# Robust scheduling of metaprograms

Ladislau Bölöni and Dan C. Marinescu
(Email: boloni, dcm@cs.purdue.edu)
Computer Sciences Department
Purdue University
West Lafayette, IN, 47907, USA

September 29, 1998

**Abstract**

Scheduling meta applications on a computational grid uses estimation of the execution times of component programs to compute optimal schedules. In a realistic case various factors (hazards) lead to estimation errors, which affect both the performance of a schedule and resource utilization. We introduce the concept of robustness and present an analysis technique to determine the robustness of a schedule. We develop methods for reducing the chance that a metaprogram exceeds its execution time due to components outside its critical path. The results of this analysis are used to compute schedules less sensitive to hazards. This translates into more accurate reservation requirements for critical systems, and reduced expected execution time for non-critical metaprograms executed repeatedly. Simulation results prove the efficiency and applicability of our algorithms.

## 1 Introduction

Informally, a metaprogram is a collection of programs which cooperate towards a common goal. A computational grid is an abstraction for a collection of autonomous and heterogeneous computers interconnected by a high speed network. A metaprogram is executed on a computational grid. Throughout this paper we assume that there is no one scheduler which controls all resources of the system and that each local scheduler accepts reservations. We also assume that a meta scheduling agent [1] has information about the execution time of each component of the metaprogram and is capable to compute schedules. A schedule associates a node of the computing grid and a start up time to each component of the metaprogram. For an overview of high performance schedulers for a grid of autonomous computers and a comprehensive bibliography on the subject we refer the reader to [2].

Several resource allocation models for a computing grid are possible: (a) Static allocation by spatial partitioning of the grid. A partition consisting of components of the grid is assigned to a metaprogram for the entire duration of its execution. (b) Dynamic allocation by temporal partitioning of the grid. A partition grows or shrinks in time depending upon the resource requirements of the metaprogram. (c) Combinations of the two strategies above, e.g. all resources needed are allocated at the time a metaprogram becomes active, then each resource is released once it is no longer needed by any component of the metaprogram.

The resource allocation model has profound implications upon resource utilization and upon the cost of running an application. Each model is suitable for a specific class of applications, e.g. hard real time applications are most likely to use (a) while soft real time or non-critical (common) applications may tolerate (b). Here we do not address the actual implementation of scheduling on a grid, the interested reader may consult [3] for a taxonomy of scheduling in distributed systems and [4] for a hierarchical decision model for scheduling.

There are two major challenges in metaprogram scheduling:

(a) The scheduling problem is NP-complete, therefore finding an optimal solution may be impossible or impractical except for trivial metaprograms and small grids. To overcome the explosion of the search space for realistic problems, one can apply approximation algorithms (heuristic-guided search), or genetic algorithms [5].

(b) The nondeterministic nature of the program execution time renders even an optimal solution approximate, or infeasible depending upon the resource allocation model. Several solutions to accommodate the nondeterministic execution time of the components of a metaprogram are possible:

(b1) Grossly overestimate the execution time of each program and minimize the risk of exceeding the alloted use of each host at the expense of the utilization of the grid,

(b2) Use dynamic algorithms which compute schedules at execution time. Once the data flow allows a program to be scheduled, gather information about the state of the grid and compute a new schedule for the remaining components of the metaprogram,

(b3) Use static scheduling algorithms to compute schedules for various scenarios, and at run time adopt the schedule which best fits the current conditions, [5]. If the grid is shared by multiple metaprograms this may not be feasible,

(b4) Use static algorithms for finding schedules less vulnerable to hazards i.e. more robust.

In this article we explore the last alternative and observe that it can be used in conjecture with any other approach for accommodating the nondeterministic program execution times.

We now introduce a formalism for metaprogram scheduling and the notations used throughout this paper:

$N_i$, A, B, C – the components of a metaprogram
$H_j$ – the hosts of the grid
$S_i$ – the schedules
$RunTime(N_i, H_j)$ – the execution time of the component $N_i$ on $H_j$
$P_i$ – a path of the metaprogram
$\mathcal{P}$ – the set of paths of the metaprogram
$\mathcal{P}_c$ – the subset of the potentially critical paths
$\mathcal{P}_n$ – the subset of non-critical paths
$\mathcal{N}$ – the set of all components of the metaprogram
$\mathcal{N}_c$ – the subset of the potentially critical components
$\mathcal{N}_n$ – the subset of non-critical components
$t_i$ – the estimated execution time of component $i$ in schedule S
$t'_i$ – the actual execution time of component $i$ in schedule S
$t_{s_i}$ – starting time of component $i$ in schedule S
$t_{f_i}$ – the completion/end time of component $i$ in schedule S
$t_{max_i}$ – the upper bound of the execution time of component $i$
$t_{spare_i}(A \rightarrow B)$ – the spare time of the link from component A to B
$\sigma_i$ – the slack of component $N_i$
$\bar{\sigma}_i$ – the adjusted slack of component $N_i$

Given a directed acyclic graph $V(N, E)$ the nodes $N = \{N_1, N_2, ...N_n\}$ of this graph are called *components* and the edges *data paths*. The acyclic graph is called a *metaprogram*. An *ordering* of the nodes is a permutation $P$ of the nodes $\{N_{P_1}, N_{P_2} ... N_{P_n}\}$ which preserves the order of the nodes in the graph, i.e. if there is an edge $N_i N_j$ in the graph, then $P_i < P_j$.

