Fault-Tolerant Scheduling in Homogeneous Real-Time Systems

C. M. KRISHNA, University of Massachusetts at Amherst

Real-time systems are one of the most important applications of computers, both in commercial terms and in terms of social impact. Increasingly, real-time computers are used to control life-critical applications and need to meet stringent reliability conditions. Since the reliability of a real-time system is related to the probability of meeting its hard deadlines, these reliability requirements translate to the need to meet critical task deadlines with a very high probability. We survey the problem of how to schedule tasks in such a way that deadlines continue to be met despite processor (permanent or transient) or software failure.

Categories and Subject Descriptors: A.1 [**Introductory and Survey**]; D.4.1 [**Process Management**]: Scheduling; D4.5 [**Reliability**]: Fault Tolerance; D4.7 [**Organization and Design**]: Real-Time Systems; J.7 [**Computers in Other Systems**]: Real-Time Systems

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Cyber-physical systems, task assignment, task scheduling, real-time systems, time redundancy

ACM Reference Format:
C. M. Krishna. 2104. Fault-tolerant scheduling in homogeneous real-time systems. ACM Comput. Surv. 46, 4, Article 48 (March 2014), 34 pages.
DOI: http://dx.doi.org/10.1145/2534028

1. INTRODUCTION

Real-time systems are an important application of computer systems. Increasingly, they are used in life-critical and complex applications, in the control of aircraft, medical equipment, automobiles, chemical plants, and power distribution systems.

The hallmark of real-time systems is that their tasks have deadlines, and missing too many such deadlines in a row can result in catastrophic failure (such as an air crash or the death of a patient). As a result, much effort has been devoted in recent years to developing techniques by which to render such systems highly reliable. These efforts have generally involved the use of massive hardware redundancy—that is, of using many more processors than are absolutely necessary to ensure that enough will remain alive, despite failures, to continue providing acceptable levels of service.

However, throwing a lot of redundant hardware at the problem is not always possible. Increasingly, real-time systems are used in cost-sensitive applications like cars, where a cost differential of even a hundred dollars can make a commercial difference. In addition, there are constraints other than cost that must be taken into account, the

This work was supported in part by the National Science Foundation under grant CNS-0931035.
Preparation of this work was supported in part by the National Science Foundation under grant CNS-0931035.
Author's address: C. M. Krishna, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003; email: krishna@ecs.umass.edu.

DOI: http://dx.doi.org/10.1145/2534028

Table I. Guide to Coverage of Survey

Section	Failure Type	Task Type	Offline/Online	Scheduling Algorithm
4.1	Transient	Periodic	Offline	Rate monotonic
4.2	Transient	Aperiodic	Online	Earliest deadline first
5.1	Software Only	Periodic	Offline	Any priority scheme
5.2	Software Only	Arbitrary	Online	Any priority scheme
6.1.1	Permanent	Periods Equal	Offline	Longest exec time first
6.1.2	Permanent	Periodic	Offline	Rate monotonic
6.2	Permanent	Aperiodic; Nonpreemptive	Online	Any scheme
6.3	Permanent	Aperiodic with Cost Functions	Offline	Any scheme

most obvious of which is the availability of power: many applications have quite notable constraints on their total power or energy budget.

Further, the vast majority of processor faults are transient in nature, indicating that what is primarily needed is a mechanism to tide the system over temporary processor outages.

For all of these reasons, researchers have considered techniques that can complement massive redundancy and to make such redundancy as is available count effectively toward improving reliability. This field is now mature enough that a survey of the more important techniques is worth carrying out.

This article is organized as follows. In Section 2, we provide some technical background on real-time scheduling, processor failure models, the fault-tolerant structures that are used, and the use of voltage scaling to provide processors with an extra reserve of speed. Section 3 covers the foundations of fault-tolerant scheduling. Dealing with transient failures is covered in Section 4. Handling software failures[1] is very similar, as we see in Section 5. Only single processors need to be considered in both Sections 4 and 5; even if the system has multiple processors, the reaction to the failure involves the affected processor only. After this, we turn in Section 6 to handling permanent hardware faults, which obviously requires a multiprocessor system, and then, in Section 7, to the use of voltage scaling. A discussion of some open problems follows in Section 8. A table of notation is provided in the Appendix.

To provide a convenient guide to the rest of the article, Table I lists the failure assumptions and scheduling algorithms included in this survey.

2. TECHNICAL BACKGROUND

In this section, we outline some of the more salient issues in real-time task scheduling of relevance to our problem. The reader should consult one of the many texts on real-time systems for further information [Cheng 2002; Krishna and Shin 1997; Liu 2000; Murthy and Manimaran 2001; Nissanke 1997].

2.1. Types of Real-Time Systems

We start by considering the two categories into which real-time systems can be classified: *hard* and *soft*. Hard real-time systems are those whose failure (triggered by missing too many hard deadlines) leads to catastrophe. For example, if the computer on a fly-by-wire aircraft fails completely, the aircraft crashes. If a robot carrying out remotely commanded surgery misses a deadline to stop cutting in a particular direction, or a cancer treatment irradiation machine delivers too high a dose of radiation (by not switching off the beam at the right time), the patient can die.

[1]For our purposes, timing overruns are also considered software failures.

In a soft real-time system on the other hand, missing any number of deadlines may be a cause of user annoyance; however, the outcome is not catastrophic. A server telling you the latest score in a cricket match may cause some users great distress by freezing at the most exciting point of the match, but that is not a catastrophe. An airline reservation system may take a very long time to respond, driving its more impatient customers away, but that may still not be classified as a catastrophe (at least in the short term).

Thus, the classification of real-time systems into hard and soft depends on the application, not on the computer itself. The same type of computer, running similar software, may be classified in one context as soft but as hard in another. For example, consider a video or audio-only conferencing application. When used for routine personal or business communications, it can be considered a soft real-time system: if the system has outages, it is annoying, but the conversations can be rescheduled. If, on the other hand, it is used by police officers and firefighters to coordinate actions at the scene of a major fire, it is a hard real-time system. Or, take a real-time database system. When used to provide sports scores (as we have seen), it is soft; however, if it is used to provide stock market data, an outage can cause significant losses and may be regarded by its users as catastrophic. In this case, it would be considered a hard real-time system.

Perhaps a good rule of thumb is to say that the designers of hard real-time systems expend a significant fraction of the development and test time ensuring that task deadlines are met to a very high probability; by contrast, soft real-time systems are really of the "best effort" variety, in which the system makes an effort (sometimes a considerable effort) to meet deadlines, but no more than that.

Note that the same computer system may run both hard and soft tasks. In other words, some of its workload may be critical and some may not be. Further, this may change with time as the system operates; the same task may be hard-real-time if the application is in one part of its operating state-space and soft-real-time in the rest.

Most of the applications for which one requires fault-tolerant scheduling require hard real-time computers. Such systems run two types of tasks: periodic and sporadic. As the term implies, a periodic task, T_i, is issued once every period of P_i seconds. Typically (but not always), the deadline of a periodic task is equal to its period: many of the results in real-time scheduling are based on this assumption. A sporadic task, on the other hand, can be released at any time; however, specifications may limit their rate of arrival to no more than one every τ seconds.

The majority of real-time systems are extremely simple, many controlled by primitive eight-bit microcontrollers. Fault-tolerance features in many such applications are rudimentary at best.

In many important applications, however, the real-time system carries a heavy workload and is in the control of significant physical systems. Such systems have been used in aerospace applications for decades; more recently, they have begun to proliferate in other hard real-time applications as well. These include cars, industrial robots, chemical plants, and mechanisms for remote surgery. The fault-tolerant scheduling approaches that we survey here are meant for such application areas.

We concentrate in this survey on *homogeneous* real-time systems. That is, we do not consider what happens if the processors in the system are of differing types, each with a different affinity to certain instruction mixes and different failure rates. We briefly mention issues related to removing this restriction in Section 8.

2.2. Task Scheduling

All real-time task scheduling theory assumes that the Worst-Case Execution Time (WCET) of tasks is known in advance. This is not a trivial assumption. Modern processors have out-of-order and speculative execution as well as multiple levels of cache: all

of these can render the problem of determining the WCET of a piece of software very difficult. Obtaining the WCET is, at present, a rather active area in real-time systems[2] but is outside the scope of this article.

We can broadly classify task scheduling into two categories: *offline* and *online*. An offline schedule is one in which the schedule is generated ahead of time and is subsequently followed by the system as it operates. Generally, most tasks run by such systems are periodic, with perhaps a small (but not unimportant) component of the workload being sporadic, with their arrival times not known in advance. We limit the load that sporadic tasks impose on the system by specifying a minimum time between two successive arrivals of a sporadic task. Space is reserved in the schedule to allow for sporadic tasks to be executed on time. Note that in an offline schedule, the task loading is bounded: barring failures, a successfully generated schedule guarantees that all tasks will meet their deadlines.

An online schedule, on the other hand, is generated on-the-fly as tasks arrive, and does not require us to know in advance the task loading bounds on the system. The price we pay for this is that we cannot guarantee in advance that all task deadlines will be met. Instead, when each task arrives at the system, the system determines whether there is enough time to execute it as well as the other tasks currently awaiting execution. If there is, the task is *guaranteed*. If there is not, the task is rejected, and it is then up to the user (or the operating system) to invoke some backup activity.[3] For example, if the computer in an intensive care unit is unable to meet all of the real-time requirements for monitoring a given patient and injecting suitable drugs into an IV line, a nurse may be designated to take over some of its functions [Ghosh et al. 1997].

Tasks may either be independent of one another or have precedence constraints. Tasks with precedence constraints can only be started when their preceding tasks have been finished (assuming that a task needs all of its inputs before it can start execution). The vast majority of fault-tolerant scheduling results apply for independent tasks; unless otherwise stated, tasks will be assumed to be independent.

Scheduling algorithms may be either *preemptive* or *nonpreemptive*. A preemptive scheduler allows tasks to be interrupted partway through their execution and then resume from where they left off. In some cases, this may not be possible; then, nonpreemptive rules apply. In general, nonpreemptive scheduling is more difficult than preemptive scheduling. In a nonpreemptive regime, when a task starts executing, we are reserving the processor for that task until it finishes, even if other much more urgent tasks arrive. Scheduling decisions for nonpreemptive task sets therefore can have anomalous results—for example, reducing the execution time of a task can result in actually lengthening the total time taken to complete a set of tasks [Graham 1969].

2.3. Faults

All computer systems are subject to hardware and software faults. Hardware faults can be classified as *permanent* or *transient* [Johnson 1989; Koren and Krishna 2007; Pradhan 1996; Sieworek and Swarz 1999]. Permanent means faults that do not go away with time: they include (for the purposes of this article) *intermittent* faults, where the fault cycles between being active and being passive. Transient faults, by contrast, go away after some time. A common form of transient fault is the inducing in memory cells

[2]As a matter of practice, many designers simply gather experimental data about the runtime of a task over a large number of runs and multiply the maximum observed runtime by some safety factor to produce an estimate of the WCET. Such estimates are not provably correct, however. There is no guarantee that even a very large number of experimental runs will catch the worst-case combination of inputs.

[3]In most of the work in this area, the guarantee or rejection takes place almost instantly; however, allowing the decision to be deferred until some previously guaranteed tasks have successfully completed execution may improve the fraction of incoming tasks that can be guaranteed [Naedele 1999].

of spurious values, caused by charged particles (e.g., alpha particles) passing through them. The charge carried by such particles may induce the memory cell to flip its value. However, no permanent damage is done to the cell: the next time it is written into, the incorrect memory state will be wiped out. The vast majority of faults are transient; their rate depends on the VLSI technology being used and the operating environment of the system.

