develop approaches that require less hard-coding of the manner in which scheduling failures are handled.

5 EXPERIMENTATION

In this section, we describe experiments performed to ascertain the usefulness of using hierarchical structures and scoped exception handling to derive a plan for the sensor reading program described in Figure 3.

We compare our approach to an approach, which we call the all-or-nothing approach, that does not use exception handling and therefore only solves the Constrained Scheduling Problem, CSP, for the original set of tasks, ignoring the alternatives possible.

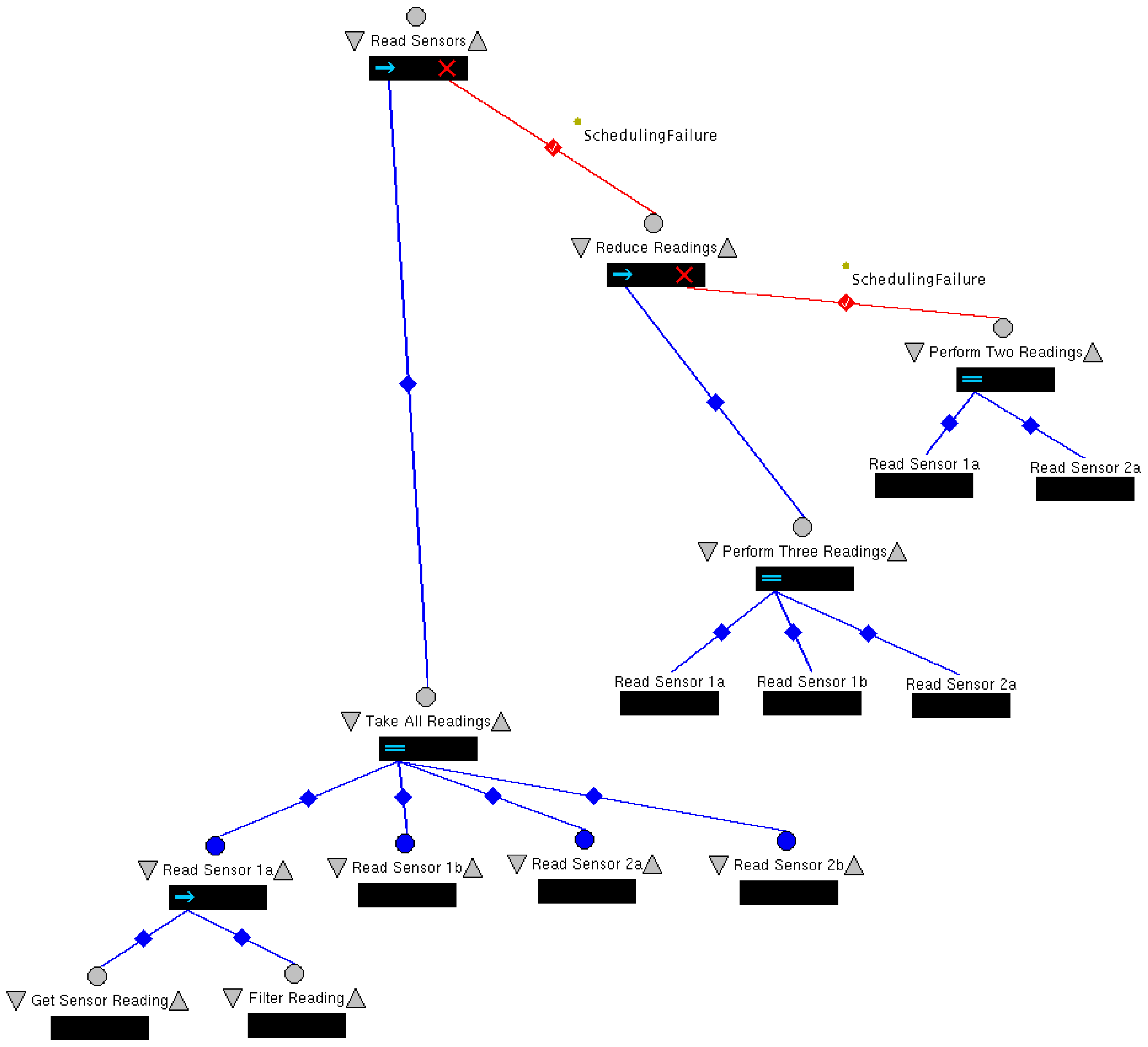


Figure 4: More complex exception handling

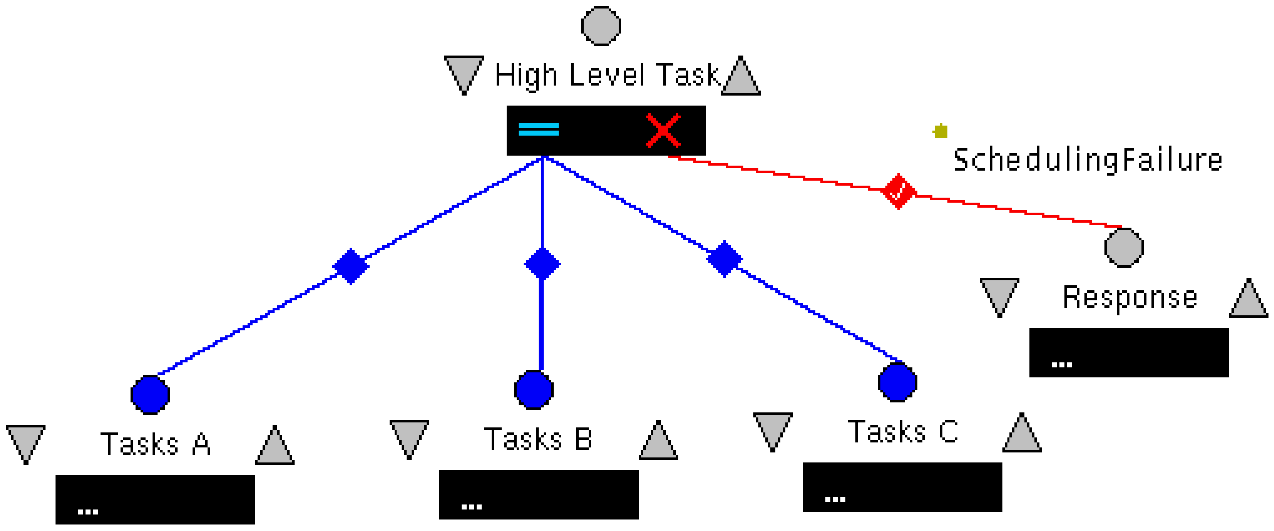


Figure 5: Handling Failures at Higher Levels

Our first set of experiments was aimed at determining what problem parameters affect the benefit of our proposed approach the most. It is clear that the approach will never fail to