Given a *grid* $G = \{H_1, H_2, ..., H_k\}$ consisting of $k$ hosts, and a metaprogram $V = (N, E)$ a *mapping of the component* $N_i$ to the grid is a function associating a unique host in $G$ to the program, $Map(N_i) = H_j$. The *mapping of the metaprogram to the grid* is a set:

$$Map(V, G) = \{Map(N_i)\} \;\; \forall N_i \in N \tag{1}$$

We associate with each pair $(N_i, H_j)$ with $N_i \in N$ and $H_j \in G$ a scalar called the *running time of program* $N_i$ *on host* $H_j$, $t_i = RunTime(N_i, H_j)$. The *running time of the metaprogram components on the grid* is the set: $RunTime(V, G) = \{RunTime(N_i, H_j)\} \;\; \forall N_i \in N, \;\; H_j \in G$.

Given a metaprogram $V$, a grid $G$, an ordering of the programs, $P$, and a mapping $Map(V, G)$ we associate to each program a scalar value called the *startup time* $t_{s_i} = Start(N_i)$. The *startup time of the metaprogram components* is the set $Start(V) = \{Start(N_i)\} \;\; \forall N_i \in N$. The *completion/end time* of a component is defined as $End(N_i) = t_{f_i} = t_{s_i} + t_i$ and the set $End(V) = \{End(N_i)\} \;\; \forall N_i \in N$ is called the *the completion time of the metaprogram components.* The *goal component* of a metaprogram is the component whose result is the output of the entire computation, it is the last executed component, and its completion time $t_{f_g}$ coincides with the completion time of the metaprogram.

The following restriction applies: the execution of any two components $N_i$ and $N_j$ of a metaprogram mapped onto the same host, $Map(N_i) = Map(N_j)$, cannot overlap:

$$P_i < P_j \Rightarrow t_{f_i} + t_i \leq t_{s_i} \tag{2}$$

Given a metaprogram $V(N, E)$, the grid $G$, and the set $RunTime(V, G)$ a *schedule* of the metaprogram on the grid is the triplet consisting of the ordering, mapping and starting times of all the components of the metaprogram, $(P(N), Map(V, G), Start(V))$. The *total running time of a schedule* $S$ *on grid* $G$ is $T(V, G, S) = \max_i(t_{f_i}) \;\; \forall N_i \in N$.

The *deterministic optimal scheduling problem* - given a metaprogram $V$ and the set of the running times of its components on a grid, $RunTime(V, G)$ find the schedule which minimizes the total running time, $T(V, G)$. The deterministic nature of a schedule is due to the fact that the values in the set $RunTime(V, G)$ are deterministic.

The *nondeterministic metaprogram scheduling problem* is a variant of the deterministic metaprogram scheduling problem when we assume that the execution times in the set $T(V, G, *)$ are random variables with known distributions and we try to find the schedule which minimizes the the total running time. For static scheduling approach, we can only hope to minimize the *mean execution time* over a number of runs.

In practice, the distribution of the execution time of a program is difficult if not impossible to obtain. Analytical expressions can only be obtained for some special cases unlikely to be of interest. A sample distribution requires empirical knowledge about the program and the *execution history of the program.*

Our model does not account for data migration delays because we are primarily concerned with coarse-grain distributed computing on a high-speed, low-latency network. The analysis may be extended however to models where data migration has a significant impact on the execution times.

In the next section we introduce the concept of robustness and describe a technique to determine the tolerance of a schedule to the hazards. An $O(n^2)$ time algorithm is given. First, we present the application of the robustness analysis for two scenarios for a real-time and a non critical system. Then we present experimental results of simulations demonstrating the usefulness of our approach.

## 2 The robustness of a schedule

An example illustrates the concept if robustness of a schedule. In Figure 1 we present a metaprogram together with the estimated execution times of its components. These estimates refer to a reference computer, and should be adjusted to take into account the performance of the computer on which the execution takes place. The communication delays are ignored. Suppose that we want to schedule the program on a grid consisting of `H1`, `H2` and `H3`. The reference computer is `H1`, while the speed of `H2` is 50% and of `H3` 25% of the reference.

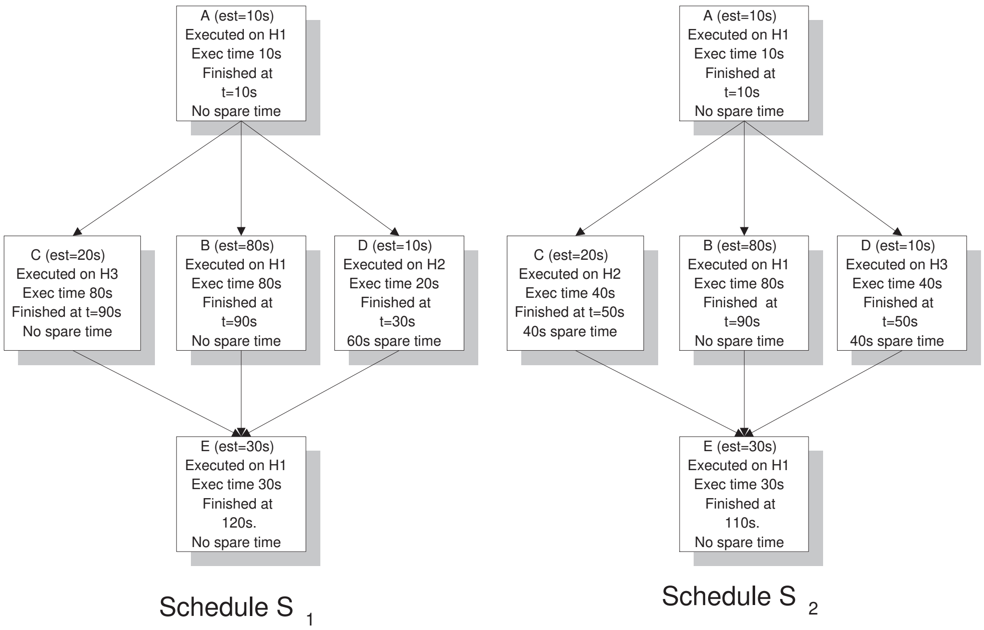


Figure 1: Two schedules of the same metaprogram with the same execution time but different robustness. The metaprograms are executed on the hosts H1, H2 and H3 with the relative speed 1.0, 0.5 and 0.25 respectively.