In general, the smaller the feature size, the more susceptible a memory cell is to changing its state in response to interactions with charged particles. As feature sizes drop ever further into the deep submicron range (32nm is quite a mature technology today; designs using 14nm feature sizes are on the anvil), we can expect the problem to increase in severity. Typically, designers use error-correcting codes to counter such effects.

The operating environment affects the charged particle flux. On the ground, in a benign environment, the most significant source of alpha particles is probably the small amount of radioactivity that is naturally present in the casing of the chip. In space applications, the particle flux can be much higher. The sun is the principal source of such particles: the solar wind regularly carries a large flux (the Earth's magnetic field shields the ground from most of this). The sun is, of course, a dynamic entity: it often generates flares, which may involve the ejection in short order of many tonnes of charged particles. When especially severe solar flares occur and where the Earth's magnetic field is not such a strong shield (near the poles), a significant increase in particle flux is seen even on the ground, affecting ground-based systems. For systems in space, the flux is far greater and the rate of transient failures goes up proportionately. Some relief can be found in shielding; however, even with such precautions, failure rates are high [Petersen 1997].

One final piece of terminology is appropriate to mention here. In fault tolerance, it is common to distinguish between a fault and an error. A *fault* is the underlying defect; an *error* is the external manifestation of that defect. Faults can exist for a long time before generating an error. In some cases, a transient fault may never generate an error. For example, a transient fault occurs in a memory cell when its value is changed; if the cell is updated before the next time it is read, that fault will never have generated an error.

2.4. Fault-Tolerant Structures

Hardware, time, information, and software redundancy are used for fault tolerance [Koren and Krishna 2007; Pradhan 1996; Sieworek and Swarz 1999]. Of these types, time and hardware redundancy will be of relevance to this article.

Of the many structures used in hardware fault tolerance, two stand out. One is the Triple Modular Redundancy (TMR) structure; the other is the Primary/Backup (PB) approach. In TMR, three processors run redundant copies of the same workload and mask errors by voting on their outputs. Since it requires triplicated hardware, TMR is expensive and used only in the most critical core of fault-tolerant systems.

In the PB approach, a processor executes code and its output is checked for correctness by an *acceptance test*. One common check is to see if the output is in the expected range. Another is to verify that it satisfies certain conditions: for example, to check that the square root of x has been correctly calculated, one can square the output to ensure that we get back a result close to x. (This is only practical because the squaring operation is of far lower complexity than the square root operation.) Acceptance tests are not perfect: they may miss incorrect outputs or falsely flag as erroneous a correct output. If the output fails the acceptance test, the execution of a backup copy of the failed task is initiated.

The principal difference between the TMR and PB approaches is that TMR entails *forward error masking*: multiple copies of the task are always executed. In PB, a backup

copy is only executed if the acceptance test flags a failure. Theoretically, there is no limit to the number of backup copies in order to respond to multiple failures of the same task execution.

It is worth pointing out that in the PB approach, the backup copies are not always clones of the original task. Backups may instead be lighter copies of the original, producing output of lower (but still acceptable) quality. The justification for this is that the backups are only occasionally invoked: it imposes lighter constraints on the scheduling algorithm if backups take less time; also, a more rudimentary implementation tends to be more reliable.

Fault-tolerant scheduling techniques generally assume the PB approach; if we use TMR, we have forward error masking and do not need to schedule backup copies.

The second type of redundancy that we need to consider is time redundancy. This is the slack that exists in a schedule between when tasks finish and their respective deadlines; this slack (during which a processor would normally be idle) can be exploited to provide a reserve of time during which to execute backup copies as necessary.

A good example of fault-tolerant structures is the *Time-Triggered Architecture* [Kopetz 1997; Kopetz and Bauer 2003]. Events are scheduled in advance and an a priori communication schedule can therefore be established. Fault tolerance is applied hierarchically. The system consists of a number of nodes interconnected by means of a communication network. The network is redundant for fault tolerance; for example, if a bus structure is used, multiple buses are provided. The nodes are connected to the network through a communications interface, and their communication profile is monitored by means of *guardians*. These guardians look for evidence of malfunction and disconnect the node from the network as needed.

Each node is a fault-tolerant unit, typically consisting of replicated processors and other hardware to ensure fault masking [Kopetz and Millinger 1999]. The clocks are synchronized by an algorithm that can tolerate Byzantine faults [Kopetz and Ochsenreiter 1987]. As long as individual unit failures do not overwhelm the fault-management capabilities of the node, they will be invisible to the rest of the system.

Replica determinism [Poledna et al. 2000] is maintained hierarchically [Kopetz and Bauer 2003]. The fault tolerance at the nodes ensures this at the interface between the node and the network. The *Time-Triggered Protocol* ensures replica determinism across the network. This supports transmissions that are scheduled a priori: messages are pulled off the interface queue and transmitted according to this schedule. The identity of a message can be discovered by reference to the very same schedule. This clearly requires effective clock synchronization across the network. The clocks can, for example, be synchronized by an algorithm that can tolerate Byzantine faults [Kopetz and Ochsenreiter 1987]. Since even the best synchronization mechanism cannot guarantee zero clock skew, each transmission time slot is flanked on each side by brief intervals of silence. Clearly, as long as the intervals of silence are longer than the maximum clock skew, all transmissions will be guaranteed to be seen by each receiver as occurring in the same clock tick.

When a system using a priori communication scheduling is used, enough slack must be provided in the schedule so that the system has the time to deal with a failure and still meet its communication schedule. Another option is to provide the system with alternative communication schedules that it can invoke whenever a fault has to be dealt with [Pop et al. 2007].

2.5. Voltage Scaling

The time taken to execute a task depends on the clock rate of the processor. The clock rate, in turn, is constrained by the circuit delays. It is well known that circuit delays are inversely proportional to the supply voltage. As a result, by changing the supply

voltage (within certain bounds), we can change the execution time of a task. The price we pay for this is power consumption, which is proportional to the square of the supply voltage and is linearly proportional to the clock frequency; in the aggregate, if we scale the frequency as allowed by the lower circuit delays that follow an increase in the voltage, the power consumption is proportional to the cube of the supply voltage. In real-time systems, the focus has been on how to *reduce* the supply voltage in order to minimize consumption while still meeting critical task deadlines [Unsal and Koren 2003]; in fault-tolerant scheduling, briefly *raising* the supply voltage (within the bounds tolerable by the processor) may give the processor the boost of speed that it needs to complete the backup. Keep in mind that failure is usually a comparatively rare event: occasionally spending additional energy to tide over a rare event is usually acceptable, even in power- or energy-constrained systems.

3. BASICS OF FAULT-TOLERANT SCHEDULING

At their foundation, all fault-tolerant scheduling procedures are similar. They aim to use time or processor redundancy to ensure that, despite the (permanent or transient) failure of a processor, the hard deadlines of critical real-time tasks will continue to be met.

The fault-tolerant scheduling approach that we describe here is meant to provide the system's first, emergency, response to a processor failure. When a transient failure occurs, such an emergency response may be all that is needed: the faulty processor may soon recover and resume its duties. If the processor failure is permanent, then if the scheduling is online, with tasks being either guaranteed or rejected as they come in, the system will take account of the reduced computational resources in guaranteeing any future task arrivals. If the failure is permanent and the scheduling is offline, then the system should generate another offline schedule (complete with backup tasks), targeted at the system with one fewer processor. Fault-tolerant scheduling algorithms provide the system with a means to continue meeting deadlines until this can be completed; their role is therefore to provide the system with some breathing room before it makes any longer-term adjustments to the task assignment and scheduling that may be appropriate.

Although there are some variations from one approach to another, the general method of responding to a failure is as follows:

—*Transient Failure*: If the system is designed only to withstand transients that go away quickly, re-execution of the failed task or of a shorter, more basic, version of that task is carried out. The scheduling problem reduces to ensuring that there is always enough time to carry out such an execution before the deadline.
—*Software Failure*: Here, the failure is that of the software, not of the processor. Software diversity is used: backup software that is different from the failed software is invoked. Again, we have to make sure, in preparing for software faults, that there is enough time for the backup version to meet the original task deadline.
—*Permanent Failure*: Backup versions of the tasks assigned to the failed processor must be invoked. The steps are as follows:
  —Provide each task with a backup copy.
  —Place the backups in the schedule, either prior to operation for offline scheduling or before guaranteeing the task for online scheduling.
  —If a processor fails, activate one backup for each of the tasks that have been affected.

In the rest of this article, we denote by π_i and B_i the primary and backup copies, respectively, of task T_i.

There are three general principles with respect to scheduling backups when permanent failures may occur:

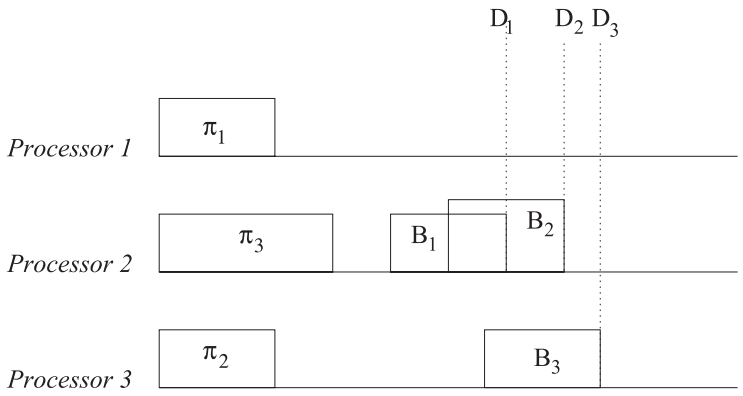

Fig. 1. Illustrating Principle A2.

—*Principle A1*: The backup of a task must not be assigned to the same processor as its primary copy: this is a rather obvious precaution against the same failure taking down both the primary and backup copy. Such a principle holds for all algorithms trying to protect against permanent processor failures or transients of significant duration. For approaches whose sole aim is to protect against *brief* transient failures or software bugs, this does not apply.

—*Principle A2*: The backup copies are *conditionally transparent* in the schedule [Krishna and Shin 1986]; that is, they can be *overloaded* [Ghosh et al. 1997]. By this we mean that unlike primary copies, which cannot be allowed to overlap other tasks in their schedule (after all, the processor can only execute one thing at a time; note that for multicore chips, each of the cores counts as a processor), we can overlap backups in the schedule *as long as their primary copies have not been scheduled on the same processor*. This is because all we are trying to protect against is a single failure. (This is like having just one spare tire in your car: you don't know ahead of time which one of the four "primary" tires is going to fail, but just one spare will be enough to replace any one of them.) Figure 1 illustrates this for a three-task system. Backups B_1, B_2 whose primary copies have been scheduled on different processors cannot both need to be activated to react to the same processor failure.

—*Principle A3*: A backup copy may not overlap any primary copy in the schedule. The reason for this should be obvious. (Once we know that a particular backup will not be needed, however, the space that it occupies in the schedule can obviously be used by other tasks.)

Throughout, unless otherwise stated, we assume that when processors fail, this is detected. In particular, it is detected by the time any task that is currently running on that processor is scheduled to complete its execution. Further, we assume that in a multiprocessor, one processor failure cannot induce another. Failure of the communication network is not considered, since the focus here is on rapid recovery from *processor* failures. Network failures can be guarded against by providing multiple paths connecting any pair of processors.

4. TRANSIENT HARDWARE FAILURE

4.1. Periodic Tasks: Static Approach

The failure model we start with is that of brief transients affecting a single processor running a workload comprised of periodic tasks [Burns et al. 1996]. It is assumed that we do not have transient failures occurring more than once every τ_F seconds.

