# Scheduling of Replicated Tasks to Meet Correctness Requirements and Deadlines

Behrooz Parhami and Ching Yu Hung

Department of Electrical and Computer Engineering
University of California, Santa Barbara

## Abstract

*We consider a coarse-grained multiprocessing environment in which multiple task* copies *or unreliable* versions *(henceforth, referred to as task* instances*) need to be scheduled to run on unreliable processors in the face of correctness and timeliness requirements that are considered met if c task instances run to correct completion before the deadline d. In this paper, we study the interplay of correctness and timeliness requirements, providing examples of how scheduling policies that are optimal in other contexts can fail to meet correctness and timeliness needs in the above environment. We then present optimal scheduling policies for certain special cases of the above problem followed by a discussion of heuristics with reasonable performance in more general cases.*

## 1 Introduction

The problem of scheduling tasks on multiple processors in order to achieve minimal completion time or to meet deadlines is quite important and has received considerable attention (see, e.g., [10]). Similarly, the related problems of scheduling to reduce the response time in interactive systems and load balancing, which in its simplest form can be viewed as dynamic adjustments to a pre-computed schedule, have been researched extensively. Unfortunately, almost all interesting scheduling problems, some with as few as two processors, are NP-complete [11]. Thus the need for approximations and heuristic algorithms (see, e.g., [8] and the references therein).

Optimal scheduling of tasks becomes even more complicated if hardware and/or software faults are taken into account [5, 12]. Here, essentially, there is a competition for the use of resources (e.g., processors) between satisfying correctness and timeliness requirements. In other words, processing power can be used to execute more tasks (meeting more of the deadlines but with a lower level of confidence in the correctness of the results) versus executing tasks more reliably. Aspects of this latter concern are need-based scheduling of multiple task versions [2, 6], adjusting checkpointing intervals to strike a balance between waste of computational resources in the event of a detected fault and waste of the same due to checkpointing overhead [9], and, finally, tradeoffs between precision of results and their timeliness [1, 3].

We endeavor to extend and refine the framework presented in [6] by considering scheduling issues in a coarse-grained multiprocessing environment in which multiple task *copies* or unreliable *versions* (henceforth, referred to as task *instances*) need to be scheduled to run on unreliable processors in the face of correctness and timeliness requirements that are considered met if $c$ task instances run to correct completion before the deadline $d$. The only work known to us on the scheduling of redundant tasks deals with the establishment of performance bounds in the case of longest-processing-time-first scheduling policy[7].

Due to the unreliability of processors and possibly of task instances, it may be necessary to schedule more than $c$ instances to assure timely completion of a given task with high probability. However, if one schedules $c' >> c$ instances, sufficient resources may not be available to meet all deadlines.

To make the problem tractable, we need to make some simplifying assumptions.

- There are $m$ tasks to be executed, and $c$ instances of each must be correctly completed for a *successful run*. Three, successively less restrictive, timing models are assumed. In all three models, the computation times are deterministic; i.e., known a priori or upper-bounded (in which case our analyses are pessimistic), as opposed to being



0-8186-1060-3425/93 $03.00 © 1993 IEEE

randomly distributed.

- In the *homogeneous (common-deadline)* model, all tasks have identical computation time $r$, the same release or starting time $s = 0$, and the same deadline $d$.
- In the *common-deadline* model, task computation times $r_i$ can be different, while the release time $s = 0$ and deadline $d$ are the same for all tasks.
- In the *general deterministic* model, computation times, release times, and deadlines can all be different. They are assumed to be $r_i$, $s_i$, and $d_i$, respectively, for task $i$.

- There are $n$ processors in the system. The processors have identical processing speeds and are synchronized. We do not account for the time spent on scheduling, voting and communication.
- There is a fixed probability $p$ that an instance is correctly completed. Correctness of the results of each instance is thus assumed to be independent of the task or the processor.
- Whether an instance has produced correct results is determined solely by voting. When among the results of $c' \geq c$ instances, there is agreement among at least $c$ instances, the results are assumed correct. No information coding or consistency checking is required.
- Scheduling, communication, and voting are all fault-free.

Some of these assumptions may at first appear unrealistic. However, they are all necessary for tractability and in most cases adequately justifiable. For example, ignoring the scheduling overhead presents no problem with preset scheduling, since in this case, the one-time overhead can be considered as part of computation set-up time. In the general case, since we are concerned with coarse-grained multiprocessing, communication, scheduling and voting overheads can be accounted for by slightly inflating task running times. Similarly, the effects of communication, control, and decision errors can be modeled by assigning a suitably pessimistic value to $p$ so that, for example, the probability that $c$ or more task instances agree on an incorrect value is subsumed by the probability that they do not agree at all.

One can question the assumption of independence of fault occurrences. Studies have shown that most hardware malfunctions are caused by random, uncorrelated intermittent or transient faults. Voting is quite effective for masking such random faults. When permanent or catastrophic faults occur in the hardware, the system would normally be reconfigured to isolate the faulty unit(s). Software faults can cause incorrect results on the same or even different versions of a program, and would have to be dealt with by more sophisticated probabilistic models.