Consider two optimal schedules with the same execution time of $t = 120s$ for the given grid (we leave the proof of the optimality to the reader). A more attentive examination of the two schedules reveals an important difference among them. For $S_1$, component `D` finishes at the time $t = 30s$, while its output is needed at time $t = 90$ and the host on which it is scheduled (`H2`) is not used again in this metaprogram. In this case we say that the component `D` has 60 seconds of *spare time.* Even if the execution of `D` takes twice the estimated value, the total execution time of the metaprogram will not be affected. Unfortunately, none of the other components in this schedule have spare time; we say they are *critical.* For $S_2$ both `C` and `D` have 40 seconds of spare time.

Intuitively, it is obvious that the schedule on right, $S_2$, is "better" than $S_1$, the schedule on left of Figure 1. To provide a quantitative assessment of the difference between the two schedules assume a probability say $p = 0.2$ that a component is late and that the execution times are independent random variables. The last assumption may not be true in practice systems, however it is a good approximation in cases when the delays are caused by discrete events independent upon the distributed application.

The probability that the metaprogram is late for the two schedules are

$$p_{late}(S_1) = 1 - (1-p)^4 = 0.5904$$

$$p_{late}(S_2) = 1 - (1-p)^3 = 0.4880$$

If the metaprogram is executed repeatedly, then then the expected running time of a more robust schedule will be smaller than the expected running time of the less robust schedule, assuming that the running time of each component has small variations around its expected value.

Although our model of the execution times is naive, the qualitative result will apply to any reasonable narrow distribution of the execution times. We assumed a bounded execution time in order to prove the robustness of the algorithm. In practice a weaker assumption along the lines of a very narrow distribution around the estimated execution time should lead to the same result.

## 2.1 Data and host dependencies

In the following we devise an analytic measure of the vulnerability of a schedule to hazards. We are interested in the question how the increase in the execution time of a component affects the total execution time of the metaprogram, or in the positive effects of an early termination.

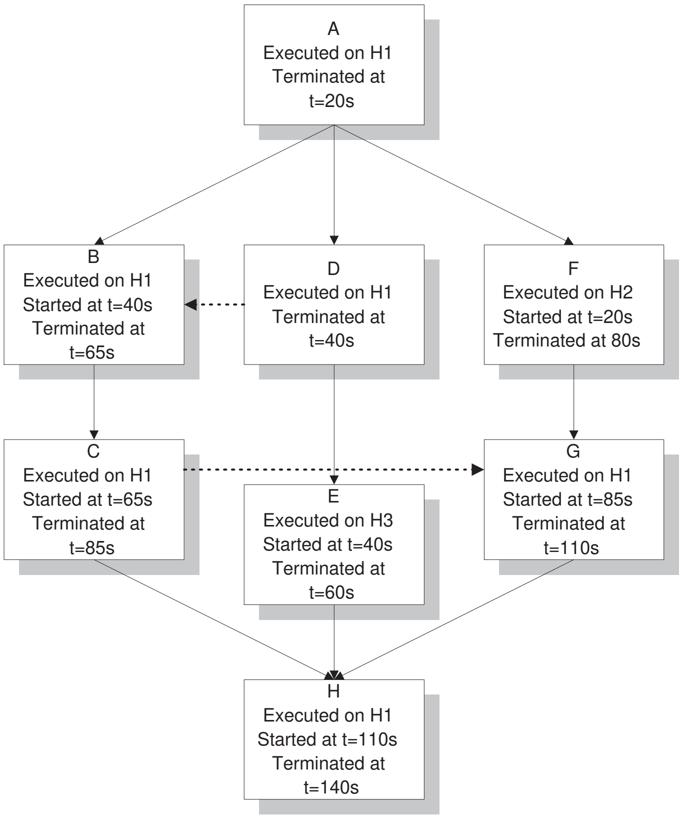


Figure 2: Augmentation of the data dependency graph of a metaprogram with links corresponding to the host dependencies. The host dependencies are drawn with dotted lines.

Given a component $N_i$ of the metaprogram its starting, execution, and completion time are respectively $t_{s_i}$, $t_i$, and $t_{f_i}$. If the completion time is exceeded we say that component $N_i$ is *late*. There are two reasons for a component to be late:

- The actual execution time of the component is longer than expected, $t'_i > t_i$, and/or
- The execution of the component begins later than expected: $t'_{s_i} > t_{s_i}$ due to interactions among the entities involved. We recognize *(a) data dependencies* some component was late in creating data needed for the execution of component $N_i$, *(b) communication dependencies*, the data transfer between the producer and the consumer takes longer than expected, and *(c) host dependencies* some component was late in completing its execution on the same host where component $N_i$ will be executed.

Host dependencies may be resolved by a dynamic scheduler, that can reassign the component to a different host. This is a nontrivial problem on its own, because the modifications in the mapping may lead to large performance penalties, and the real delay is not known the moment of the scheduling. We may sacrifice our precomputed optimal schedule for an insignificant delay. In this paper we deal only with precomputed static schedules. While data and communication dependencies are invariants of the metaprogram, the host dependency is a property of a particular schedule.

In the following we introduce a robustness metrics for the schedules. In this case data and host dependencies can be treated identically. Our approach is to *augment the data dependency graph* of the metaprogram with the host dependency links as shown in Figure 2. If two programs are scheduled one after another on the same host, a new link is be added between them. If a data dependency link between the two components is already in place no new link will be added. For optimal schedules most of the host dependencies follow the data dependencies because optimal schedules try to avoid moving data around.