These tasks are independent of one another and have no precedence constraints—that is, one does not provide input to another. The Rate Monotonic (RM) scheduling policy is used [Liu and Layland 1973]. RM is perhaps the best studied of all real-time scheduling algorithms; it is a preemptive algorithm meant for a workload comprising only periodic tasks,[4] where the priority of a task is inversely related to its period. That is, if we have two tasks T_1, T_2 with periods P_1, P_2, respectively, such that $P_1 < P_2$, then T_1 has higher priority than T_2. If two tasks have identical periods, then the tie can be broken arbitrarily. RM is a preemptive algorithm: a task only executes when there is no higher-priority task waiting to execute. It is assumed that the cost of preemption is negligible. To begin with, we will assume that transients are of negligible duration—that is, a processor crashes and comes back up again very quickly. Later, we will relax this assumption.

Under these assumptions, there is just one processor; if a task is affected by failure, it is re-executed in the form of a backup.

To check whether a given task set is schedulable (i.e., whether all its tasks can meet their deadlines), we follow the approach due to Joseph and Pandya [1986]. The worst-case response time for a task occurs when that task is released at the same time as one copy of every higher-priority task in the system (a proof is provided in Liu and Layland [1973], but the intuition behind it should be obvious). Let us number tasks in descending order of priority, with T_1 being the highest-priority task and T_n the lowest (for some n). Suppose that e_i is the WCET of task T_i. Then, the response time of task T_i is bounded by

$$R_i = \begin{cases} e_i & \text{if } i = 1 \\ e_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{P_j} \right\rceil e_j & \text{otherwise.} \end{cases} \quad (1)$$

This expression is not hard to derive. T_1 is the highest-priority task in the system, so it is not delayed by any other task. It therefore starts executing the moment that it becomes ready to execute and finishes no later than its WCET.

For T_i when $i > 1$, a task may be delayed by having to give way to any higher-priority tasks that are ready to execute. The maximum number of iterations of task T_j that lies in the interval $[0, R_i]$ is $\lceil \frac{R_i}{P_j} \rceil$. Since T_j takes at most e_j time to execute, the maximum time that any iteration of T_i will be delayed by having to wait for higher-priority tasks to execute is $\sum_{j=1}^{i-1} \lceil \frac{R_i}{P_j} \rceil e_j$. Adding the WCET of T_i itself completes the expression.

For $i > 1$, Equation (1) is an implicit equation in R_i; however, it is quite easy to solve iteratively. Let $R_i^{(k)}$ be the kth iteration of R_i. Consider the iterative equation

$$R_i^{(k+1)} = e_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^{(k)}}{P_j} \right\rceil e_j$$

and start with $R_i^{(0)} = e_i$. This iteration will converge to the solution for (1) or we will, at some point, get a response time that exceeds its deadline, which indicates inability to meet the deadline.

Now, consider what happens if there is a transient failure. We need to make assumptions about which tasks are affected by the failure. Let us assume that the failure destroys all results associated with the task that is executing at the moment the fault strikes and nothing else. That is, if a task has been preempted and is in a partially

[4]Traditionally, periodic tasks in an RM-scheduled system are numbered in decreasing order of priority—that is, T_i has higher priority than T_j if $i < j$. In addition, the RM framework has been extended to allow for sporadic tasks by reserving space for them in the schedule [Lehoczky et al. 1987].

completed state, the already-done portion of its computation will remain unaffected. This is reasonable if we assume that such tasks have their state stored in memory with sufficient error-correction coding.

Based on this, we can write an implicit equation for the response time of task T_i as follows:

$$R_i = \begin{cases} e_i + \left\lceil \frac{R_i}{\tau_F} \right\rceil e_i & \text{if } i = 1 \\ e_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{P_j} \right\rceil e_j + \left\lceil \frac{R_i}{\tau_F} \right\rceil \max(e_1, e_2, \ldots, e_i) & \text{if } i > 1. \end{cases} \quad (2)$$

By comparing Equations (1) and (2), we see that the only new term on the RHS of (2) is $\lceil R_i/\tau_F \rceil \max(e_1, e_2, \ldots, e_i)$. This comes about because the fault may strike at a time when any task is executing. If the executing task at the moment of failure is of lower priority than T_i, its interruption makes no difference to when T_i completes. If it is task T_i itself that is hit, it will cause a maximum additional load of e_i to be imposed on the system; if it is some higher priority task, T_j, it will cause an additional load of e_j that will be visible to T_i. Such a delaying effect happens no more than once every τ_F seconds.

In our derivation so far, we have assumed that the transient fault is brief, which is why its duration is not taken into account in our equations. If we want to relax this assumption, we can simply add the transient-recovery time, t_{rec}, to our equations. Thus, Equation (2) will be replaced by

$$R_i = \begin{cases} e_i + \left\lceil \frac{R_i}{\tau_F} \right\rceil (e_i + t_{rec}) & \text{if } i = 1 \\ e_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{P_j} \right\rceil e_j + \left\lceil \frac{R_i}{\tau_F} \right\rceil \{\max(e_1, e_2, \ldots, e_i) + t_{rec}\} & \text{if } i > 1. \end{cases} \quad (3)$$

If this worst-case response time is no greater than the task's relative deadline (relative to its release time) for all of the tasks, the task set is fault-tolerant schedulable.

One other notable result in fault-tolerant schedulability is the scheduling bound derived in Pandya and Malek [1998], where it is shown that under assumptions typically made in real-time periodic scheduling (deadlines equal to periods, all tasks periodic and independent), any periodic task set with a utilization no greater than 0.5 can sustain up to one transient failure without missing any deadlines. This is true even under the assumption that not only the executing task but all uncompleted and preempted jobs are lost upon being struck by a fault. Note that this is a *sufficient*, not a *necessary*, condition: although every task set that has utilization no greater than 0.5 can tolerate up to one fault, there are several task sets with utilizations exceeding this bound that are also fault tolerant. Bounds for multiple faults have also been derived: see Ghosh et al. [1997].

4.2. Sporadic Tasks: Online Convolutional Approach

We now present an online convolutional approach, originally presented for sporadic tasks using the EDF scheduling algorithm [Liberato et al. 2000]. In contrast with many of the other algorithms surveyed here, this algorithm can deal with multiple failures. *Convolutional* refers to the computational approach used to determine whether the system can sustain schedulability. As for the sporadic task loading, note that any procedure that works for sporadic task sets can be extended to periodic task sets by considering every task that is released over the Least Common Multiple (LCM) of the task periods. Finally, although the algorithm was originally presented on an EDF scheduling base, there is nothing to prevent some other priority rule from being used.

The focus will be on manipulating the unfinished work at any point in time. In the absence of faults, the unfinished work function consists of jumps when a task arrives, bringing new work, which are then worked off as the processor executes. Denote by $\mathcal{W}(\mathcal{T}, t)$ the unfinished work at time t due to task set $\mathcal{T}$. If we reckon time in discrete

units of one (rather than as a continuous quantity), we can write a recursion for the unfinished work as follows:

$$\mathcal{W}(\mathcal{T}, t+1) = (\mathcal{W}(\mathcal{T}, t) - 1)^+ + \alpha(t+1), \tag{4}$$

where $\alpha(t+1)$ is the amount of work, if any, brought in by a task arriving at time $t+1$, we start things off with $\mathcal{W}(\mathcal{T}, 0) = \alpha(0)$, and define $(x)^+ = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$.

Let f_i denote the time when T_i completes execution if there are no faults. Now, consider the fault pattern $\Phi = (\phi_1, \phi_2, \ldots, \phi_n)$, where ϕ_i is the number of faults that strike during execution of T_i, $i = 1, \ldots, n$. Rather than condition this on when these failures actually occur, we "book" them for accounting purposes as if they all arrived at time f_i and that each failure of a given task imposes a load equal to the WCET of that task. Under such a fault pattern, the unfinished work recursion can be written as

$$\mathcal{W}^{\Phi}(\mathcal{T}, t+1) \;=\; (\mathcal{W}(\mathcal{T}, t) - 1)^+ + \alpha(t+1) + \beta_i(t+1), \tag{5}$$

$$\text{where } \beta_i(t) \;=\; \begin{cases} \phi_i e_i & \text{if } t = f_i \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

Now, define

$$\delta(\mathcal{T}, \Phi, t) = \mathcal{W}^{\Phi}(\mathcal{T}, t) - \mathcal{W}(\mathcal{T}, t). \tag{7}$$

Define by $\delta^w(\mathcal{T}, t)$ the maximum *additional* unfinished work at time t due to running task set $\mathcal{T}$ and having been struck by w failures. Note that this captures the worst-case unfinished work given w failures: that is, it represents the additional unfinished work if the failures arrive at the most inconvenient times possible.

Key to the algorithm is the observation that the lowest-priority task $T_\ell \in \mathcal{T}$ meets its deadline under the w-fault case if and only if $\delta^w(\mathcal{T}, t) = 0$ for some $t \in [f_\ell, D_\ell]$. For a proof, see Liberato et al. [2000]; however, the result should be quite intuitive to the reader.

To obtain $\delta^w(\mathcal{T}, t)$, we use a convolutional approach. Note that, by definition, $\delta^0(\mathcal{T}, t) = 0$ for all t. Now, assume that the tasks are numbered from 1 in order of their finishing time in the fault-free schedule. Define $\delta_i^w(\mathcal{T})$ as the maximum additional work at time $t = f_i$ caused by w faults that may strike any task executing in the interval $[0, f_i]$. We can now write the key recursion for $\delta_i^w(\mathcal{T})$ (for convenience, define $\delta_0^w(\mathcal{T}) = 0$):

$$\delta_i^w(\mathcal{T}) = \begin{cases} 0 & \text{if } i = 0 \\ we_1 & \text{if } i = 1 \\ \max\left\{ \left(\delta_{i-1}^w(\mathcal{T}) - slack(f_{i-1}, f_i)\right)^+, \delta_i^{w-1}(\mathcal{T}) + e_i \right\} & \text{otherwise,} \end{cases} \tag{8}$$

where $slack(f_{i-1}, f_i)$ is the free time in the fault-free schedule in the interval $[f_{i-1}, f_i]$ (which time is available to execute some of the fault-induced additional workload).

The cases for $i = 0$ and $i = 1$ should be obvious. The case for $i > 1$ chooses the greater of two instances: the first in which all w faults affect tasks $T_1, \ldots, T_{i-1}$, and the other where $w - 1$ faults affect tasks $T_1, \ldots, T_i$ and an additional fault strikes T_i.

From the definition of $\delta^w(\mathcal{T}, t)$, we can write the following expression:

$$\delta^w(\mathcal{T}, t) = \left(\delta_i^w(\mathcal{T}) - slack(f_i, t)\right)^+, \quad f_i \le t < f_{i+1}.$$

If T_ℓ is the lowest-priority task in $\mathcal{T}$ and $\delta^w(\mathcal{T}, t) = 0$ for some $R_\ell < t \le D_\ell$, we conclude that task T_ℓ will meet its deadline despite the fault pattern $\mathcal{F}$.

Table II. Task Characteristics

Task	r_i	e_i	D_i
T_1	0	3	10
T_2	3	7	15
T_3	4	2	12
T_4	13	5	20

To check that *every* task in $\mathcal{T}$ can meet its deadline, we evaluate $\delta^w(\mathcal{T}, t)$ over each of the task sets $\mathcal{T} = \{T_1\}, \{T_1, T_2\}, \ldots, \{T_1, T_2, \ldots, T_n\}$ to check for schedulability of $T_1, T_2, \ldots, T_n$, respectively, where the tasks are now numbered in ascending order of absolute deadline (and thus descending order of priority with respect to the EDF scheduling algorithm). Pseudocode for the algorithm is provided in Algorithm 1.