We assume that the correctness of computation results is determined solely by voting. Ideally, there is a reliable, yet simple, acceptance test that can be utilized to verify the correctness of each set of generated results. Then, adaptive scheduling schemes can yield significant benefits by avoiding the execution of extra instances with very high probability. In most practical cases, however, such an acceptance test simply does not exist. Thus, we will assume that agreement of computation results among $c$ instances is used to judge correctness. Although, strictly speaking, there is always a nonzero probability that $c$ instances *agree* on an incorrect result, this probability can be made negligibly small compared to other sources of error.

In this paper, we study the interplay of correctness and timeliness requirements, providing examples of how scheduling policies that are optimal in other contexts can fail to meet correctness and timeliness needs in the above environment. We then present optimal scheduling policies for certain special cases of the above problem, followed by a discussion of heuristics with reasonable performance in more general cases. Evaluation of the algorithms along with directions for further research concludes the paper.

## 2 The Homogeneous Model

We begin with the homogeneous (common-deadline) model. Every task instance has the same computation time $r = 1$, as well as the same release time $s = 0$ and deadline $d$. Both preset scheduling and adaptive scheduling are considered. With preset scheduling, we do not know in advance which scheduled instance will produce correct results, so it is optimal to distribute the resources evenly among tasks. With adaptive scheduling, we can concentrate the resources on tasks that have not yet had enough correct completions.

### 2.1 Preset Scheduling

With the preset scheduling strategy, we wish to find the optimal number of task instances to be scheduled

for execution in order to maximize the probability of successful runs. Since every task has the same timing parameters, it is intuitively obvious that the optimal strategy is to divide the resources evenly among the $m$ tasks. This is formalized in the following theorem.

**Theorem 1** *Let tasks be numbered from 1 through $m$ and $b_i$ denote the number of instances of task $i$ in an optimal schedule. If $cm \leq nd$, the optimal schedule has $\sum b_i = nd$ and, for any two $b_i$ and $b_j$, $| b_i - b_j | \leq 1$. In other words, resources are evenly distributed among all tasks.*

**Proof** Define the function $f(b)$ as the probability that a task having $b$ instances scheduled will have $c$ correct completions.

$$f(b) = \sum_{k=c}^{b} \binom{b}{k} p^k (1-p)^{b-k}.$$

The value of $f(b)$ is defined to be 0 for $b < c$ and is a strictly increasing function of $b$ for $b \geq c$. Let $P$ be the probability of a successful run. We have

$$P = \prod_{i=1}^{m} f(b_i)$$

Note that $cm \leq nd$ implies that there is an assignment of $b_i$'s so that $b_i \geq c$, and consequently $P > 0$. Furthermore, $n$ processors in time $d$ cannot execute more than $nd$ instances, so $\sum b_i \leq nd$.

The first assertion, $\sum b_i = nd$, is easy to prove by contradiction. To prove the next assertion, we compute:

$$\begin{aligned}
\Delta f(b) &\equiv f(b+1) - f(b) \\
&= \sum_{k=c}^{b+1} \binom{b+1}{k} p^k (1-p)^{b+1-k} - f(b) \\
&= \sum_{k=c}^{b+1} \left( \binom{b}{k} + \binom{b}{k-1} \right) p^k (1-p)^{b+1-k} - f(b) \\
&= (1-p) f(b) + p f(b) + \binom{b}{c-1} p^c (1-p)^{b-c+1} - f(b) \\
&= \binom{b}{c-1} p^c (1-p)^{b-c+1}.
\end{aligned}$$

Next, we show that $f(b)/\Delta f(b)$ is a strictly increasing function of $b$ for $b \geq c$.

$$\begin{aligned}
\frac{f(b)}{\Delta f(b)} &= \frac{\sum_{k=c}^{b} \binom{b}{k} p^k (1-p)^{b-k}}{\binom{b}{c-1} p^c (1-p)^{b-c+1}} \\
&= \frac{\sum_{k=0}^{b-c} \binom{b}{c+k} p^{c+k} (1-p)^{b-c-k}}{\binom{b}{c-1} p^c (1-p)^{b-c+1}} \\
&= \sum_{k=0}^{b-c} \frac{\frac{b(b-1)\cdots(b-c-k+1)}{(c+k)(c+k-1)\cdots 1}}{\frac{b(b-1)\cdots(b-c+2)}{(c-1)(c-2)\cdots 1}} \frac{p^k}{(1-p)^{k+1}} \\
&= \sum_{k=0}^{b-c} \frac{(b-c+1)(b-c)\cdots(b-c-k+1)}{(c+k)(c+k-1)\cdots c} \frac{p^k}{(1-p)^{k+1}}
\end{aligned}$$

The above expression is clearly a strictly increasing function of $b$ for $b \geq c$, and is positive. Thus we know that its reciprocal, $\Delta f(b)/f(b)$ is a strictly decreasing function of $b$ for $b \geq c$.