Figure 2 illustrates the effect of host dependencies. Component `C` terminates at $t = 85s$, and the data it generates is needed only by component `H` which starts at $t = 110$, so one is tempted to believe that we have a comfortable spare time for `C`. However the host dependency link from `C` to `G` shows that any delay in terminating `C` will delay the start and implicitly the termination of component `G` on the host `H1`. Component `G` is critical, so the host dependency will make the component `C` critical, too.

In our analysis we do not differentiate between delays caused by a late start or longer execution time.

Given a metaprogram and a schedule, a *shifted schedule* is one where the mapping and order of execution of the components on every host is the same, but the startup of a component is adjusted such that it is launched immediately when its data and host dependencies are satisfied. We assume that the scheduler can automatically shift the schedule, if needed.

An upper limit of the effect of the delay is given by the following theorem.

**Theorem 1** *Given a metaprogram V, a grid G, and a static schedule S, let the execution time of each component be $t_i$, and the total execution time be $T(V, G, S)$. If the actual execution time of each component changes by $\Delta t_i$ there is a schedule S' whose execution time is smaller than:*

$$T(V, G, S) + \sum \Delta t'_i$$

*where*

$$\Delta t'_i = \begin{cases} \Delta t_i & \text{if } \Delta t_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

Proof:

Consider a schedule $S'$ where the order of the execution of the components is identical with $S$, but the starting time of the components is $t'_{s_i} \geq t_{s_i}$. We call such a schedule a *shifted schedule*.

For each component we have the starting and finishing time:

$$t'_{s_i} = t_{s_i} + \Delta t_{s_i} \qquad t'_{f_i} = t_{f_i} + \Delta t_{s_i} + \Delta t_i = t_{f_i} + \Delta t_{f_i}$$

The value $\Delta t_{f_i} = \Delta t_{s_i} + \Delta t_i$ represents the lateness of completion of component $i$ relative to the deterministic case.

We will consider that shifted schedule where each schedule is shifted with the amount just enough for it's dependencies to be satisfied. For every component $i$ we have

$$\Delta t_{s_i} \leq \max_{j \in Dep(i)} \Delta t_{s_j}$$

where $Dep(i)$ denotes the set of components $C_i$ depends upon.
We use induction to prove that $\Delta t_{f_i} \leq \sum_{j=0}^{i} \Delta t_i$ for this schedule.
For $i = 1$ we have $\Delta t_{s_1} = 0$ and $\Delta t_{f_i} = \Delta t_i$, which verifies our assumption.
We assume that for a particular $i$ the assumption is verified: $\Delta t_{f_i} \leq \sum_{j=0}^{i} \Delta t_i$.
For i+1 we have

$$\Delta t_{f_{i+1}} = \Delta t_{i+1} + \Delta t_{s_{i+1}} \leq \Delta t_{i+1} + \max_{j \in Dep(i)} \Delta t_{s_j} \leq \Delta t_{i+1} + \sum_{j=0}^{i} \Delta t_j \leq \sum_{j=0}^{i+1} \Delta t_j$$

For the schedule built in this way we have $\Delta t_{f_i} \leq \sum_{i=0}^{n} \Delta t_i$. **q.e.d.**

This theorem provides an upper limit for the delay of the schedules. However this upper limit is very disappointing, and raises the question if it cannot be improved. Consider a critical path $P_{crit} = \{C_1, C_2 ... C_n\}$ where $C_1$ is the starting component and $C_n$ is the goal component of the metaprogram. Obviously a lower limit of the total execution time would be:

$$T(V, G, S) + \sum \Delta t_{C_i}$$

This lower limit shows us that if a component on the critical path is late, this delay will propagate into the total execution time of the metaprogram. A metaprogram can also be late because of components outside the critical path. In this paper we develop methods for reducing the chance that a metaprogram is late due to components that are not on the critical path constructed assuming deterministic execution times.

Unfortunately, the influence of a late termination is more probable to affect the schedule than an early termination. This is due because most components have more than one dependency. Any late dependency forces the component to be late, while all dependencies should be early to permit an early start of the component.

## 2.2 Spare time

Consider a metaprogram and a static schedule. Given component $A$ we want to determine the way in which the delay of component $A$ influences the execution time of the metaprogram.

We assume that there is a path from each component to the goal component and there is only one goal component. If there are more then one, we introduce a *final goal component* which depends on all of the goal components, and has zero execution time.

*The spare time of a link* is defined as:

$$t_{spare}(A \rightarrow B) = t_{s_B} - t_{f_A}$$

The **slack of a component** $\sigma_i$ is the minimum spare time on any path from the component to the goal component, or equivalently the length of the shortest path to the goal component in the augmented graph of spare times. We call a component **critical** if its slack is zero. Even if the spare time of a component is zero, its slack can be nonzero. The importance of the slack is demonstrated by the following theorem.

**Theorem 2** *Consider a metaprogram $M$, a schedule $S$ and a component $N_i$ with a nonzero slack $\sigma_i$. If component $N_i$ will exceed its estimated execution time by $\Delta t_i \leq \sigma_i$ the shifted schedule will have the same execution time as the original one, provided that all other components meet their deadlines.*

**Proof:**

We use a constructive proof, by building the corresponding shifted schedule. We modify the starting time of each component which depends on component $N_i$. First, we observe component $N_i$ cannot be the last component, because the slack of the last component is always zero.

Assume we have a path from component $N_i$ to the final component $N_n$, $P = C_1C_2...C_m$, where $C_1 = N_i$ and $C_m = N_n$.