ALGORITHM 1: Convolution Algorithm: Feasibility Check

Define $\mathcal{T}_i$ as the task set consisting of the i highest priority tasks;
Initialize ℓ to 1. `// ℓ will be the index of the lowest-priority task in the task set under consideration; we will start with` $\mathcal{T}_1$`, consisting of just task` T_1
for $j \leftarrow 1$ **to** n **do**
  Simulate EDF under the task set $\mathcal{T}_j$ and record the task finishing times, $f_1, \ldots, f_j$;
  Define $f_{j+1} = D_\ell$. Record the slack over the intervals $[f_i, f_{i+1}]$ for $i = 1, \ldots, j$;
  Renumber the tasks in ascending order of their finishing times;
  Compute δ_i^w, $i = 1, \ldots, j$; $w = 1, \ldots, k$;
  if *($\delta_i^w > slack(f_i, f_{i+1})$ for all $i = \ell, \ldots, j$* **then**
    the lowest-priority task in $\mathcal{T}_j$ cannot be feasibly scheduled: *reject the incoming task as not schedulable and exit the algorithm;*
  end
  else
    if *($j == n$)* **then**
      the incoming task can be scheduled together with all previously guaranteed tasks in the system. *Issue a guarantee for the incoming task and exit the algorithm*;
    end
  end
  Set ℓ equal to the index of the lowest-priority task in $\mathcal{T}_{j+1}$;
end

As an example, consider sporadic tasks that arrive according to Table II, where r_i is the release time and D_i is the absolute deadline of task T_i: we want to tolerate up to $k = 2$ failures. Let us consider what happens upon the arrival of each task (tasks are assumed to arrive at their release times; the execution time of the online algorithm to check schedulability is assumed to be negligible).

When task T_1 arrives at time 0, the task set is $\mathcal{T} = \{T_1\}$. We have $f_a = 3$. We have $\delta_1^{w=2}(\{T_1\}) = 2e_1 = 6$. The slack time over $[3, 10]$ for task set $\{T_1\}$ is 7. Since $\delta_1^{w=2}(\{T_1\}) < 7$, the system can run this task successfully even in the face of up to two failures; T_1 is therefore guaranteed by the system.

Now, T_2 arrives at time $t = 3$. It is easy to see that $f_b = 10$. The slack over $[10, 15]$ is 5 (since none of that interval is needed by either T_1 nor T_2). We have

$$\begin{aligned}\delta_2(\{T_1, T_2\}) &= \max\left(\left(\delta_1^{w=2}(\{T_1\}) - slack(3, 10)\right)^+, \delta_2^{w=1}(\{T_1, T_2\}) + e_2\right)\\ &= \max((6-0)^+, 7)\\ &= 7.\end{aligned}$$

Since $\delta_2(\{T_1, T_2\}) > slack(10, 15)$, T_2 cannot be guaranteed and is therefore rejected. The guaranteed task set remains $\{T_1\}$.

At time $t = 4$, task T_3 arrives. In the absence of failures, T_3 would finish executing at time $f_3 = 6$. The slack over the period $[6, 12]$ is equal to 6 since there is nothing scheduled for that time in the no-fault case. We have

$$\begin{aligned}\delta_3(\{T_1, T_3\}) &= \max\left((\delta_1^{w=2}(\{T_1\}) - slack(3, 6)\right)^+, \delta_3^{w=1}(\{T_1, T_3\}) + e_c \\ &= \max(6 - 1)^+, 2 + 2) \\ &= 5 < 6.\end{aligned}$$

Hence, T_3 can be guaranteed by the system.

At time $t = 13$, task T_4 arrives. By this time, tasks T_1 and T_3 have left the system; hence, we treat T_4 as the first task:

$$\begin{aligned}\delta_4(\{T_4\}) &= 2e_4 \\ &= 10 > slack(18, 20).\end{aligned}$$

It follows from this that T_4 cannot be guaranteed to survive two failures. T_4 is therefore rejected by the system.

4.3. Aperiodic Tasks: Adaptive Fault Tolerance

When tasks arrive in a sporadic and unpredictable manner, one approach is to give that task as high a level of fault tolerance as possible, consistent with the need to meet the deadline of all tasks [Gonzalez et al. 1997].

Three fault-tolerance approaches were considered in Gonzalez et al. [1997]: TMR, PB, and Primary/Exception (PE). TMR and PB have been described in Section 2.4. PE is similar to PB, except that instead of initiating a backup task upon the failure of the primary, the system generates an exception. One variation on PB is SCP-TMR, where the primary element consists of a duplex, forming a Self-Checking Pair (SCP). Self-checking is carried out by having each element of the duplex execute the task and comparing the results generated by them. If they agree, the self-check is passed; if they disagree, a third processor (the backup) is invoked and has the casting vote.

When a task arrives, the scheduler guarantees to it whichever level of fault tolerance the system can afford, based on what has already been committed to other tasks; a guarantee is then issued to this task for this level of fault tolerance. Alternatively, the scheduler can postpone committing to a particular level of fault tolerance to the point when the task actually starts executing. In this case, initially, only a minimal level of fault tolerance is guaranteed to the application. If a higher level is feasible at the point when this task starts executing, this is provided. This late-commitment approach allows the scheduler to potentially take better care of other tasks that arrive after this task did but before it commenced execution. If a task arrives to find insufficient resources, it cannot be guaranteed by the scheduler.

Two metrics were used to characterize this algorithm: the *guarantee ratio*, which is the fraction of the number of tasks guaranteed execution by the scheduler, and the *average replication factor*, which is the ratio of the sum of all task copies to the number of task arrivals to the system. The greater the guarantee ratio, the more tasks can be guaranteed; the greater the replication factor, the greater is the expected fault tolerance.

Detailed simulation results for this algorithm are provided in Gonzalez et al. [1997]. They indicate that adaptive techniques are generally significantly better than static approaches, where the fault-tolerance level is fixed a priori.

5. SOFTWARE FAULTS

Software fails when it is given a set of inputs for which it produces an incorrect output. The fault lies in the software; it is assumed that the underlying hardware is fault free. As before, we assume that there is some effective mechanism to detect when an output is erroneous.

To survive software failure, we use alternative versions. Unlike in the cases considered earlier, where we were trying to protect against hardware faults, these backup versions cannot be copies of the same software: if they were, they would all fail on the same set of inputs, and multiple copies would be useless. Equally, note that the hardware fault-tolerant approaches mentioned earlier would also work for software failures as long as the re-executed versions were different from the one that failed for that input.

Quite often, they are simpler versions of the original software: the aim is to provide some rudimentary alternative that will allow the system to continue functioning acceptably. As mentioned earlier, a rudimentary piece of code is often easier to debug than one that is more complex and can therefore be more reliable. In addition, it can take less time to execute and might therefore fit more easily into a schedule.

5.1. Periodic Tasks

We now look at an approach that seeks to deal with software faults [Han et al. 2003]. There is a single processor, and there is no requirement on the part of the algorithm to protect against hardware faults.

All tasks in the system are assumed to be periodic. If the LCM of the periods is P, then the algorithm carries out preprocessing to fix the position of the alternatives in the schedule over the interval $[0, P]$. This same schedule then repeats endlessly. That is, if an alternative is scheduled to execute at some point in time $t \in [0, P]$, it is also scheduled to execute at times $t + P, t + 2P, \ldots$.

The basic idea behind this algorithm can be stated as follows. The alternatives are the key to reliability: we have to ensure that they have enough time to be executed. We reserve time in the schedule to ensure that this happens (barring a failure of the alternatives). Outside this reserved area, the system can execute primary versions, secure in the fact that enough time is available to run an alternative.

If both the primary and its backup fail, there is no remedy left. However, the same approach can be applied recursively—that is, one can have backups to the backups, and so on.

The algorithm can be broken down into two phases: offline and online. In the offline phase, we develop a task schedule for the alternatives over the period $[0, P]$. In the online phase, we use this task schedule as a guide to deciding which task to run. The details are broadly as follows:

—*Offline Phase*:
  —Over the interval $[0, P]$, schedule all of the alternatives, according to principles that we will describe later. Denote by A the union of intervals during which the alternatives are scheduled. This schedule repeats with a period of P—that is, if $t \in A$ over $[0, P]$, we must have $t + iP \in A$ as well for $i = 1, 2, \ldots$. Denote by α_i the time when the alternative to task T_i is scheduled to start executing.

—*Online Phase*:
  —Different priority schemes apply depending on whether we are in an interval within A or not. Within A, the alternative that is scheduled for that time has priority over everything else. Outside A, the primaries have priority; these can be scheduled according to any priority rule. RM and EDF are obvious alternatives; however, the

Task	Period	Execution Time	
		Primary	Backup
T_1	3	2	1
T_2	4	1	1

(a) Task Parameters

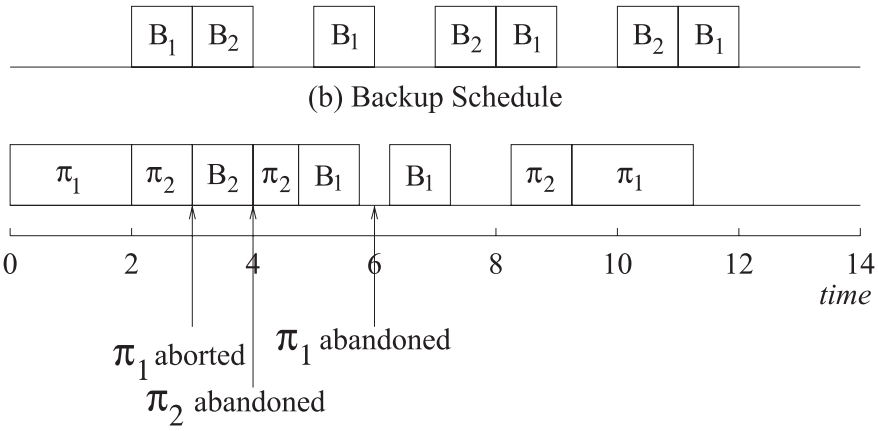

Fig. 2. Guarding against software faults.

task priorities may also be modulated by the value that the task confers on the system.

—Before any primary starts executing, the system checks to see if enough time is available in the schedule to complete it. If not, execution of the primary is abandoned.

—If the primary of task T_i finishes before α_i, its alternative is canceled and its assigned interval within the schedule is removed from A.

—If the primary of task T_i does not finish before α_i, or it is known to have failed before then, all further execution of the primary is abandoned and the alternative starts executing.

—Although α_i is the point at which a backup for task T_i is supposed to start executing, if the processor finds itself idle before then, it can start executing backups early. However, as mentioned previously, outside the set of intervals, A, the backup(s) have lower priority than primary tasks.

We can pick any mechanism to schedule the alternatives that allows their deadlines to be met. However, logic dictates that the primaries be given as much of a chance to finish executing as possible. The obvious heuristic is to schedule the backups as late as possible. The approach taken is to start at the end of the scheduling interval $[0, P]$ and work backward. That is, we schedule a backup with execution time e_b and absolute deadline P to occupy the interval $[P - e_b, P]$. We then go backward from this point, scheduling backups in this way in descending order of their absolute deadlines, with ties being broken arbitrarily.

Let us now consider an example. We have a task set with the parameters shown in Figure 2(a). The LCM of the periods is 12. First, we generate an offline schedule of the backups over the interval $[0, 12]$. This is shown in Figure 2(b).

At time 0, the first iterations of T_1 and T_2 are issued. Let us use the RM algorithm to schedule the primaries. Task T_1 is issued, and its primary, π_1, runs to completion; it finishes at time 2. Since T_1 has executed successfully, its backup, B_1 (scheduled for the interval $[2, 3]$), can be canceled. There is now time for the primary of T_2 to execute. Suppose that π_2 fails to produce its output by time 3. Then, it is aborted and its backup runs in the slot provided for it (in the interval $[3, 4]$).