Now back to the problem of $m$ tasks. Assume that in some optimal schedule, the instances are not evenly allocated, i.e., there are $b_i$ and $b_j$ such that $b_i - 2 \geq b_j \geq c$. We want to show that substituting one instance of task $j$ for one instance of task $i$ would increase the success probability. Let $P'$ be the success probability for the schedule resulting from this substitution. We have:

$$\begin{aligned}
\frac{P'}{P} &= \frac{f(b_1)f(b_2)\cdots f(b_i-1)\cdots f(b_j+1)\ldots f(b_m)}{f(b_1)f(b_2)\cdots f(b_i)\cdots f(b_j)\ldots f(b_m)} \\
&= \frac{f(b_i-1)f(b_j+1)}{f(b_i)f(b_j)} = \frac{f(b_j+1)/f(b_j)}{f(b_i)/f(b_i-1)} \\
&= \frac{(f(b_j+1)-f(b_j))/f(b_j)\ +1}{(f(b_i)-f(b_i-1))/f(b_i-1)\ +1} \\
&= \frac{\Delta f(b_j)/f(b_j)\ +1}{\Delta f(b_i-1)/f(b_i-1)\ +1}
\end{aligned}$$

Since we know that $\Delta f(b)/f(b)$ is strictly decreasing, and $b_i - 1 > b_j$, the above expression is strictly greater than 1, leading to the conclusion $P' > P$. This contradicts the optimality of the original schedule. Therefore the numbers of scheduled instances must not differ by more than one in an optimal schedule. □

## 2.2 Adaptive Scheduling

With preset scheduling, we schedule some number of instances of each task and perform voting only after all scheduled instances are completed. There is some waste of processing power when, for certain tasks, more than $c$ instances turn out to be correct. With adaptive scheduling, it is possible to tell, some of the times, if $c$ correct instances of a task have already been completed and to concentrate our resources on the remaining tasks. The overall probability of successful

runs should be much higher than for preset scheduling. The price we have to pay is more voting and the extra scheduling activity during run time.

Even with the simplest model, there are many possible scheduling strategies:

- Selection of task instances can be *depth-first*, *breadth-first*, or *random*. Depth-first selection would concentrate on one task until the required number of correct completions are observed, then move on to the next task. Breadth-first selection would try to advance all tasks together.

- The number of instances to be immediately executed can be *projective* or *on-demand*. Assume that at one point, $b$ instances of some task have run to correct completion. On-demand scheduling would schedule $c-b$ instances, while projective scheduling would schedule more, say $g(c-b)$, instances, where $g$ is a suitable projection function. On-demand scheduling doesn't waste resources, but there is a possibility that some of the instances will not yield correct results and thus there will be some more demand in the next unit of time. Projective scheduling has a higher probability of obtaining an adequate number of correct completions sooner, but there is a possibility that some resources are wasted.

- Cooperation of the processors can be *global* or *partitioned*. Partitioned systems let each group of processors work on its share of task instances, and may be preferred if the network topology favors such partitioning, or if higher communication overhead of global cooperation is to be avoided. Global cooperation would allow all processors to work on instances of the same task if needed. We assume uniformity and close coupling of processors, so global cooperation is chosen here.

The following theorem states that selection of task instances would not be an issue if no time is wasted executing extra (in excess of $c$) task instances.

**Theorem 2** *Any scheduling scheme that does not force processors to be idle or to execute extra task instances has the same probability of a successful run.*

$$Pr = \sum_{k=cm}^{nd} \binom{nd}{k} p^k (1-p)^{nd-k}$$

**Proof** First we describe a way to compute the probability of successful runs for any scheduling scheme. We have assumed that each task instance has an identical probability of being correct, and that each instance is independent. The scheduling activity can thus be modeled by a program $S$, an input vector $X = (X_1, X_2, \ldots, X_{nd})$, and an output variable $Y$. $X_i$ is boolean; $X_i = 1$ indicates the instance run on a particular processor at a particular time step is correct. $Y$ is also boolean for deterministic schemes where $Y = 1$ indicates that the run is successful. For random schemes, $Y$ is the conditional probability that given the input vector $X$, the program would lead to a successful run. In either case, a scheduler $S$ can be viewed as a *function* from input $X$ to output $Y$.

Each input vector $X$ has a probability $\Pr(X)$ associated with it. Specifically, if $X$ contains $k$ ones, the probability is $\binom{nd}{k} p^k (1-p)^{nd-k}$. The overall probability of success of the scheduler is then the sum of $\Pr(X)Y(X)$ over all possible $X$'s.

Now we consider two programs. The first program is a general one with $m$ tasks, with $c$ instances of each required, $n$ processors and $d$ units of time, and is written as $S_{(m,c,n,d)}$. The second program $S_{(1,cm,1,nd)}$ is for a single-processor, single-task scheme with equivalent load. All the scheduling schemes that do not waste time can differ only in the assignment of tasks and processors, and thus behave the same when applied to the single-task single-processor case. In other words, $S_{(1,cm,1,nd)}$ is the same for all schedulers that do not waste time.

We assert that, if the scheduler does not waste time, both programs $S_{(m,c,n,d)}$ and $S_{(1,cm,1,nd)}$ are the same function from $X$ to $Y$. This is because any input vector $X$ such that $S_{(m,c,n,d)}(X) = 1$ must have at least $cm$ 1's, and so $S_{(1,cm,1,nd)}(X) = 1$. And, any input vector $X$ such that $S_{(m,c,n,d)}(X) = 0$ must have less than $cm$ 1's (no time is wasted in doing tasks that already have $c$ correct completions) , and so $S_{(1,cm,1,nd)}(X) = 0$.

Since any scheduler that doesn't waste time represents the same function from $X$ to $Y$ as $S_{(1,cm,1,nd)}$, the probability of a successful run is the same as well. Clearly, the probability of success for $S_{(1,cm,1,nd)}$ is as shown in the statement of the theorem. □

Theorem 2 seems to simplify the task of designing scheduling algorithms, since some aspects of scheduling are irrelevant to the performance. However, the condition that no time be wasted is realistic only with one or two processors or when $m = 1$. Otherwise, there is some chance that time will be wasted.