Because the slack of $N_i$ is $\sigma_i$ we have

$$\Delta t_i \leq \sigma_i \leq \sum_{j=1}^{m-1} t_{spare}(N_{C_j} \to N_{C_j+1})$$

Now we change the starting time of each component on the path as follows:

$$t'_{s_{C_i}} = max(t_{s_{C_i}}, t_{s_{C_i}} + \Delta t_i - \sum_{j=1}^{j=i-1} t_{spare}(N_{C_j} \to N_{C_j+1}))$$

In this case the starting time of the component $C_m = N_n$ will be:

$$t'_{s_{C_m}} = max(t_{s_{C_m}}, t_{s_{C_m}} + \Delta t_i - \sum_{j=1}^{j=m-1} t_{spare}(N_{C_j} \to N_{C_j+1}) \leq t_{s_{C_m}}) = t_{s_{C_m}}$$

but

$$\sum_{j=1}^{j=m-1} t_{spare}(N_{C_j} \to N_{C_j+1}) > \Delta t_i$$

From here we have $t'_{s_{C_m}} = t_{s_{C_m}}$.

We do this computation for every path $P$ from $N_i$ to $N_n$, and take the maximum value of the $t'_{s_j}$ for every component. Nevertheless, for each path we obtain the same result for $t_{s_{N_m}}$ Which means that the last component will start on time, so the schedule is not delayed.

We still have to prove that the schedule modified in this way by changing on every path it is a valid schedule, i.e. all the dependencies for the elements on the path considered are satisfied. We shall prove this by contradiction.

Assume that element $C_i$ on path $P$ does not have it's dependencies satisfied at the calculated starting time $t'_{s_{C_i}}$. Because $t'_{s_{C_i}} > t_{s_{C_i}}$ this can happen only if there is another path from $C_1$ to $C_i$, $C'_1C'_2C'_r$ with $C'_1$ being $N_i$ and $C'_r$ being $C_i$, where the dependencies are not satisfied. This means that the starting time of $C_i$ is earlier than the finishing time of the preceding component on the path:

$$t_{s_{C'_i}} < t_{f_{C'_{i-1}}}$$

We can now build a path from $N_i$ by going to $C_i$ on the second path and from than on in the first path to the last component of the schedule. However this path was considered before, so the $t'_{s_i}$ is specified such that the dependency is satisfied in this path. This is a contradiction to the previous assumption.

We have proved in this way that the shifted schedule built by modifying the starting times using the algorithm described above is a valid schedule and the execution time is identical to the execution time of the original schedule.

**Corrolary 1** *If a component $N_i$ has at least one non-critical component on all paths to the goal state, the component is non-critical.*

This definition gives us a simple algorithm to compute the slack of all components of a metaprogram.

```
1. augment the metaprogram graph with the host dependency links.
2. label each link with the spare time on the link.
3. for each component compute the shortest path to the goal
   component in the graph of the spare times.
```

The spare time can be computed using Dijkstra's shortest path algorithm or improvements of it in $O(n^2)$ time [6]. The intuition behind the slack is that spare time on the shortest path from a component to the goal component may back-propagate and allow a component with no spare time to be late.

## 2.3 Adjusted slack

One of the drawbacks of using the slack of a measure of the robustness of a schedule, is the fact that every component is treated separately. Theorem 2 proves the slack of a component as an useful measure only for the case when all other components meet their deadlines, a rather strong assumption. In a practical case more than one component can be late. A delay in a component on which our component depends may cause a decrease in the slack of the current component. Intuitively, the same slack $\sigma$ is better at the beginning of a computation, than close to the end, where possible delays from earlier components can "chop of" parts of it.

We are interested in a measure which proves a result similar to Theorem 2, but allows more than one component to be late. In the general case it is difficult to construct such a measure because it depends on the distribution of the execution time of the previous components. In the following we assume that the actual execution time of component $N_i$ does not exceed its estimated execution time by more than a factor $q \geq 1$, $t'_i \leq q \times t_i$ and that q is the same for all components. This assumption bounds the starting time of a component as well

$$t'_{s_i} \leq q \times t_{s_i}$$

A bounded time metaprogram is one where the execution times of all components are bounded.

We define the *adjusted slack* as the slack modified to account for a late start.

$$\bar{\sigma}_i = \max(0, \sigma_i - (q-1)t_{s_i})$$

**Theorem 3** *Consider a bounded time metaprogram V and a schedule S. If $\Delta t_i < \bar{\sigma}_i \;\; \forall N_i \in N$ then there is a shifted schedule S' with the same execution time as the original schedule S.*

**Proof:**

We apply the delays one by one, creating a set of schedules $S^{(0)}, S^{(1)} \ldots S^{(n)}$ where $S^{(0)} = S$ and $S^{(n)} = S'$. We want to prove by induction that the termination time of $S^{(n)}$ is equal with the termination time of $S^{(0)}$ , $t_n^{(0)} = t_n^{(n)}$.

We prove this by induction. The first step of induction is $t_n = t_n^{(0)}$.

Assume that schedule $S^{(i-1)}$ has the same termination time, $t_n^{(i-1)}$. Moving to schedule $S^{(i)}$ we apply the delay $\Delta t_i$ to component $N_i$. This component had a slack $\sigma_i$ in the original schedule $S$, which may be decreased by the previous delays applied: $\sigma_i^{(i)} = \sigma_i - \Delta\sigma_i$. However we know that the slack was decreased with less than the limit of delay occurred in the preceding steps, which is exactly $(q-1)t_{s_i}$. Moreover, this slack can not be negative. Negative slacks can happen only for invalid schedules and the way in which we built the consecutive schedules assures that they are valid. We have:

$$\sigma_i^{(i)} = \max(0, \sigma_i - (q-1)t_{s_i})$$

exactly the adjusted slack $\sigma_i'$. However, $\Delta t_i < \sigma_i'$ from the conditions of the theorem. We can use Theorem 2 to build the new schedule $S^{(i)}$ with the same termination time. It follows that all schedules $S^{(i)}$ have the same termination time and $S^{(n)} = S'$ has also the same termination time. **q.e.d**

This theorem shows that the adjusted slack is a more useful metric than the slack, because allows any of the components to be late within the limits of its its adjusted slack.