A time 3, the second iteration of T_1 is released; however, it is not allowed to start executing since the processor is running a backup in its assigned slot. Thus, it has to wait until time 4. At time 4, the processor checks that there is not enough time to finish π_1 by time 5, so the primary of T_1 is abandoned. At time 4, the second iteration of T_2 arrives: it has lower priority than the primary of T_1, but since that has been abandoned, it is allowed to start executing. Suppose now that this primary finishes early: at time 4.5. This causes its backup (scheduled for the interval $[7, 8]$) to be canceled. At time 4.5, there is no primary waiting to execute, so the backup for task T_1 can start early. This backup runs over the interval $[4.5, 5.5]$, at which time the processor falls idle. At time 6, the third iteration of T_1 arrives. It is again abandoned since there is insufficient time to execute it. And so on. See Figure 2(c).

In the previous discussion, we assumed that we know precisely how long we should allow for the primary to execute before abandoning it. The actual execution time, however, is a random variable, which can differ substantially from its estimated WCET. In many instances, not allowing a task to start running may be overly conservative. Letting a primary run if the available time is less than its WCET is a gamble: if it finishes, we have gained a primary execution (which, presumably, is superior in quality to that of its backup); if it fails, we have wasted processor time that could well have been used to successfully execute some other primary. If we have the probability distribution function of the primary execution time, we could conduct offline experiments to determine what primary execution time we should allow for in order to obtain good results. The more mathematically ambitious could conduct an analysis based on Markov decision theory [Hillier and Lieberman 2001] to determine the appropriate primary execution time allowance. Note that we can safely gamble on the execution time required by the primary, since there is always enough time for the backup to execute if the primary should fail to deliver on time.

In the special case where periods are multiples of one another—for example, $P_i = m_i P_{i-1}$ where $m_i \in \{1, 2, \ldots\}$—we can use the approach of Liestman and Campbell [1986]. Once again, the aim is to ensure that the backups always have enough time to execute successfully; the primary versions are regarded as contributing to performance but not essential to reliability. Hence, the aim is to schedule all backups feasibly and do a best-effort attempt on the primary versions.

We start by drawing up a schedule over $[0, P_1]$. This always contains the backup B_1; if there is enough time, it includes the primary, π_1. Call this schedule S_1. Next, we move to construct S_2, a fault-tolerant schedule over $[0, P_2]$, consisting of tasks $\{T_1, T_2\}$, then S_3 over $[0, P_3]$, and so on.

To construct S_i, we start by laying out m_i copies of S_{i-1} end to end. Then, B_i is scheduled on top of this schedule, removing one or more already-scheduled primary copies if necessary. Then, we check to see if there is enough idle time to schedule π_i. If there is, we do so. Otherwise, we may either give up trying to schedule π_i or explore the possibility of dropping one or more copies of some other primary, π_j to make room for it.

As an example, consider a three-task system, with task periods $P_1 = 10, P_2 = 20, P_3 = 40$; the primary execution times are $e_1 = 7, e_2 = 4, e_3 = 6$, and the backup execution times are $\xi_1 = 2, \xi_2 = 3, \xi_3 = 3$. Figure 3 shows the process of constructing the fault-tolerant schedule. We start by constructing S_1, consisting of just the task T_1.

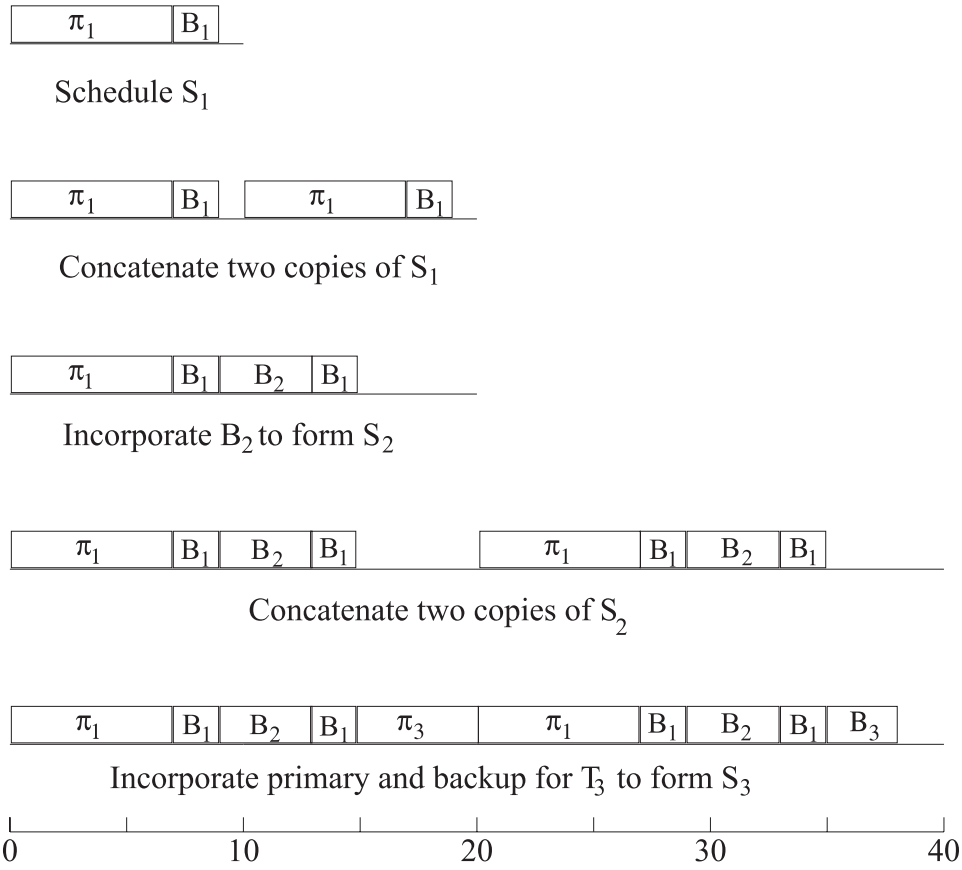

Fig. 3. Concatenation approach.

It is easy to check that there is enough time to schedule both the primary and backup copies.

To construct S_2, we concatenate two copies of S_1 and use that as a framework within which to schedule T_2. We have a total of two time units available: this is not enough to schedule either the primary or backup of T_2. Hence, we drop one of the two iterations of π_1: this releases enough time to schedule B_2.

Finally, we construct S_3. Concatenate two copies of S_2: there is enough time to fit both a primary and a backup copy of T_3 in this schedule.

Finally, if we can accelerate the processor by means of voltage scaling, the time required to be reserved for the backups will be reduced. This, however, is only acceptable as long as the probability of activating the backups is small enough that the additional expected energy expenditure is affordable.

An online variation on this problem has been addressed as well [Caccamo and Buttazzo 1998]. The focus is on tasks that are scheduled as they arrive; as before, each task has primary and backup copies. Primary copies are susceptible to timing overruns, whereas backup tasks have well-defined WCET. If the primary cannot be scheduled before the backup is to run, we rely on the output of the backup. If it is scheduled, based on its estimated execution time, and then proceeds to overrun this time, the primary is aborted and the backup is executed. If, on the other hand, the primary completes successfully without overrunning its time, the backup need

not execute; its budgeted execution time can be released for use by other tasks. The scheduling algorithm must ensure that, at all times, the backup has enough time to execute successfully in the event that the primary does not.

5.2. Sporadic Tasks

When the task set is sporadic, a dynamic scheduling approach can be taken. When tasks arrive, a server is responsible for deciding whether or not a task can be guaranteed execution within its deadline. If it cannot, the application is notified. If it can, primary and backup copies of the task are assigned to processors.

One approach to dynamic fault-tolerant scheduling in a multiprocessor system is reported in Manimaran and Murthy [1998]. As tasks arrive, the scheduler checks the current loading of each of the processors. It then schedules the newly arriving tasks so that (a) the primary and backup are not assigned to the same processor, and (b) the backup is not scheduled to start before the scheduled completion time of the primary, with (a) ensuring that that failure of a single processor does not kill both copies of the same task and (b) that potentially wasteful overlapped execution of the primary and backup copies does not happen. Any dynamic scheduling approach can be used; for instance, Manimaran and Murthy [1998] incorporate their approach within the myopic scheduling algorithm [Ramamritham et al. 1994] of the Spring real-time kernel [Stankovic and Ramamritham 1989].

The myopic algorithm uses a heuristic to schedule tasks one by one on a multiprocessor. The schedule is therefore built up incrementally, task by task. In the event that the latest increment of the schedule is unable to schedule the next task on the list, backtracking is used; the algorithm rolls itself back to an earlier point in the evolution of the schedule and tries again.

Manimaran and Murthy [1998] modify the myopic scheduling algorithm to include fault tolerance. The algorithm proceeds by specifying a minimum distance (in time) between the primary of a task and its backup. It also ensures that primaries and backups of the same task do not get scheduled on the same processor. Within these constraints, the standard myopic algorithm can be run to develop a fault-tolerant schedule. Overloading can also be built in (see Principle A1, shown earlier in article): two backups can intersect in the schedule as long as their primaries are not assigned to the same processor. This obviously presumes that we only have one processor failure at a time.

6. PERMANENT FAULTS

To deal with permanent faults, we obviously require hardware redundancy. By far, the simplest way to deal with such faults is to have spare processors in the system. These are idle to begin with; when a processor fails, its schedule is switched over to that of the replacement processor. Such an approach, although simple, may not be appropriate as an emergency (rather than as a long-term) response for the following reasons:

—If, as is generally the case, we have processors with private, rather than shared, memory, then one of two possibilities exist:
  —Copies of all tasks that this spare processor may have to execute are preloaded in its memory. This may require more memory capacity at each of the spares than we can afford.
  —Once the system knows which tasks need to be run on the spare processor, those tasks can be loaded into its memory. This obviously takes time, which may not be available in a real-time system. (As mentioned earlier, this may well be a good long-term response to a failure; we are concentrating in this article on short-term responses).

—Having idle spares usually wastes energy. If all available processors are actively involved in executing the workload, the workload per processor is less. As a result, processors can be run at a lower clock rate and still meet all task deadlines. This, in turn, allows us to use a lower supply voltage, which saves energy. Recall that energy consumption to execute a given workload is a quadratic function of the supply voltage.

6.1. Periodic Tasks

The task set, which is known in advance, consists of periodic tasks. We make the common assumption that the deadline of each task iteration is equal to its period—that is, the deadline of an iteration occurs at the same time as the release time of the next iteration of that task.

The overall task scheduling problem breaks down into two parts. The first is an assignment of tasks to processors. The second is to use the selected uniprocessor algorithm to schedule the assigned tasks on each processor.

The problem of task assignment and multiprocessor scheduling for any but the most trivial cases has long been known to be NP-complete [Garey and Johnson 1979]. Heuristics must therefore be used. The 0-1 knapsack heuristics [Cormen et al. 2004] are probably the most commonly used such procedures. In particular, we will explore here the use of the *first fit* assignment algorithm.

In the first-fit algorithm, one tries to assign tasks to processors one by one in some order. A task is assigned to the first processor on which it can be feasibly scheduled. Let us start by illustrating such an assignment procedure *without* any provision for fault tolerance.

Suppose that we are using the RM scheduling algorithm. We pick one task after another, in order of their RM priority (ties may be broken arbitrarily). Recall that the RM priority[5] of a task is inversely related to its period.

For each task, we identify the first processor on which it can be feasibly assigned. By feasible assignment, we mean that this task together with the tasks, if any, already assigned to that processor, can all meet their deadlines. This can be checked by using the response-time expression (Equation (1)) encountered earlier.

There are variations on this algorithm that immediately suggest themselves. First, the order in which the tasks are assigned need not be in RM priority order. They could follow some other ordering—for example, we could choose to order the longer tasks (i.e., those with a longer WCET) first or simply pick them at random. However, the convenience in assigning tasks in descending order of priority is that once we calculate the worst-case response time of an assigned task, that time does not change with subsequently assigned tasks, since these are always of lower priority.