schedule A_1 when the traditional approach would do so. However, the benefit of our approach should be less for some task structures than for others.

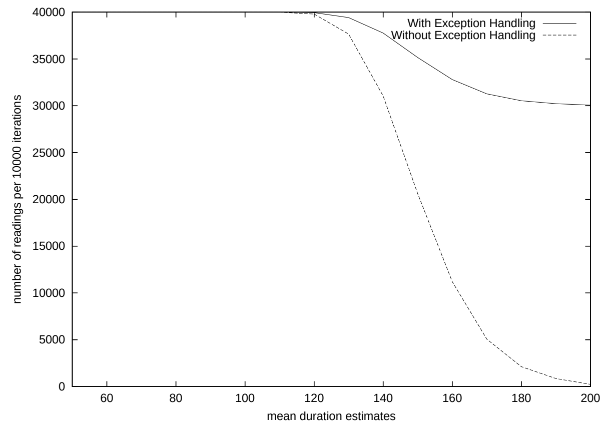
In order to determine how full the scheduling window on average needs to be in order for our approach to be of benefit, we modeled the problem as follows:

- We consider two alternatives $A = \{A_1 = \{T_{1a}, T_{1b}, T_{2a}, T_{2b}\}, A_2 = \{T_{1a}, T_{1b}, T_{2a}\}\}$. Clearly, we should first attempt to schedule A_1 and then schedule A_2 if that is unsuccessful.
- $\forall i : r_i = \{R1\}$, where $R1 = \{R_1, R_2\}$ is a resource class with only two instances. This means that only two of the leaf tasks of A can execute concurrently.
- $d_{root} = 300$ time units for each of the two roots.
- $\forall i : t_i \sim N(\mu, \sigma)$, where we varied μ and σ for different experimental runs such that $50 \leq \mu \leq 200$, $\sigma = \frac{\mu}{10}$.