We call a component *safe* if there its adjusted slack is larger than the upper bound of delay on the component. A safe component can not cause the total execution time to be late (provided that the upper bound on delays holds). An immediate application of this analysis is the identification of the safe components.

### 2.4 Potentially critical paths and components

Call $T_i$ the cost of a path $P_i$, $h_i$ the cumulative effect of the hazards on that path, and $T_i' = T_i + h_i$ the cost of the path in the presence of the hazards, and write $P_i = (T_i, h_i)$.

The effect of the hazards partitions the set $\mathcal{P}$ of all paths into two disjoint subsets, $\mathcal{P}_c$ paths that have the potential of becoming critical path, and $\mathcal{P}_c$ paths that cannot become critical, $\mathcal{P} = \mathcal{P}_c \cup \mathcal{P}_n$ such that

$$\min(T_i',\ \forall P_i \in \mathcal{P}_c) > \max(T_j',\ \forall P_j \in \mathcal{P}_n)$$

We call a component $N_i$ potentially critical if it appears on at least one potentially critical path. The set of components is $\mathcal{N} = \mathcal{N}_c \cup \mathcal{N}_n$ with $\mathcal{N}_c$ the set of potentially critical components and $\mathcal{N}_n$ components that are not critical.

The analysis supported by Theorems 2 and 3 can be restricted only to components in $\mathcal{N}_c$. Theorem 1 provides a tighter bound when applied only to components in $\mathcal{N}_c$. For example consider a metaprogram with 3 paths $P_1 = (1000, 10)$, $P_2 = (10, 600)$ and $P_3 = (60, 800)$. In this case $\mathcal{P}_c = \{P_1\}$ and $\mathcal{P}_n = \{P_2, P_3\}$. Assume that hazards effect only one component in each path, and these components are different. Applied to the entire set of components $\Delta t' = 1410$, and applied to $\mathcal{N}_c$ we have $\Delta t' = 10$.

The concept of robustness of a schedule can be expressed in terms of path criticality. If we call $p_i^{(h)}$ the probability that path $P_i$, $\quad 1 \leq i \leq n$ becomes critical subject to hazards $h$ and $\sum_{i=1}^{n} p_i = 1$ then we can define the *entropy* of a schedule as

$$H^{(h)}(S) = -\sum_{i=1}^{n} p_i^{(h)} \log p_i^{(h)}$$

If a schedule with $n$ paths has only one critical path say $P_1$ then $p_1 = 1$, $p_i = 0 \quad i = 2..n$ and we have $H(S) = 0$. If all n paths can become critical subject to hazards $h$ then $p_i = \frac{1}{n}$ and $H(S) = \log n$.

The entropy of the path is then a measure of robustness. We say that schedule $S_j$ is more robust than $S_i$ if

$$H^{(h)}(S_i) < H^{(h)}(S_j)$$

**Example:**

Let us label the three paths of schedule $S_1$ in Figure 1 from left to right as $P_1$, $P_2$ and $P_3$. Assuming that the hazards lead to an increase of the execution time no more that 30s, we have

$$p_1 = p_2 = \frac{1}{2} \quad p_3 = 0 \quad H(S_1) = 1$$

For $S_2$ the paths are $P_1'$, $P_2'$ and $P_3'$ and

$$p_1' = p_2' = 0 \quad p_3 = 1 \quad H(S_2) = 0$$

# 3 Constructing robust schedules

In this section we show how the robustness analysis can be used to improve a scheduling algorithm. Given a metaprogram $V$ and a grid $G$ we can partition the set of all schedules into equivalence classes based upon the total execution time of the metaprogram. Schedules with the same execution time form a class of iso-schedules, they either have the same critical path or equal cost (execution time) critical paths and in a deterministic case are indistinguishable from one one another but exhibit different performance under non-deterministic component execution time assumptions. In this paper we discuss metrics to differentiate amongst the members of a class based upon the robustness.

Two scenarios are studied: in the first one we assume a real time system where our goal is to maximize the number of safe components, while in the second one we consider a common metaprogram where the objective is to minimize the average execution time over a number of runs.

## 3.1 Scenario 1: Real time system

In this scenario we consider a real time distributed system, where meeting the deadlines is critical.

A component can be executed using static allocation and spatial partitioning of the grid. We call this case *strict scheduling conditions* and assume that a component scheduled this way will meet its deadline. An alternative way of executing a component on a grid is by dynamic allocation based upon temporal partitioning of the grid (*weak scheduling conditions*). This execution mode is more efficient from the point of view of resource utilization, but may cause delays in the execution of the components because a component may have to wait for a resource used by another component. We will assume that even under weak scheduling conditions we have an upper bound of the execution time $t_{max} = q \times t$.

Strict scheduling decreases the throughput of the system. We are interested in minimizing the number of components which need strict scheduling.

Theorem 3 shows that if the adjusted slack of a component is larger than the bound on execution time (safe components), the possible variation in the execution time can not affect the total execution time, regardless of the rest of the system. This implies that a safe component does not require strict scheduling.

Our goal is to maximize the number of safe components, and to keep the estimated execution time smaller or equal than the deadline. An outline of a timeout limited heuristic search robust algorithm is given below:

```
while(not timeout)
  1. find a new schedule according to the heuristic
  2. if the execution time is longer than the deadline, reject it
  3. otherwise
```

```
        3.a perform robustness analysis
        3.b compute the number of safe components
        3.c if number of safe components larger
              than in the current schedule
           {
                make it the current schedule
           }
return current schedule
```

The same principle can be used to create a genetic algorithm for the same objective function.