In addition, the order in which we try the processors need not be the same throughout: we could simply try them randomly. However, it has been observed that first-fit is generally more successful in making task assignments than, say, heuristics, which try to assign the next task to the processor with the lightest load (such an algorithm aims to balance the processor utilizations). The reason is that in first-fit, the later-numbered

[5]It is a common mistake to assume that the highest-priority task is always the one of greatest practical importance to the application. That is not necessarily true. The RM algorithm always assigns priority in order of the frequency with which a periodic task is executed. Under this assumption, and the assumption that the task deadline coincides with its period (i.e., that a particular iteration of a task has to be completed before the next iteration arrives), it can be shown that this is an optimal static-priority uniprocessor scheduling algorithm: *static priority* because the relative priority of tasks are fixed for all time; *optimal* in that if this algorithm cannot find a feasible schedule for a set of tasks on a given *single* processor, there is no other static priority algorithm that can. By contrast, practical importance to the application depends on the needs of the application. There is no necessary link between the period of a task and its contribution to the application.

processors tend to be less heavily loaded so that longer tasks are more likely to find themselves a home in them.

6.1.1. All Periods Equal. We start with a particularly simple case: one in which all of the tasks are periodic, have the same period, and are released at the same time [Oh and Son 1992]. Although this is simple, it is by no means uncommon in real-time systems. This approach is designed to protect against up to one failure.

The idea is to assign one copy of each task to processors, using, for example, the first-fit procedure, with the tasks being assigned in decreasing order of their execution times. Then, provide each processor, $\mathcal{P}$, with a twin and assign to that twin all of the tasks assigned to $\mathcal{P}$. In other words, processors are grouped in pairs, and both processors in a pair are assigned exactly the same tasks. Checking that a set of tasks assigned to the same processor can all meet their deadlines is very simple when all tasks are released together and have the same period: simply check that the total execution time of the assigned tasks is not greater than their (common) period.

Once the tasks have been assigned, they can be scheduled. The tasks assigned to the same processor are scheduled in decreasing order of execution time. Then, a swapping procedure is carried out within the schedule of tasks in each pair to ensure that each task has a primary copy on one processor in a pair and a nonoverlapping backup copy on the other.

The swapping procedure proceeds as follows for each pair. For definiteness, label one processor $\mathcal{P}_A$ and the other $\mathcal{P}_B$. Suppose that the total length of the schedule for each processor in some pair is L. Let the task completion times be $f_1 \leq f_2 \leq \cdots \leq f_n$. (Without loss of generality, number the tasks according to their finishing times in this schedule; that is, the finishing time of T_i is f_i.) Let k be the largest number such that $f_k \leq L/2$. All tasks in the set $\tau_A = \{T_1, \ldots, T_k\}$ are considered to be primary tasks in processor $\mathcal{P}_A$ and backups in processor $\mathcal{P}_B$; tasks in the set $\tau_B = \{T_{k+1}, \ldots, T_n\}$ are considered to be primary tasks in processor $\mathcal{P}_B$ and backups in processor $\mathcal{P}_A$.

Now, we proceed to swap the positions of the tasks in the schedule of processor $\mathcal{P}_B$. Move all tasks in τ_B to the beginning of its schedule; shift the tasks in τ_A to after those in τ_B. It is possible to prove that the tasks will not overlap in time on the two processors.

It is trivial to see that each pair of processors can suffer up to one failure without any deadlines being missed. However, such a pairing approach cannot use backup overloading, and this can be expected to require somewhat more processors than other approaches.

As an illustration, consider the task set with execution times $e_1 = 7, e_2 = 5, e_3 = 4, e_4 = 3, e_5 = 1$. The common period of all tasks is $P = 20$; note that all of these tasks will (just about) fit on a single processor. Assign one copy of each task on each member of a processor pair.

We schedule the tasks on both processors in descending order of their execution time so that their finishing times are $f_1 = 7$, $f_2 = 12$, $f_3 = 16$, $f_4 = 19$, and $f_5 = 20$. f_1 is the largest time that is no greater than $\lfloor P/2 \rfloor = 10$. Thus, we label T_1 on Processor 1 as the primary copy and all other tasks on that processor as backups. Conversely, the copies of tasks T_2, T_3, T_4, and T_5 assigned to Processor 2 are the primaries and those assigned to 1 are secondaries.

We now swap task positions so that the primaries on each processor are all scheduled ahead of the secondaries on that processor. This is already the case for Processor 1; for Processor 2, this is done by moving T_1 to the end of the schedule and sliding the schedule for T_2, T_3, T_4, and T_5 up in time. See Figure 4.

6.1.2. General Task Periods. We now relax the condition that all task periods are equal and introduce the fault-tolerant assignment algorithm of Bertossi et al. [1999]. As

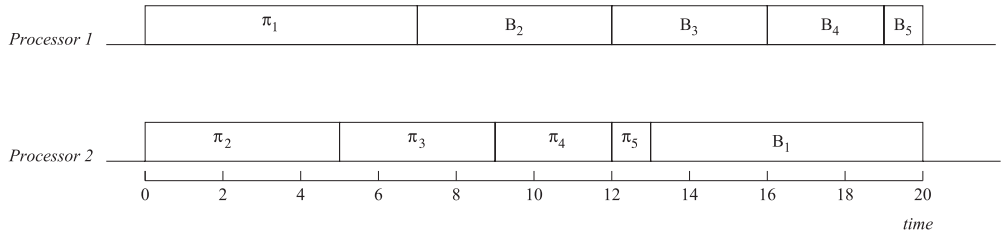

Fig. 4. Swap procedure for fault-tolerant scheduling.

before, tasks have primary and backup copies associated with them. These are defined as follows:

—*Active or Primary Tasks*: These are the tasks to be run in the absence of a failure.
—*Backup Tasks*: These will be activated upon a processor failure that affects their corresponding active task.
  —*Passive Backup Tasks*: These tasks can be scheduled so that they start executing only *after* the scheduled completion time of their primary. As a result, they can remain dormant until that primary has failed.
  —*Active Backup Tasks*: If the schedule is so tight that we cannot find time to schedule the tasks as passive backups, they must be scheduled as active backups. That is, they must start running even before we know that their primary has failed since there will be no time to execute them to completion if we wait until we know that the primary task has failed.

For the reasons mentioned earlier, we will restrict ourselves here to dealing with just one processor failure.[6] In such a case, once that failure has happened, we know that we will not be expected to deal with any other failure for the moment; hence, no backup (including active backups) whose primary is not on the failed processor need be executed. For this reason, to check the schedulability of active backups on any processor, we only need to ensure that

—For the no-failure case, all primaries and active backups on that processor can execute without any deadlines being missed.
—When a failure happens, since we are not required to be capable of responding to a second failure, we can immediately cancel the execution of all active backups, and ignore the schedulability needs of all passive backups, whose primaries are not on the failed processor.

By definition, a passive backup only executes if its primary has failed. This is only guaranteed to be known after the primary's scheduled completion time. Hence, a passive backup may only be scheduled in the interval between the scheduled finishing time of its primary and its deadline.

The fault-tolerant assignment algorithm is shown as Algorithm 2. The logic behind this algorithm is quite straightforward. We want to be able to sustain up to one processor failure at any one time. When assigning a primary task, we want to ensure that the processor to which it is assigned has the ability to run this task together with all active backups assigned to it as well as the passive backup sets that could possibly be

[6]This does not mean that we only allow for one failure for the lifetime of the computer system. Rather, we assume that the system will only have to deal with one failure at a time; that a second failure will not come about before the first has been dealt with and the system schedule has been stabilized with, if necessary, the reassignment of tasks. That is, the second failure will not happen until the "memory" of the first failure has faded away.

ALGORITHM 2: Fault-Tolerant Assignment Algorithm

```
Tasks are numbered in order of their RM priority;
P = (P_1, P_2, ..., P_m) is the set of processors;
b_i = worst-case execution time of backup B_i
f_i = scheduled finishing time of the primary of task T_i on its assigned processor
for (i = 1; i ≤ n; i++) do
    Assign the primary of task T_i using the first-fit procedure;
    The processor, P_k, to which it is assigned:
    (i) must be able to run this task as well as all other primary and active backup tasks
    already assigned to it, AND
    (ii) for (j = 1; j ≤ m; j++) do
        if (j ≠ k) then
            Must be able to activate on processor P_k all the backups of tasks assigned to P_j and
            still meet all task deadlines
        end
    end
    if (such a P_k cannot be found) then
        declare failure and quit;
    else
        Let f_i be the scheduled finishing time of T_i on P_k, relative to the release time of T_i
        if ((P_i − f_i) ≥ b_i) then
            Define B_i to be a passive backup
        else
            Define B_i to be an active backup
        end
    end
    if (B_i is an active backup) then
        Assign B_i using the first-fit procedure: it is forbidden to assign B_i to P_k. The processor,
        P_b, to which it is assigned:
        (i) must be able to run this task as well as all other primary and active backup tasks
        already assigned to it without missing any deadlines, and
        (ii) must be able to activate on processor P_b all of the backups of tasks assigned to P_k
        and still meet all task deadlines
        If such a P_b cannot be found, declare failure and quit;
    end
    if (B_i is a passive backup) then
        Assign B_i using the first-fit procedure: it is forbidden to assign B_i to P_k. The processor,
        P_b, to which it is assigned must satisfy the following property:
        It must be possible to activate on processor P_b all of the backups of tasks assigned to P_k
        as well as all of the primaries assigned to P_b and still meet all task deadlines
        If such a P_b cannot be found, declare failure and quit;
    end
end
```

activated. For each of the other $m-1$ processors in the system, there is a set of passive backup tasks that must be activated on the failure of that processor.

Each backup task is classified as active or passive depending on the scheduling of its corresponding primary. If the primary finishes so late that there is not enough time left for a backup task to be initiated and then complete before its deadline, then the backup must be initiated even before we know that the primary has failed. Such a backup is therefore designated as active. All others are designated as passive.

An active backup must run only as long as:

—All processors are up, or
—One processor is down, and that processor was assigned the primary copy corresponding to that active backup.

Hence, when checking the schedulability of an active backup, we only check that it is schedulable for one of these two cases. Similarly, when checking for the schedulability of passive backup β_i, we only need to check that the processor to which it is assigned can meet all deadlines of its primaries as well as of those backups whose primaries are on the same processor as that of π_i, the primary "partner" of β_i.

Let us now turn to how the schedulability tests are done. Note that this problem differs from the traditional RM model in which task deadlines equal their periods; in our case, passive backups may not start before the scheduled finishing time of their primaries. Thus, we cannot use the response time expression in Equation (1) to judge schedulability. However, the expression that we do end up with has the same logical foundation as (1).