For example, let $n = 3$ and $m = 2$, and assume that at one point there is one remaining correct instance to be completed for each of the 2 tasks. If we schedule

2 instances of task 1 and 1 instance of task 2, we are at the risk of getting 2 additional correct instances of task 1 and none of task 2, and thus one time step that could have been used to run task 2 is wasted on task 1. Scheduling 1 instances of task 1 and 2 instance of task 2 runs a similar risk. The success probability deteriorates in either case.

Theorem 2 also suggests that on-demand schemes are superior to projective schemes because they waste less time. Time is wasted only when the demand is less than the available processing power, $n$, and that usually happens near the end of computation. Projective schemes, on the other hand, might waste at most $\max_x(g(x) - x)$ units of time for each task. For on-demand schemes, the upper bound for wasted time is reduced if the *surplus*, defined as $n$ minus the total demand in any time step, is evenly distributed among tasks not having enough correct completions. Theorem 3 gives the upper bound, and Theorem 4 provides a lower bound for the success probability, given an upper bound for the wasted time.

**Theorem 3** *For on-demand scheduling, if the surplus is evenly distributed among tasks that have not had enough correct completions, the worst case waste is*

$$w_{MAX} = \sum_{k=2}^{l} \lceil n/k \rceil - l + 1, \ \textit{where } l = \max(n-1, m).$$

**Proof** The above expression is obtained by observing that the $k$th tardiest task to finish can waste no more than $\lceil (n-k)/k \rceil = \lceil n/k \rceil - 1$ units of time. □

**Theorem 4** *Suppose a scheduler has an* upper bound *$w$ for the total wasted time among all processors. Then the probability of successful runs is*

$$Pr \geq \sum_{k=cm+w}^{nd} \binom{nd}{k} p^k (1-p)^{nd-k}.$$

**Proof** Similar to the proof of Theorem 2, we construct a program $S$ from a no-waste scheduler and another program $S'$ from a scheduler that wastes at most $w$ units of time. Any (random) input vector $X$ that has at least $cm$ ones would produce $Y = 1$ on $S$, but may or may not produce $Y = 1$ on $S'$. However, an $X$ that has at least $cm + w$ ones must produce $Y = 1$ on $S'$, because in the worst case, after ignoring $w$ ones, there are still at least $cm$ ones left for the useful executions. The probability of successful runs is thus at least the probability of getting $cm+w$ heads in $nd$ coin-flips, with $p$ being the probability of head in each flip. □

Table 1: Worst-case wasted time for on-demand scheduling ($m \geq n-1$ assumed)

| $n$ | $w_{MAX}$ |
|---|---|
| 2 | 0 |
| 3 | 1 |
| 4 | 2 |
| 5 | 4 |
| 6 | 5 |
| 7 | 8 |
| 8 | 9 |
| 9 | 13 |
| 10 | 14 |

Table 2: An example for the worst-case wasted time

| | → advancing time | | |
|---|---|---|---|
| P1 | 1 | 1 | 1 |
| P2 | 2 | 2 | 1 |
| processor P3 | 3 | 2 | 2* |
| P4 | 4* | 3* | 2* |
| P5 | 4* | 3* | 2* |
| wasted time | 1 | 1 | 2 |

Table 1 shows the worst-case wasted time for some small values of $n$, assuming $m \geq n-1$. To show that the bound is tight, we present an example. Assume at one point of time there are 5 processors named P1 through P5, and 4 tasks numbered 1 through 4, and each task requires 1 additional correct completion. Table 2 shows a schedule and the correctness of task instances that causes the worst-case wasted time. Instances marked by asterisk (*) are the correct ones, while the unmarked ones are incorrect. The total wasted time is 4, as suggested by Table 1.

Theorems 3 and 4 together provide a lower bound on success probabilities. The distribution of wasted time and its exact effect on the success probability depends on the scheduling algorithm and is thus very difficult to analyze. For large $m$ and $n$, the worst-case wasted time predicted by Theorem 3 is large, and might give too pessimistic a measure of performance. In such cases, simulation may be used to obtain a closer estimate.

## 2.3 Example

In this example we consider $m = 5$ tasks, each requiring $c = 3$ correct instances, and result correctness probability of $p = 0.8$. The number of processors $n$ and the deadline $d$ are chosen to first form a constant product of 30, and then $d$ is fixed at 10 while $n$ varies. For the preset schedules, the probability of success is computed as $\prod f(b_i)$, as defined in Section 2.1. For