Identification of safe components has important consequences upon resource utilization in case of strict scheduling conditions. Figure 3 shows the resource allocation graph for the components of the grid. The graph (a) shows the *static allocation* of all the resources of the grid for the time of execution of the metaprogram. The gray contour on the graph (b) shows the actual time intervals where different components are executed on the elements of the grid. The lower boundary of the contour is formed by the starting times of the first component, while the upper boundary by the finishing times of the last component executed on the specific host. Inside this we have a hashed contour which represent the time frame where critical components are executed. This region represents the time and space frame where resource allocation is required. In the gray region only safe components are executed, which does not require resource allocation, while in the white region no component of the metaprogram is executed. The hashed region corresponding to the strict resource allocation requirements still span the entire time between the start and end of the metaprogram - corresponding to the critical path of the schedule. The robustness analysis permits a *dynamic allocation of resources* by identifying components outside the critical path which do not require strict resource allocation and/or selecting the schedules which maximizes the number of such components.

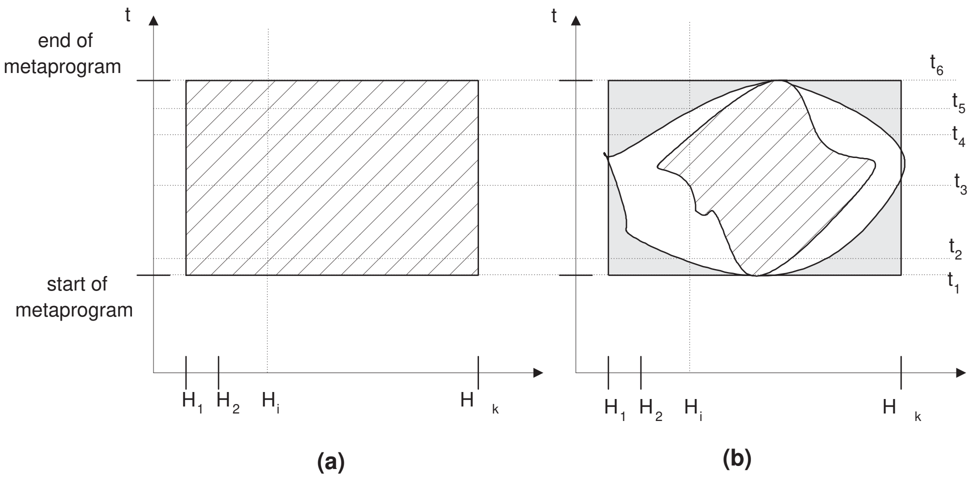


Figure 3: The effect of the robustness analysis upon the resource allocation. On the schedule on left host $H_j$ is allocated exclusively to the metaprogram at time $t_1$ and deallocated at time $t_6$. On the schedule on the right the resource is available for other tasks during the intervals ($t_1$, $t_2$) and ($t_5$, $t_6$), it executes non-critical tasks in ($t_2$, $t_3$) and ($t_4$, $t_5$) and critical tasks in ($t_3$, $t_4$)

### 3.2 Scenario 2: Common metaprograms

In this scenario we are assuming that our system is not critical (i.e. there is no deadline). We are still assuming a bounded distribution of the execution time around the estimated value. In this case we are interested in obtaining a minimal average execution time for the distributed application for a number of runs.

To design a search algorithm we need a unique *robustness measure*, to compare schedules. A good robustness measure should have the following properties:

- Increase in the components slack.
- Penalize the slack larger than the time bound.

Unfortunately, devising a measure which accurately predict which of the iso-schedules gives the minimal average execution time depends on the shape of the distribution of execution times, an information difficult to obtain in practice. We are proposing an empirical formula which has the advantage of being simple, easy to compute, and performs well in experiments.

$$R(S) = \sum_i \frac{min(\sigma_i, t_i^{upper} - t_i)}{t_i^\gamma}$$

where $\gamma \in (0, 1]$ is a constant depending on the shape of the distribution and can be determined experimentally.

An outline of a timeout-limited heuristic search robust algorithm is given below:

```
while(not timeout)
  1. find a new schedule according to the heuristic
  2. if the execution time is longer than the current schedule
        reject it
  3. if it is shorter make it the current schedule
  3. if they have equal length
        3.a perform robustness analysis on the new schedule
        3.b compute the robustness measure R(S) for the new schedule
        3.c if it is larger than for the current schedule
               make it the current schedule
return current schedule
```

## 4 Experimental results

In this section we study two questions:

- What is the cost of taking into consideration the robustness in scheduling?
- What is the average improvement achievable by robust scheduling?

The answer of these question depends on the nature of applications, the distribution of the execution times, so a general answer is difficult to give. An extensive testing process was performed using the Bond environment [8, 7] to build up some confidence in the results.

All examples presented in these article were hand-crafted in order to prove the validity of ideas. Nevertheless they did not answer an important question: how often in practice the possibility of improving the schedule using the robustness metric arises, and how dramatic these improvements are? These questions can not be answered without a knowledge of the application domain, some type of metaprograms may permit more optimizations than others.

Our experiments were made using randomly generated schedules. We were motivated by the desire to provide a fair evaluation of the algorithms provided in this paper. Carefully selected schedules may favor an algorithm with a poor performance on most schedules.

The random schedules were generated using following algorithm:

```
1. The number of components NC is generated as a random number
   in the range 15..30. The type of each component was chosen randomly
   from a collection of 5 different components.
2. The number of links are generated in the range 1.. NC*(NC-1)/2
3. The links are generated by generating random pairs of numbers.
   The connection always goes from lower ranking to higher ranking
   component, in order to avoid cycles.
```

In the testing process 100 random metaprograms were generated. For each metaprogram a time-out limited optimal static scheduling algorithm was applied. The algorithm was modified to maintain the optimal schedule according to the robustness measures presented for the two scenarios presented in Section 3. The robustness analysis was performed only if needed (i.e. if the decision could not be done using the total execution time). Every schedule was run 200 times with the execution times as a random normal distribution with the mean being the estimated execution time, and the upper bound being $q = 1.4$.