We start by upper bounding the cumulative work, $\omega(\mathcal{T}, t)$, that arrives at some single processor, $\mathcal{P}$, by task set $\mathcal{T}$ during the interval $[0, t]$.[7] Let Π be the set of primaries, β_a the set of active backups, and β_p the set of passive backups assigned to processor $\mathcal{P}$. Denote the finishing time of the first iteration of π_i, the primary copy of T_i, by f_i. Its deadline (relative to when it was released) is equal to the period, P_i. Denote the WCET of the primary of task T_i by e_i and that of its backup by ξ_i. Then, we can write

$$\omega(\mathcal{T}, t) = \sum_{\{i|\pi_i \in \Pi\}} \lceil t/P_i \rceil e_i + \sum_{\{i|B_i \in \beta_a\}} \lceil t/P_i \rceil \xi_i + \sum_{\{i|B_i \in \beta_p\}} \xi_i \left(\mathbf{1}_{t \le f_i} + \{1 + \lceil (t - f_i)/P_i \rceil \mathbf{1}_{t > f_i}\} \right), \quad (9)$$

where $\mathbf{1}_C$ is the *indicator variable* defined as follows:

$$\mathbf{1}_C = \begin{cases} 1 & \text{if logical condition } C \text{ is true} \\ 0 & \text{otherwise.} \end{cases}$$

Let us justify Equation (9) by examining its RHS term by term. The first term on the RHS is the contribution of the primary tasks assigned to processor $\mathcal{P}$. The worst-case contribution of this term obviously occurs when the first iteration of all such tasks are released at time 0 (equivalently, we can say that the *phasings* of the primaries are all zero). The second term is the contribution of the active backups, assuming again that their phasings are zero. The third term accounts for the passive backups in set $\mathcal{T}$. Now, in the worst case, the phasing of the primaries associated with these backups can be such that the first iterations of these primaries are scheduled (on whichever processor to which they have been assigned) to complete at time 0; hence, we can assume that all passive backups corresponding to these first iterations become eligible to execute starting at time 0. If $t \le f_i$, then there can be only one such copy of each backup task that can be released over $[0, t]$; if $t > f_i$, this number is equal to $1 + \lceil (t - f_i)/P_i \rceil$.

In order for task $T_i \in \mathcal{T}$ to meet its deadline, it is sufficient to have $\omega(\mathcal{T}, t) \le 1$ for some $t \le P_i$. This is justified as follows. The worst-case response time for any task occurs when it is released at the same time as all of the higher-priority tasks (whether primary or backup) in the system. Thus, if we assume that the first iteration of all tasks are released at time 0, the worst-case response time of any task is that seen by its first iteration. If $\omega(\mathcal{T}, t) < t$ for some $t = \tau$, this means that the total work that came in over the interval $[0, t]$ was less than t; hence, the processor must have been idle over some part of that interval. Hence, all iterations that were released at time 0 must have been done by time τ (since otherwise the processor would not be idle). If $\tau \le P_i$, then that must mean that the first iteration of task T_i met its deadline. If $\omega(\mathcal{T}, t) = t$ for some $t = \tau$, then all of the tasks issued over $[0, \tau]$ are done by time τ, and if $\tau \le P_i$, the first iteration of T_i is done by its period.

[7]Do not confuse $\omega(\mathcal{T}, t)$ with $\mathcal{W}(\mathcal{T}, t)$ (encountered in Section 4.2). The former is the cumulative work that arrives during the interval $[0, t]$; the latter is the cumulative unfinished work *at* time t, regardless of when that unfinished work actually arrived at the system.

From this, we can now write sufficient conditions for schedulability for each of the following two cases. For the fault-free case, no passive backups are activated; the task set $\mathcal{T}_{fault-free}$ that processor $\mathcal{P}$ needs to be able to run is the set of primaries and the active backups that have been assigned to it. For this case, we must have

$$\forall(i \in \mathcal{T}_{fault-free}) \qquad \omega(\mathcal{T}_{fault-free}, t) \leq 1 \text{ for some } t \leq P_i. \tag{10}$$

The remaining case to consider is when some processor $\mathcal{P}_j$ ($\mathcal{P}_j \neq \mathcal{P}$) fails. This will require any passive backups (assigned to $\mathcal{P}$) of primaries scheduled on $\mathcal{P}_j$ to be capable of running. As mentioned earlier, once the system knows, at time t_j, that $\mathcal{P}_j$ has failed, it has to run backups associated with primaries assigned to $\mathcal{P}_j$ and not run any other backups (including active backups of other tasks). (Before the system knows that $\mathcal{P}_j$ has failed, it obviously has to keep running all active backups). Let us denote this set of tasks to be scheduled on $\mathcal{P}$ by $\mathcal{T}_{\mathcal{P}_j-fails}$: this set will not assume that any other backups execute after time t_j.

The condition to be satisfied to ensure schedulability on $\mathcal{P}$ is of the same form as Equation (10), except the task set is different:

$$\forall(i \in \mathcal{T}_{\mathcal{P}_j-fails}) \qquad \omega(\mathcal{T}_{P_j-fails}, t) \leq 1 \text{ for some } t \leq P_i. \tag{11}$$

One can think of several variations on this theme. We have already considered the possibility of assigning tasks in a different order to that of their RM priorities. This may result in more successful task assignments; however, the schedulability checks will be more time-consuming. Work along these lines is presented in Bertossi et al. [2006]. If tasks are assigned in descending order of RM priority, the assignment of a new task can never affect the schedulability of a previously assigned task (since all previously assigned tasks would be of higher priority than the one currently being assigned).

We have assumed in our discussion that the system is able to rapidly carry out the feasibility check of tasks as they arrive. A variation on this is where the incoming tasks are queued and there is a central scheduler that carries out the feasibility check every so often on the tasks that it finds in that queue [Manimaran and Murthy 1998]. When faced with multiple tasks to be scheduled, the scheduler can either assign them one by one (without any concern for how the scheduling of one task may affect the schedulability of others) or use some backtracking strategy. This strategy checks if scheduling one task creates constraints that make it impossible to schedule some other task; if so, it backtracks and tries to revise the schedule of the first task to relax such a constraint.

Another variation consists of the classification of backup tasks into passive and active. A passive task is constrained to execute within the interval between the finishing time of its primary and its deadline. If this interval is very short, then the available slack may not be sufficient to obtain a feasible schedule in many cases. Hence, if a passive backup cannot be feasibly scheduled, the algorithm might consider reclassifying it as active; this immediately relaxes the constraint that the backup be scheduled entirely after the scheduled finishing time of its primary. The algorithm can then make another attempt to assign it.

Further, note that we are assuming an active backup needs to continue executing through to completion. We do need to start such a backup before we know whether or not its primary has failed; however, if such information becomes available before its active backup has finished executing, the backup can safely be aborted, thus yielding additional slack in the schedule. We can also split what would have been an active backup into two parts: the part that is scheduled for before the finish time of its primary and the rest. The former can remain designated an active backup, whereas the latter can be made passive. An approach along these lines is outlined in Tsuchiya et al. [1995a, 1995b].

In the approach covered previously, any processor that does not carry the primary is a candidate for being allocated the backup. An alternative that has been suggested is that processors be organized into multiple groups with the proviso that both the primary and the backup must be allocated to processors within the same group [Al-Omari et al. 2001]. Groups may be formed either statically or dynamically; dynamic groups offer somewhat better guarantee rates.

Finally, there is the use of checkpointing [Ahn et al. 1997; Speirs and Barrett 1989]. A real-time task can checkpoint as it goes. Every so often, it records its state. If its processor should fail and the checkpoint is preserved by putting it in safe place, the backup can resume execution not from the beginning but from the latest checkpoint. Evaluation of such an approach should include the overhead for taking checkpoints.

6.2. Sporadic and Nonpreemptive Tasks

We now turn to scheduling a workload of sporadic and nonpreemptive tasks [Ghosh et al. 1997]. Here, tasks arrive dynamically, and the system has to build its schedule as it goes along. Since the workload is not known in advance, it cannot be checked for schedulability. Instead, when a task arrives (along with information about its WCET, and its deadline), the system determines if it has enough time to guarantee its execution. If so, the task is accepted; if not, the task is rejected and is no longer its responsibility.

As before, we use backup copies for fault tolerance: the system will not guarantee a task unless it can feasibly schedule both its primary and backup.

The algorithm starts by scheduling the primary copy of the incoming task. As before, one can use a first-fit approach to choose the processor that will be allocated as the primary. Alternatively, some other order may be chosen in which to try the processors. If we find a processor that can feasibly schedule this primary, we try to identify a processor that can feasibly schedule its backup according to Principles A1 to A3 listed in Section 1. We evaluate the goodness of the backup scheduling on each of the processors (except, obviously, the one that has been assigned the primary) according to a figure-of-merit function, which will be described next. The backup is assigned to whichever processor maximizes this figure of merit. If no such processor is found, we have to look for some other processor that can accept the primary (and satisfying A1 to A3) and repeat this process to find a home for its backup. If we are unable to find such a pair of processors for the primary and backup copies, the system rejects the task.

The figure of merit mentioned in the algorithm arises because there are two, often contradictory, impulses driving the scheduling of the backup. One is to schedule the backup as late as possible. This is important because the backup can be deallocated once the primary succeeds; having a long interval between when the primary succeeds and when the backup was scheduled to start gives the system a better opportunity to find something else to fill the slot vacated by the backup. The second impulse is to exploit the conditional transparency of backups and maximize the overlap between this backup and any others that may already be scheduled. Quite often, maximizing the overlap may lead us to schedule the backup for earlier than we otherwise might, whereas delaying the backup by as long as possible may reduce the overlap that we may otherwise be able to exploit.

This issue can be resolved by creating a figure of merit, composed of a weighted sum of the two quantities:

$$F = \alpha(t_{primaryfinish} - t_{backupstart}) + (1 - \alpha)\text{overlap\_length}, \tag{12}$$

where $\alpha \in [0, 1]$ is a tuning parameter. Users may experiment with various tuning parameters for the type of workloads that they anticipate in order to pick a suitable one.

6.3. Dynamic Programming Approach

A dynamic programming approach will now be presented [Krishna and Shin 1986]. This is an offline technique, in which multiple schedules are constructed for each processor. The first is the backup schedule, the second a primary schedule, and the rest are the contingency schedules to be invoked upon backup activation.

In the absence of faults, the primary schedule is executed. When a backup needs to be activated, this is done according to the backup schedule, with primary tasks having to give way to the backups. It is assumed that if iteration i of any given primary fails, it is only $i + 1$ and later iterations that have to be salvaged; hence, all backups can be passive. Either there is sufficient redundancy (e.g., TMR) to mask the results of this iteration's failure or the application dynamics allows the system to miss producing an occasional output. (This assumption can obviously be relaxed by categorizing backups as passive or active, along the lines encountered previously.) The primary schedule is constructed by invoking a dynamic programming algorithm, taking into account the *notification time* of each backup—that is, the time by which the system will be made aware of the need to execute that backup.

As a foundation upon which to construct fault-tolerant schedules, the dynamic programming approach can use any scheduling algorithm that consists of a task assignment part (Algorithm *Alloc*) and a uniprocessor scheduling part (Algorithm *Sched*) that is responsible for scheduling on each processor the tasks assigned to it by *Alloc*. Both *Alloc* and *Sched* may take both task deadlines and cost functions into account: they are heuristics that seek to reduce the total cost of the schedule while ensuring that all deadlines are met. These heuristics are not the focus of our discussion; rather, we are concerned with how they can be used as subroutines to obtain fault-tolerant schedules. That is, our focus is the fault-tolerant component that calls these subroutines.

The tasks assigned to a processor are numbered in order of their notification time. In what follows, we will concentrate on showing how to schedule tasks on some processor, $\mathcal{P}$: the same activity will be carried out independently for every other processor in the system. Denote by $P_{\zeta(1)}, P_{\zeta(2)}, \ldots, P_{\zeta(\theta)}$ the θ primaries and by $B_{\psi(1)}, B_{\psi(2)}, \ldots, B_{\psi(\gamma)}$ the γ backups assigned to processor $\mathcal{P}$, respectively, where $\zeta(i)$ and $\psi(i)$ denote the tasks whose primaries and backups are assigned to $\mathcal{P}$, respectively. The backup task cost functions are all zero: the focus of their scheduling is on meeting deadlines.