Table 3: Prob. of success for a homogeneous example

| $n$ | $d$ | Preset | Adaptive lower bound | Adaptive depth-first | Adaptive brdth-first | Preset with cancell. |
|---|---|---|---|---|---|---|
| 1 | 30 | 0.91803 | 0.99995 | 0.99995 | 0.99995 | 0.99995 |
| 2 | 15 | 0.91803 | 0.99995 | 0.99996 | 0.99996 | 0.99939 |
| 3 | 10 | 0.91803 | 0.99977 | 0.99987 | 0.99990 | 0.99438 |
| 5 | 6 | 0.91803 | 0.99051 | 0.99981 | 0.99978 | 0.99464 |
| 6 | 5 | 0.91803 | 0.97438 | 0.99974 | 0.99975 | 0.91867 |
| 10 | 3 | 0.91803 | 0.42751 | 0.99848 | 0.99733 | 0.96723 |
| 2 | 10 | 0.36893 | 0.80421 | 0.80450 | 0.80450 | 0.36961 |
| 3 | 10 | 0.91803 | 0.99977 | 0.99987 | 0.99990 | 0.99398 |
| 4 | 10 | 0.99386 | 1.00000 | 1.00000 | 1.00000 | 1.00000 |
| 5 | 10 | 0.99961 | 1.00000 | 1.00000 | 1.00000 | 1.00000 |
| 6 | 10 | 0.99997 | 1.00000 | 1.00000 | 1.00000 | 1.00000 |

adaptive scheduling, Theorems 3 and 4 are used to find a lower bound on the success probability, and two scheduling schemes are evaluated with simulation programs to obtain the respective probabilities. Table 3 shows the results.

The depth-first on-demand scheme imposes a fixed priority on tasks and in each unit of time schedules $c-b_1$ instances of task 1, $c-b_2$ instances of task 2, and so on, until the available processors are exhausted[1]. In case there are surplus processors, they are evenly distributed among the instances of all tasks. The breadth-first on-demand scheme chooses the tasks in a circular order while skipping the tasks that already have enough correctly completed instances.

Optimal preset schedules can be made adaptive by a simple cancellation rule. We call such a scheme *preset with cancellation*. A task instance on the schedule is cancelled if enough correct instances of the task have already been observed, and the next item on the schedule is executed. In case the preset schedule runs out, the remaining processing power is evenly distributed among instances of tasks that have not had enough correct completions.

As predicted, adaptive schemes have higher probability of success than the preset scheme. Performance of the preset with cancellation scheme is between the preset and adaptive schemes. As for the two adaptive schemes, they appear to be very close; simulation programs give only estimates of the probabilities. Although the worst case wasted time is the same for both, intuitively the depth-first scheme should waste less time, on the average, because there are usually fewer unfinished tasks when the processors begin to have surplus time steps.

[1] $b_i$ is the number of instances of task $i$ that have already run to correct completion.

# 3 The Common-Deadline Model

## 3.1 Preset Scheduling

In the common deadline model, the computation time for each task can be different, with task $i$ having running time $r_i$. Thus uniform distribution of scheduled task instances is not always optimal. Intuitively, replacing a long instance with several short instances might increase the overall chance of success.

The scheduling problem can be formulated as an integer programming problem with linear constraints and a nonlinear value function. Let $b_{ij}$ be the number of instances of task $i$ executed on processor $j$. The problem is:

$$\begin{aligned} \text{Maximize } v &= \prod_{i=1}^{m} f\left(\sum_{j=1}^{n} b_{ij}\right) \quad \text{subject to} \\ b_{ij} &\geq 0, \text{ for all } 1 \leq i \leq m, 1 \leq j \leq n, \\ \sum_{i=1}^{m} b_{ij} r_i &\leq d, \text{for all } 1 \leq j \leq n. \end{aligned}$$

This optimization problem is very difficult even for modest numbers of tasks and processors. As the context is preset scheduling, sufficient resources should be invested to obtain the optimal solution.

The following approximation algorithm can be used if optimal solution is too expensive to obtain or if optimality is not required.

The algorithm is described in terms of bin-packing. Sizes of objects correspond to task computation times, while the capacity of each bin is $d$. We have an unknown number of objects being packed into $n$ bins.

In the first stage, we pack $cm$ objects, which represent $c$ instances of each task, into a number of bins. The objective is to pack as few bins as possible and to leave as little room on the partially packed bins as possible.

Next we pack more objects into the bins, starting from the object that gains most on the value function per unit space used, and pack it into the smallest available space. As the free space on the bins dwindles, the best task might not fit into any bin. In such cases, the next best task is tried, until free space on every bin is smaller than the smallest object.

We take the logarithm of success probability as our value function.

$$\log(f(b_1)f(b_2)\cdots f(b_m)) = \log f(b_1) + \log f(b_2) + \ldots + \log(b_m).$$

Table 4: Logarithmic gain $L(b)$ ($c = 3, p = 0.8$)

| $b$ | $L(b)$ |
|---|---|
| 3 | 0.4700036 |
| 4 | 0.1397619 |
| 5 | 0.0425596 |
| 6 | 0.0124225 |
| 7 | 0.0034508 |
| 8 | 0.0009182 |
| 9 | 0.0002360 |
| 10 | 0.0000590 |

The per-unit-time gain by scheduling one more instance of task $i$ is thus

$$\frac{log(f(b_i+1)/f(b_i))}{r_i},$$

where $b_i$ is the number of instances of task $i$ already scheduled. Logarithm is used so that the gains in successive actions can be added to find the final gain. In addition, dividing the logarithmic gain over a size, $\log(f(b+1)/f(b))/r$, gives a correct measurement of the per-unit-time gain. We define $L(b)$ to be

$$\begin{aligned} L(b) &= \log(f(b+1)/f(b)) \\ &= \log(1+\ \Delta f(b)/f(b)), \text{ for } b \geq c. \end{aligned}$$

For $b < c$, the value of $L(b)$ is conveniently defined to be infinity to reflect the urgency to schedule at least $c$ instances of each task. Table 4 shows several values of $L(b)$ for $c = 3$ and $p = 0.8$. Since $L(b)$ needs to be used many times in the bin-packing, we can precompute values of $L(b)$ and store them in a table. Finding the per-unit-time gain can thus be accomplished by a table-lookup step followed by a division.