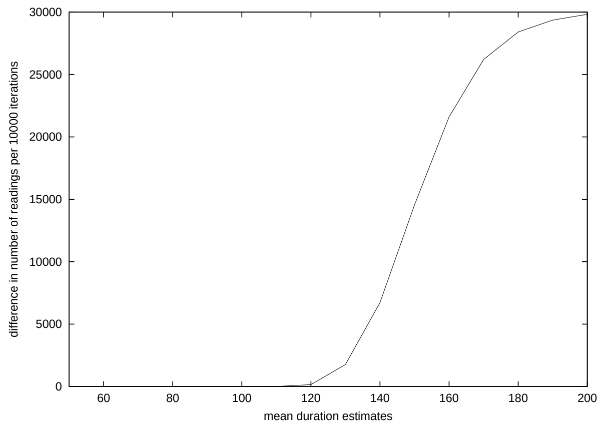
Given tasks with the above characteristics, we experimented over 10000 iterations of A and measured the performance in terms of the metric $W = \sum_{0 \leq i \leq N} |A(i)|$, where $|A_1| = 4$ and $|A_2| = 3$ and $N = 10000$ is the number of iterations.

Figure 6(a) shows the results of this first set of experiments. As can be seen, when $\mu \leq 100$, both the all-or-nothing approach and our approach schedule the same number of readings. However, as the means get higher, more scheduling failures are noticed for A_1, and the more we have to resort to A_2, scheduling only three tasks. For $100 \leq \mu \leq 130$, the difference in the number of readings scheduled is less than two thousand, out of a possible total of forty thousand. For $\mu \geq 150$, the difference starts to be very noticeable, and when $\mu = 200$, our approach is able to schedule over thirty thousand tasks, whereas the all-or-nothing approach schedules less than three hundred. Roughly, in this problem when $\mu \leq 100$, our approach does not provide a benefit, while when $\mu > 130$, our approach's exception handling-based scheduling provides significant benefits.

This experiment shows that with $\mu = 100$ and $\sigma = 10$, there is almost always sufficient time to execute the complete set of leaves, while with $\mu = 150$ and $\sigma = 15$, the time window of 300 units was over-full



(a) Both Approaches



(b) Difference Between Approaches

Figure 6: Results – Varying Mean Duration Estimates

approximately half the time.

In order to determine the sensitivity of these results to deviations in processing times, we undertook two additional sets of experiments.

First, for the case with $\mu = 150$, in which the window is full about half the time, we expect to see that with even small deviations the system will fail to schedule A_1. To test this hypothesis, we model the problem as follows:

- $\forall i : t_i \sim N(150, \sigma)$, $1 \leq \sigma \leq 100$.

Second, for the case with $\mu = 100$, we would expect, since the time window is under-full by 100 time units, on average, that the system would be able to schedule A_1 very often but that for very large deviations we would expect to see failures to schedule A_1, resulting in attempts to schedule A_2. To test this hypothesis, we model the problem as follows:

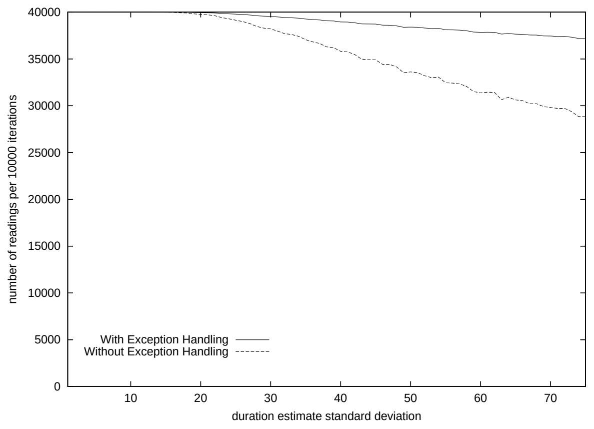
- $\forall i : t_i \sim N(100, \sigma)$, $1 \leq \sigma \leq 75$.

As can be seen in Figure 7(b), when the window is full on average approximately one half of the iterations cause a scheduling failure and therefore the approach is not very sensitive to large variances in the execution time durations. Even with a large variance, the approach was consistently one and a half times better that the all-or-nothing approach. However, as shown in Figure 7(a), where $\mu = 100$ and the average case does not come very close to filling the time window, the approach is more sensitive to variance in the execution times. As the variance becomes larger, the benefits of our approach increase.

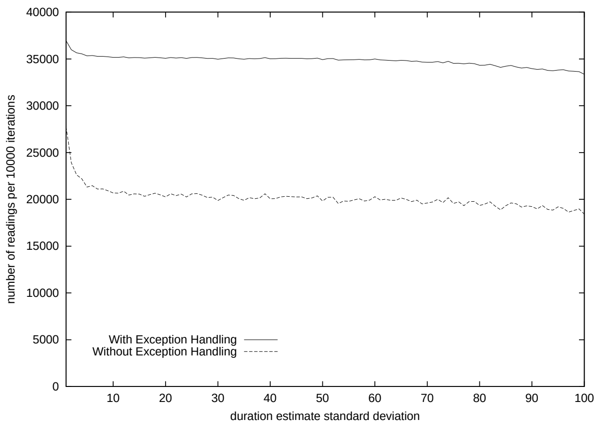
5.1 Analysis

Our experiments quantitatively show the benefits of using our approach under different circumstances. When designing a real-time system, a designer has many scheduling approaches to choose from. Depending on the characteristics of the tasks to be scheduled and the time window in which to schedule them, some approaches will be better suited than others. Our experimentation has provided some guidance to help such a designer.