We have collected the following data:

- The number of schedules checked, and the number of iso-schedules found.
- The number of schedules for which the robustness analysis was performed.
- The best and worst number of safe components.
- The minimal average execution time, and the execution time of the first optimal schedule found (which would have been retained without the robustness analysis)

In our algorithm the robustness analysis was performed only when the total execution times were identical.

The robustness analysis was performed 129,683 times for the 8,768,794 schedules generated, representing less than 1.5% of the cases. We can conclude that the robustness analysis is easy to implement and cheap in respect to the computational demand.

The Table 4 shows the relative improvement in the number of safe components and mean execution time.

| | no robustness analysis | with robustness analysis |
|---|---|---|
| No. of safe components | 5.23 | 8.74 |
| Mean execution time for 200 runs | 68.93s | 67.91s |

Table 1: The effects of the robustness analysis on the number of safe components and the mean execution time of a metaprogram - the numbers represent the average of 100 randomly generated metaprograms

The results show an average of 67% increase in the number of safe components. The improvement in the execution time is not as spectacular, being in the range of 1.5%. Nevertheless, 1.5% improvement in the execution time represents approximately 10% improvement in the expected delay. For the best case tested the number of safe components was improved from 3 to 16, and the mean execution time decreased by 6%. In the worst case there was no improvement in the indicators. The variation of the results is caused by the internal structure of the metaprograms. The robustness analysis cannot create spare times, it can only redistribute it to places where is needed, and even these redistribution has specific limitations.

As a side effect of the robustness analysis the safe components of a schedule are identified. Even if we can not increase their number, due to the specifics of the metaprogram, by identifying the safe components we can employ dynamic resource allocation, reducing the cost of running the metaprogram.

As a conclusion, robustness analysis is a practical method to improve the quality of metaprogram scheduling algorithms. We argue that the robustness analysis can be integrated into scheduling algorithms with ease and at a low cost and that the gain in terms of better quality schedules outweighs the computation and implementation costs.

# 5 Conclusions

Scheduling of dependent tasks with deterministic execution times is known to be NP complete. The problem of scheduling tasks whose execution time is non-deterministic as a result of various hazards is conceptually more challenging.

Given a schedule for an augmented dependency graph one can construct classes of iso-schedules, schedules with the same cost of the critical path (the cost of the critical path is the elapsed time from the initial component to the goal component). Some of the schedules in one class are less sensitive to the effects of the hazards because components on non-critical path have some spare time or slack in our terminology. The startup time of such a component can be delayed or its execution time may take longer in such a "shifted schedule", without increasing the total execution time of the entire graph. We call this property of a schedule robustness.

In this paper we assume bounded variations of the execution time of components and devise an analytic measure of the vulnerability of a schedule to hazards. We provide an upper bound for the execution time of a schedule subject to hazards and prove two theorems regarding shifted schedules.

We introduce the concept of a critical component, one whose increase of the execution time due to hazards may cause the execution path to become critical. Then we discuss measures of robustness. A possible measure is the number of critical components within a schedule, the fewer, the more robust the schedule. An alternative measure of robustness introduced in this paper is the entropy of a schedule. The entropy of a schedule is based upon the probability of an execution path of becoming critical. In the general case, determining this probability is a non-trivial task and more research is needed before this measure may prove its usefulness.

Then we present algorithms for constructing robust schedules and report experimental results. We show that robustness analysis is not computationally intensive. We use randomly generated graphs in our experimental setup because we want a fair evaluation of the analysis algorithm. As a result of the robustness analysis in our experiments the number of safe components was increased by 67% and the total execution time decreased by a modest 1.5% in average. We could probably obtain better results on specially crafted schedules.

# 6 Acknowledgments

The research reported here is partially supported by the National Science Foundation grant MCB-9527131, by the Scalable I/O Initiative, by a grant from the Computational Science Alliance, and by a grant from Intel Corporation.

## References


[1] L. Bölöni, K. K. Jun, and D.C. Marinescu: *QoS and Reliability Models for Network Computing* Department of Computer Sciences, Purdue University CSD-TR #97-051, 1997

[2] F. Berman *High Performance Schedulers*, in "The Computational Grid: Blueprint to a New Computer Infrastructure", I. Foster and C. Kesselman Eds., Morgan Kaufmann, 1998.

[3] T. L. Casavant and J. G. Kuhl *A Taxonomy of Scheduling General-Purpose Distributed Computing Systems* IEEE Trans. on Software Engineering, 14(2), pp. 141-154, 1988.

[4] Kuei Yu Wang, D. C. Marinescu, and O. F. Carbunar: *Dynamic Scheduling of Process Groups*, Concurrency: Practice and Experience, Vol 10(4), pp. 265-283, 1998

[5] J.R. Budenske, R.A. Ramanukan, and H.J. Siegel *On-line Use of Off-Line Derived Mappings for Iterative Automatic Target Recognition Tasks and a Particular Class of Hardware Platforms* Proceedings of the HCW'97 Heterogeneous Computing Workshop, pp. 74-82, April 1997

[6] Bertsekas, D.P. *Linear Network Optimization: Algorithms and Codes*, MIT Press, 1991.

[7] D.C. Marinescu and L. Bölöni *Reflections on Metacomputing, the Bond View* Purdue University CSD-TR #98-006, 1998

[8] L. Bölöni, K.K. Jun, M. Sirbu and D.C. Marinescu: *Seamless Metacomputing in Bond* Purdue University CSD-TR #98-010, 1998