## 3.2 Adaptive Scheduling

Adaptive scheduling for the common-deadline model is more difficult than adaptive scheduling in the homogeneous case because the computation time can be different for each task. First, we explain why the heuristics for the homogeneous case do not work as well in this model. Then, we discuss a unit-time-gain scheme for this model.

The on-demand approach aims to schedule no more than the necessary number of instances of every task. With the common-deadline model, the computation times are different, and so a uniform distribution of the number of instances is not always optimal. Furthermore, neither the depth-first on-demand nor the breadth-first on-demand schemes takes the computation time into account.

We suggest a unit-time-gain heuristic for the adaptive scheduling of tasks in the common-deadline model. The same function $L(b)$ as in the previous subsection is used for the logarithmic gain. Again we might want to precompute and store $L(b)$ to obtain the unit-time-gain quickly. The value of $L(b)$ for $b < c$, defined to be infinity, can assume a suitably large value.

The scheduler selects tasks with the highest unit-time-gain, i.e., $\max_i L(b_i)/r_i$, with $b_i$ being the number of instances scheduled for task $i$. Note that $b_i$ is a most optimistic projection of the number of correct completions. Every extra instance scheduled increases $b_i$ by exactly one. This has the effect of regulating the number of processors running in parallel for any task according to the value function.

The unit-time-gain heuristic suffers from fragmentation when the deadline is tight. There might not be enough time to do the tasks having the highest unit-time-gain if they happen to be longer ones and the deadline is close. In these cases, using the preset schedules with cancellation or even without cancellation, might yield a higher success probability.

## 3.3 Example

In this example we consider $m = 3$ tasks, each requiring $c = 3$ correct instances, and the result correctness probability of $p = 0.8$. The execution times are $r_1 = 1$, $r_2 = 2$, and $r_3 = 2$. The same set of values for the number of processors and deadlines as in Section 2.3 is used.

The optimal assignment of number of instances for the single-processor case is found to be $b_1 = b_2 = b_3 = 6$, i.e., a uniform assignment. In general, especially when the computation times vary over a wider range, the optimal assignment might not be uniform. This uniform assignment fits into schedule for the next 4 cases. Consequently the success probabilities for the first 5 cases are the same. For $n = 10$ and $T = 3$, although the processor-time product remains the same, the uniform assignment would not fit. The best schedule has $b_1 = 10, b_2 = b_3 = 5$, and has lower probability than the first 5 cases. In the constant deadline part, $b_1 = b_2 = b_3 = 2n$ is optimal.

Unit-time-gain heuristic is used for the adaptive scheduling. In the constant processor-deadline product part, this scheme is better than the preset scheme for $n = 1, 2, 3, 5$ and 6. For $n = 10$, the preset scheme is better. In the constant deadline part of the table, adaptive scheme is consistently better, and the success probability approaches one much faster than for the preset scheme.

Table 5: Probability of success for a common-deadline example

| $n$ | $d$ | Preset | Adaptive | Preset with cancellation |
|---|---|---|---|---|
| 1 | 30 | 0.94998 | 0.99788 | 0.99823 |
| 2 | 15 | 0.94998 | 0.99710 | 0.99124 |
| 3 | 10 | 0.94998 | 0.99052 | 0.97937 |
| 5 | 6 | 0.94998 | 0.97821 | 0.98970 |
| 6 | 5 | 0.94998 | 0.95911 | 0.94959 |
| 10 | 3 | 0.88744 | 0.70802 | 0.88701 |
| 2 | 10 | 0.54976 | 0.71902 | 0.55228 |
| 3 | 10 | 0.94998 | 0.99052 | 0.97937 |
| 4 | 10 | 0.99631 | 0.99993 | 0.99947 |
| 5 | 10 | 0.99977 | 1.00000 | 1.00000 |
| 6 | 10 | 0.99999 | 1.00000 | 1.00000 |

We also have the preset with cancellation scheme that takes the optimal preset schedule for each case and adaptively replaces unnecessary task instances with useful ones. The order in which task instances are scheduled does not make any difference in the preset scheme, but with cancellation it matters; a larger number of different tasks being run tends to waste fewer time steps and thus leads to higher probability of success. Finding an optimal preset schedule to be run adaptively with cancellation is harder than finding an optimal preset schedule. In this example, we avoid the problem by considering a fixed order for the tasks, longest task first. The success probability we obtain is not very predictable; in some cases it is between that of the preset and adaptive schemes, while in others it is the best among the three.

# 4 The General Deterministic Model

## 4.1 Preset Scheduling

In our most general deterministic model, each task has its own release time $s_i$, computation time $r_i$, and deadline $d_i$. Execution of tasks can be *preemptive* or *nonpreemptive*. Nonpreemptive scheduling is more difficult so we assume preemptive scheduling.