As can be seen in Figure 6(b), the difference between the all-or-nothing approach and our approach is negligible in non time-constrained situations ($\mu \leq 120$), but in time-constrained situations ($\mu \geq 150$), the difference is quite noticeable.



(a) Mean = 100



(b) Mean = 150

Figure 7: Results – Varying Standard Deviations

Based on our experimentation, we would therefore advise the designer to use the approach presented in this paper when:

- The problem is more naturally described using hierarchical representations;
- The task processing times is expected to vary significantly from iteration to iteration; or
- The time window duration is such that it will be, on average, a tight fit for the tasks.

This analysis, however, does not address the costs of this approach. In our approach using exception handling for planning, we have to schedule A_1 just as the traditional approach does. In addition, if that attempt fails, we have to then attempt to schedule A_2. We can approximate the cost of the approach by assuming that the scheduling algorithm is linear in the number of tasks (which is supported by results presented in [8]). Therefore, if the cost to schedule $A_1 = 4x$, then the cost to schedule $A_2 = 3x$. In the case when $\mu = 150$, in which the scheduler fails in approximately half the cases to schedule A_1, the expected scheduling cost will be: $\frac{4x}{2} + \frac{7x}{2} = 5.5x$. This cost seems quite reasonable for the benefits we can get.

In order to more carefully pinpoint the problem characteristics for which this approach is useful, further experimentation is needed. For example, our current experiments did not vary the resource availability. A designer of a dynamic real-time system is likely to face varying resource availability, for example, due to failures or noise levels.

Another potential difference between our models and actual real-time systems is the uniformity of the tasks. In our experiments, we have assumed four tasks, each with similar mean processing times. However, in real-world systems, task characteristics are likely to be much more varied.

6 LESSONS LEARNED AND FUTURE WORK

6.1 Benefits of Hierarchy

We have made good use of hierarchy. In addition to the standard software engineering arguments in favor of hierarchy as a way of providing abstraction and complexity control, we have found that hierarchy has some scheduling benefits. We have used the hierarchy and a scoped exception handling mechanism to provide planning that helps us get more work scheduled in a broadly applicable class of problems.

In addition to this quality benefit, there are other benefits of hierarchy:

- Clarity of precedence relationships – Specifying the precedence relationship between tasks in this hierarchical way makes it clear which groups of tasks must precede which other groups.

- Working list size – The size of L_w in our scheduling algorithm is kept relatively small by virtue of only containing, during each invocation of the scheduling algorithm those tasks that can currently be scheduled. Keeping track of which tasks that can be scheduled and adding to the list when new tasks are scheduled is aided by the hierarchy.
- Multiple alternative types – In this work, we have concentrated on the class of problems in which the alternatives $A_2, A_3, ..., A_n \subset A_1$. The hierarchically-scoped exception handling mechanism allows the specification of differing alternatives. We plan to experiment with the more general problem as part of our future work.

6.2 Applicability of Approach

We have found through experimentation that for problem domains in which processing times deviate a lot, or in which scheduling windows are very tight, our approach has an advantage over an all-or-nothing approach. We believe that there are important real-time problem domains that fit this description. We plan in future work to investigate the use of this approach with real-world problems.

6.3 Planning

Dealing with real-time issues requires not just scheduling, but planning. The example hierarchical program shown in Figures 1 and 3 exhibit such a planning problem. Instead of just scheduling alternative A_1, the approach must first determine which alternative to schedule, and then schedule it. In the example, we have hard-coded the planning by using the exception handling mechanism of Little-JIL.

Other approaches to hard-coding the plan should be investigated. For example, allowing the specification of optional tasks is one way to give the programmer flexibility in expressing a plan. Instead of explicitly defining A_1 and A_2, the programmer could specify that $A_1 = \{T_{1a}, T_{1b}, T_{2a}, T_{2b}\}$, and specify that task T_{2b} is optional. This is, in effect, defining two alternatives, A_1 with T_{2b} and A_1' without T_{2b}. The plan would still be hard-coded (because the alternatives are completely, though implicitly, specified and the order in which to attempt them is defined), but this would provide a simpler way to specify the plan.

If T_{2b} and T_{1b} were both defined to be optional, there would be four alternatives. Further, the order in which to attempt the alternatives would not be fully specified. If the number of optional tasks is not limited, the number of alternatives possible increases exponentially. This provides one way in which we might allow a programmer to avoid hard-coding the plan. However, in order to keep the approach tractable, we might try only those alternatives in which a single optional task is removed. Choosing which alternative to attempt first could be determined intelligently. Perhaps, for example, the programmer would specify priorities on tasks that can be used by the exception handler to determine how to construct the next-attempted alternative.

We might also try a multi-version approach similar to that described by Marlin et al. [7]. In that approach, a version variable can be used by the higher-level task to determine which subtasks to include. The planner can then attempt to schedule the highest-numbered version first, and respond to failure by attempting the second highest. In this way, the programmer could specify a chain of responses.

The system development complexity would be much easier for the programmer to handle if plans did not have to be hard-coded into the programs. We could rely, for example, on an artificially intelligent planning component. Such a component could use a scheduler to determine if the tasks can be done in time, as was done in the work described by Durfee and Lesser [2]. If the scheduler fails to produce a schedule, the planner can re-plan, deciding which tasks to remove based on how long the tasks were supposed to take or on other criteria such as quality and cost [9]. The scheduler can then attempt to schedule the newly planned tasks. This iteration would continue until a feasible schedule is produced.

While we seek solutions that do not introduce undue complexity, we do seek more intelligent and flexible solutions. Some of the approaches mentioned here should be investigated to provide this needed flexibility in simple ways. More complex solutions from intelligent scheduling should also be investigated [12, 3, 9].

ACKNOWLEDGEMENTS

This research was partially supported by the Air Force Research Laboratory/IFTD and the Defense Advanced Research Projects Agency under Contract F30602-97-2-0032, by the U.S. Department of Defense/Army and the Defense Advance Research Projects Agency under Contract DAAH01-00-C-R231 and by the National Science Foundation under contract IRI-9619588. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of the Defense Advanced Research Projects Agency, the Air Force Research Laboratory/IFTD, the U.S. Dept. of Defense, the U. S. Army, the National Science Foundation, or the U.S. Government.

REFERENCES

[1] S.-C. Cheng, J. A. Stankovic, and K. Ramamritham. Scheduling algorithms for hard real-time sytstems – a brief survey. In J. A. Stankovic and K. Ramamritham, editors, *Hard Real-Time Systems Tutorial*, pages 150–173. IEEE Comp. Society Press, Washington, DC, 1988.

[2] E. H. Durfee and V. R. Lesser. Planning to meet deadlines in a blackboard-based problem solver. In J. A. Stankovic and K. Ramamritham, editors, *Hard Real-Time Systems Tutorial*, pages 595–608. IEEE Comp. Society Press, Washington, DC, 1988.

[3] M. S. Fox. ISIS: A retrospective. In M. Zweben and M. S. Fox, editors, *Intelligent Scheduling*, pages 3–28. Morgan Kaufmann, San Francisco, 1994.

[4] N. Gehani and K. Ramamritham. Real-Time Concurrent C: A language for programming dynamic real-time systems. *J. of Real-Time Systems*, 3(4):377–405, Dec. 1991.

[5] A. L. Liestman and R. H. Campbell. A fault-tolerant scheduling problem. *IEEE Trans. on Soft. Eng.*, 12(11):1089–95, Nov. 1986.

[6] J. W. Liu, K. Lin, W. Shih, A. C. Yu, J. Chung, and W. Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, May 1991.

[7] C. Marlin, W. Zhao, G. Doherty, and A. Bohonis. GARTL: A real-time programming language based on multi-version computation. In *Proc. of the Int. Conf. on Comp. Languages*, pages 107–115, Mar. 1990. New Orleans, LA.

[8] K. Ramamritham, J. A. Stankovic, and P. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, Apr. 1990.

[9] T. Wagner and V. Lesser. Design-to-Criteria scheduling: Real-time agent control. Technical Report 99-58, U. of Massachusetts, Dept. of Comp. Sci., 1999.

[10] A. Wise. Little-JIL 1.0 Language Report. Technical Report 98-24, U. of Massachusetts, Dept. of Comp. Sci., Apr. 1998.

[11] A. Wise, A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, and S. M. Sutton, Jr. Using Little-JIL to coordinate agents in software engineering. In *Proc. of the Automated Software Engineering Conf.*, Sept. 2000. Grenoble, France.

[12] M. Zweben and M. S. Fox, editors. *Intelligent Scheduling*. Morgan Kaufmann, San Francisco, 1994.