This is an even harder problem than the optimization problem for the common-deadline mode. Algorithms exist for preemptive scheduling of tasks with arbitrary release times, computation times, and deadlines (see, e.g., [4]). Using such an algorithm as a filter, we may screen all possible combinations of $b_1, b_2, \ldots, b_m$, then pick among the schedulable ones the one with greatest value of $\prod f(b_i)$. One variation is to prescreen the sets of $b_i$ values using simple feasibility tests such as

$$\sum b_i r_i \leq n(\max d_i - \min s_i), \text{ and}$$

$$c \leq b_i \leq \left\lceil \frac{n(d_i - s_i)}{r_i} \right\rceil.$$

The search space can still be huge for a modest number of tasks. If resources are not available to find the optimal solution, approximation algorithms may have to be used.

One possible strategy is as follows. First we compute an optimal assignment of number of task instances with the least restrictive timing constraints, i.e., $n' = 1$ and $d' = nd$, where $d = \max d_i - \min s_i$. Then, we try to fit this assignment into the originally imposed release times and deadlines, using a traditional scheduling algorithm. If the assignment does not fit, we try to accommodate as many instances as possible, by successively cutting down the number of instances for some tasks. If an assignment fits, we then try to accommodate more (possibly smaller) task instances. This method should give a reasonably good solution if timing constraints are not tight. In evaluating the simple example in Section 4.3, this method is used.

## 4.2 Adaptive Scheduling

Many possible adaptive schemes can be derived from scheduling schemes found in the context of real-time computing, such as earliest deadline and minimum slack. Also, preset schedules may be used adaptively with cancellation.

As the preset scheduling problem for the general deterministic model is very difficult, one might expect the preset with cancellation scheme to perform better than a strictly run-time scheduling scheme. However, the generality of timing constraints and the nonlinearity of the objective function makes it is very difficult to find a scheme that is ***universally*** better than others, or even one that is better than others *most of the times*; some distribution functions for the timing parameters will have to be specified first.

In this study we experiment with an *earliest-deadline projective* scheme. The scheme orders the tasks in ascending order of their deadlines, and tries to keep $\lceil (c - b_i)/p \rceil$ instances of task $i$ running, where $b_i$ is the number of completed instances known to be correct. Processor time is allocated according to the deadline order. The number $\lceil (c - b_i)/p \rceil$ is a projection of how many extra instances are required to reach the goal of $c$ correct completions. This linear projection function may not appear to be aggressive enough,

but the idea is to give the task with the earliest deadline enough processing power without jeopardizing the other tasks or wasting too much resources on unnecessary task instances.

## 4.3 Example

In this example we consider $m = 3$ tasks, each requiring $c = 3$ correct instances, and the same result correctness probability of $p = 0.8$ as in the previous examples. To compare with the less restrictive common-deadline model, a comparable set of number of processors and timing parameters are used. The release time, computation time, and deadline for task $i$ are denoted as $s_i$, $r_i$, $d_i$, respectively. Computation time for task 1, $r_1$, is always 1. Computation times for tasks 2 and 3 are always 2. Task 1 is assumed to have a release time of 0, and deadline $d_1$ about one third of the total time $d$. Task 2 starts at the deadline for task 1, $s_2 = d_1$, and has deadline at the end of total time, $d_2 = d$. Task 3 can use the total available time; $s_3 = 0$ and $d_3 = d$. The product $nd$ is again kept constant for the first part of the example, and then $d$ is fixed at 10 while $n$ varies in the second part. The results are shown in Table 6.

For the preset schedules, the optimal assignment of number of task instances for $n = 1, d = 30$ is $b_1 = b_2 = b_3 = 6$. Similar to Section 3.3, The first 5 cases can fit this assignment into schedule. For $n = 10$, again $b_1 = 10, b_2 = b_3 = 5$ is the best assignment. For the second part, the optimal assignment is $b_1 = b_2 = b_3 = 2n$, and fits into the schedule in all 5 cases, as in Section 3.3. Results in the first column are thus identical with those in the first column of Table 5.

Results for the earliest-deadline projective and preset with cancellation schemes are shown as well. With the extra timing constraints, the preset with cancellation scheme does not perform as well as the adaptive scheme. For all cases except for $n = 10, d = 3$, the adaptive scheme has the highest success probability, the preset with cancellation scheme has the next highest, followed by the preset scheme. For $n = 10, d = 3$, the adaptive scheme is the worst, and the preset with cancellation scheme should have the same probability of success as the preset scheme, since no cancellation can occur in this case.

We note that the adaptive scheme is better than both preset and preset with cancellation in all but the tightest deadline case. The poorer performance of the adaptive scheme in the tightest deadline case may be due to the stingy projection function, but stinginess may contribute to the good overall performance. This

Table 6: Probability of success for a general deterministic example

| $n$ | $d$ | Preset | Adaptive | Preset with cancellation |
|---|---|---|---|---|
| 1 | 30 | 0.94998 | 0.99788 | 0.97621 |
| 2 | 15 | 0.94998 | 0.99069 | 0.98356 |
| 3 | 10 | 0.94998 | 0.98934 | 0.97823 |
| 5 | 6 | 0.94998 | 0.97272 | 0.97125 |
| 6 | 5 | 0.94998 | 0.97623 | 0.95140 |
| 10 | 3 | 0.88744 | 0.88050 | 0.88705 |
| 2 | 10 | 0.54976 | 0.55068 | 0.54991 |
| 3 | 10 | 0.94998 | 0.98934 | 0.97823 |
| 4 | 10 | 0.99631 | 0.99990 | 0.99865 |
| 5 | 10 | 0.99977 | 1.00000 | 0.99994 |
| 6 | 10 | 0.99999 | 1.00000 | 1.00000 |

is an example of how a simple heuristic would do well in some of the cases and not so well in others.

# 5 Conclusions

A framework has been constructed to investigate scheduling multiple instances of tasks in a coarse-grained multiprocessor environment. Three successively less restrictive timing models, the homogeneous model, the common-deadline model, and the general deterministic model were dealt with. For each of the three models, we considered preset schedules, adaptive schedules, and adaptive use of preset schedules.

For the homogeneous model, theoretical results were presented that allow the construction of optimal preset schedules, and aid in the design of heuristics for adaptive scheduling. Optimal schedule was shown for preset scheduling and two heuristics (depth-first on-demand and breadth-first on-demand) were proposed for adaptive scheduling. For both the common-deadline and the general deterministic model, preset scheduling is a hard problem. Algorithms were presented to find approximate solutions to the problems. For adaptive scheduling, a unit-time-gain heuristic for common-deadline model, and an earliest-deadline projective heuristic for general deterministic model were provided.

We conclude our study by presenting an example where neither the earliest-deadline-first nor the least-laxity-first algorithm (both of which are quite effective in other contexts) yields an optimal adaptive scheme. Let $c = 2$ and $n = 4$, $r_1 = r_3 = 2$, $r_2 = 4$, $d_1 = 6$, $d_2 = 7$, $d_3 = 8$. Both of the above algorithms yield optimal preset schedules as shown in Figures 1 (a) and (b) ($T_{i,j}$ represents instance $j$ of task $i$). If cancellations are allowed, neither of the above schedules can make use of

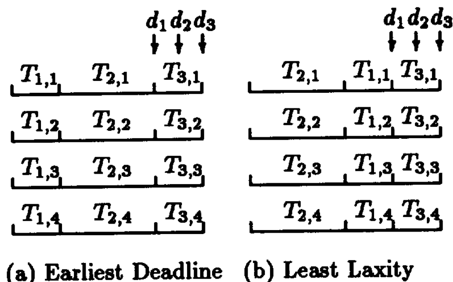


Figure 1: An example in which optimal preset schedules are not optimal preset-with-cancellation schedules

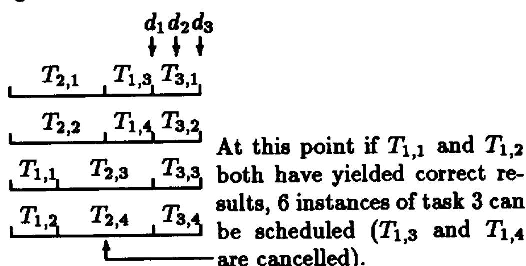


Figure 2: A better preset-with-cancellation schedule

the resultant flexibility. However, the schedule shown in Figure 2 can achieve a higher success probability.

This study can be extended in a number of directions. Since the two heuristics for adaptive scheduling in the homogeneous model appear to have very good performance, it might be possible to establish their optimality. In the common-deadline and general deterministic model, there is a tradeoff between on-demand and projective scheduling, and thus a more complex heuristic that adaptively adjusts the projection function may be worth investigating. The deterministic running times of tasks can be changed to randomly distributed ones. Tasks with precedence constraints can be modeled by making the release time of a dependent task later than the deadline of the prerequisite task. However, this approach adds to timing constraints and is likely to decrease the probability of meeting all deadlines. It is more desirable to have a model that allows the precedence constraints to begin with. Finally, one may consider *soft* deadlines by introducing some penalty or cost function for missing a deadline.

## References

[1] J.-Y. Chung, J. W. S. Liu, and K.-J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Trans. Computers*, 39(9):1156–1174, Sep. 1990.

[2] A. L. Liestman and R. H. Campbell. A fault-tolerant scheduling problem. *IEEE Trans. Software Engineering*, 12(11):1089–1095, Nov. 1986.

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

[4] C. Martel. Preemptive scheduling with release times, deadlines, and due dates. *Journal of the ACM*, 29(3):812–829, July 1982.

[5] J. K. Muppala, S. P. Woolet, and K. S. Trivedi. Real-time systems performance in the presence of failures. *Computer*, 24(5):37–47, May 1991.

[6] B. Parhami. A unified approach to correctness and timeliness requirements for ultrareliable concurrent systems. In *Proc. Int'l Parallel Processing Symp.*, 733–747. Fullerton, CA, Apr. 1990.

[7] D.-T. Peng. Performance bounds in list scheduling of redundant tasks on multi-processors. In *Proc. Int'l Symp. Fault-Tolerant Computing*, 196–203, Boston, July 1992.

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

[9] K. G. Shin, T.-H. Lin, and Y.-H. Lee. Optimal checkpointing of real-time tasks. *IEEE Trans. Computers*, 36(11):1328–1341, Nov. 1987.

[10] J. A. Stankovic and K. Ramamritham (Editors). *Tutorial: Hard Real-Time Systems*. IEEE Computer Society Press, 1988.

[11] J. D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, June 1975.

[12] J. Xu and D. L. Parnas. On satisfying timing constraints in hard-real-time systems. In *Proc. ACM SIGSOFT, Software for Critical Systems*, New Orlean, Dec. 1991.