Once a processor has been assigned a set of primary tasks, the *holes* in its schedule can be identified. A hole is a interval of time in which it is impossible to schedule any of the primaries allocated to that processor due to their release times and deadlines. For example, if we have a processor assigned a single sporadic task whose release time is r and absolute deadline is d, then the following intervals are holes created by that task: $[0, r)$ and (d, ∞). The hole resulting from a set of tasks is the intersection of the holes induced by each of them. Holes are important in that backups can be scheduled on them without affecting the primary schedule. Assume that the set of primaries and backups assigned to a given processor are denoted by P and B, respectively.

We can improve the efficiency of the scheme by tuning Algorithm *Sched* to schedule backups in the holes to the extent possible and to allow it to overlap backups according to Principle A2. Let $\nu_{B_{\psi(1)}}, \nu_{B_{\psi(2)}}, \ldots, \nu_{B_{\psi(\gamma)}}$ be the notification times of backups respectively assigned to that processor. For convenience, set $\nu_{B_{\psi}(0)} = 0$ and $\nu_{B_{\psi(\gamma+1)}} = \infty$. The algorithm generates a sequence of $\gamma + 1$ primary schedules, $S_{p0}, S_{p1}, \ldots, S_{p\gamma}$, and then pieces them together.

The algorithm is shown as Algorithm 3 . In Step 1, the task allocation algorithm, *Alloc*, is called to assign tasks to processors. Such an assignment specifies the holes in the schedule of each processor.

ALGORITHM 3: Dynamic Programming Fault-Tolerant Scheduling Algorithm

1 Record the holes as defined by the task allocation;
2 Run *Sched* on the task set comprising all of the primaries and backups assigned to that processor. Note the position of the backups in schedule S_B;
3 Define a copy of this schedule: $S'_B = S_B$;
4 Set $j = 0$;
5 **while** *(B is nonempty)* **do**
  if *(j>0)* **then**
    set $B = B - \{B_{\psi(j)}\}$ and remove backup $B_{\psi(j)}$ from S'_B;
  end
  Generate a schedule beyond $t = \nu_{B_{\psi(j)}}$ for the primary workload not already scheduled up to that time. While doing so, assume that the processor is unavailable, for $t \geq \nu_{B_{\psi(j)}}$, to execute primaries in the time intervals occupied by S'_B. This is schedule S_{P_j};
  If any primary task cannot be scheduled to finish by its deadline, declare failure and exit;
  end
6 **if** *(all the primary workload has not yet been scheduled)* **then**
  schedule the remaining primary workload beyond $\nu_{B_{\psi(\gamma)}}$;
  end
7 **if** *(any primary task cannot be scheduled to finish before its deadline)* **then**
  declare failure and exit;
  end

In Step 2, the uniprocessor scheduling algorithm is executed on a task set comprising the primaries and all backups. The position of the backups in this schedule is recorded in a backup schedule, S_B. The position of the primaries in the schedule is recorded in schedule S_{p0}.

In Step 5 (some auxiliary steps are taken in Steps 3 and 4), we move to the first notification time, $\nu_{B_{\psi(1)}}$. By this time, the system will know whether or not the primary (on some other processor) associated with backup $B_{\psi(1)}$ has failed. If not, then there is no need to reserve space in the schedule for backup $B_{\psi(1)}$: any space reserved for $B_{\psi(1)}$ beyond time $\nu_{\psi(1)}$ in S_b can be released. We run algorithm *Sched* on a workload comprising the primary workload *not* scheduled over $[0, \nu_{B_{\psi(1)}}]$. Such scheduling is done based on the assumption that the processor is not available to the primaries over the periods occupied by backups in S_b. The schedule of tasks is obtained and recorded as S_{p1}.

We do the same thing with respect to each of the remaining notification times as we proceed through the Step 5 loop. After exiting this loop, in Step 6, we schedule any remaining primary workload and end the algorithm.

The output of the algorithm is a backup schedule and a set of primary schedules, $S_{p1}, S_{p2}, \ldots, S_{p\gamma}$.

During operation, we use these schedules as a template as follows in the absence of any faults: over the interval $[\nu_{B_{\psi(i)}}, \nu_{B_{\psi_{i+1}}})$, we execute primaries according to schedule S_{p_i}. If backup B_i is the first one to have to be activated, the machine will follow S_{p_i} throughout and never switch to $S_{p_{i+1}}$. This is illustrated in Figure 5.

This dynamic programming approach allows the user to introduce *cost functions* for the various tasks. That is, the aim is not just to meet all real-time deadlines; in addition, each task T_i has a cost function $F_i(R_i)$ of its response time, R_i, which represents the cost to the system of having such a response time. Such cost functions are relevant when the computer is in the control of some cyber-physical control system. In such a case, the computer is in the feedback loop of the controlled process, and any incremental delay of the controller potentially contributes to a degradation in the quality of control provided. Such degradation is captured in the form of cost functions. It is beyond the scope of this article to describe how these functions may be obtained: see Krishna and

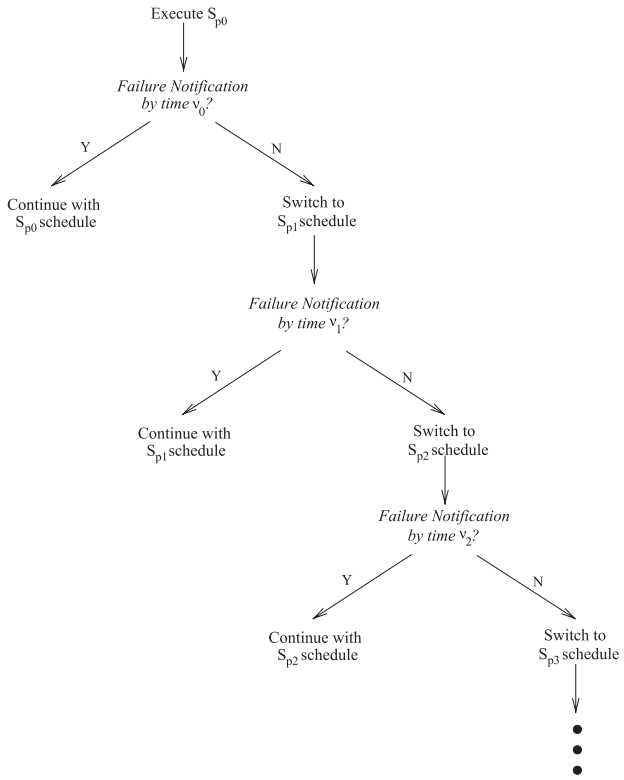

Fig. 5. Example of schedules followed by dynamic programming algorithm.

Table III. Parameters for Example

Task	Release	Exec Time	Abs Deadline	Cost Function	ν_i
T_1	1	15	130	R_1^4	N/A
T_2	1	15	110	R_2^3	N/A
T_3	11	15	90	R_3^2	N/A
T_4	1	15	60	R_4	N/A
T_5	0	15	22	N/A	0
T_6	24	15	39	N/A	20
T_7	40	15	55	N/A	40
T_8	90	15	130	N/A	90

R_i^x is the response time of task T_i raised to power x.

Shin [1987], Shin and Cui [1995], and Shin and Krishna [1987] for ways by which the control dynamics of the controlled process can translate into cost functions. Such cost functions can be used to calculate the total cost of a schedule: this is given by $\sum_i F_i(R_i)$.

Consider now the following example. Assume that primaries of sporadic tasks T_1, T_2, T_3, and T_4, as well as backups of sporadic tasks T_5, T_6, T_7, and T_8, have been assigned to some processor. Assume also (just for this example) that the execution time of the primary and backup copies of a task are the same and that the backups are not allowed to overlap since their primaries have been assigned to the same processor. We now show how a schedule for that processor is constructed.

The parameters of each of the tasks and their cost functions are shown in Table III. Figure 6 (based on Krishna and Shin [1986]) shows the development of the

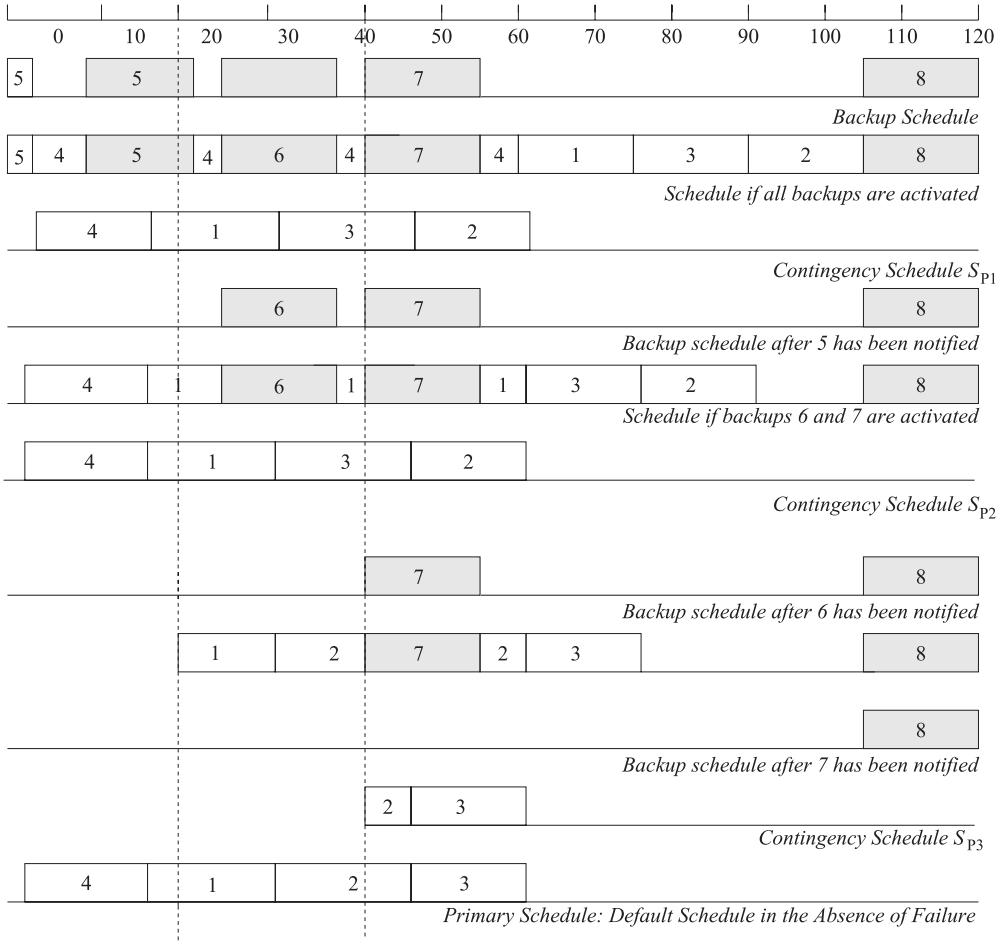

Fig. 6. Dynamic programming approach: Example.

fault-tolerant cost-sensitive schedule. The cost in this example is a polynomial function of response time and is *not* used directly by the fault-tolerant scheduling algorithm. Rather, it is taken into account by the underlying non–fault-tolerant scheduling algorithm that is called by the fault-tolerant algorithm. The non–fault-tolerant heuristic obtains a schedule with a view to reducing the total cost of the schedule. We do not describe either of the allocation or non–fault-tolerant cost-based scheduling heuristic since our focus is on the fault-tolerant algorithm; indeed, this approach is designed to work with any suitable allocation or non–fault-tolerant cost-based scheduling algorithm.

We first set up the schedule of backups: for this, schedule the entire primary and backup tasks and record the position of the backups in this schedule. This is the backup schedule shown at the top of Figure 6. If all backups were to be activated, then these, together with the primaries allocated to the processor, would be scheduled (second schedule of Figure 6). Hence, the schedule that we would want to follow for the primaries, before we are notified that one or more of the backups will not need to run, is given by Contingency Schedule S_{p_1}.

At time 0, we can be notified that backup 0 will not need to be activated. Accordingly, backup copy 5 can be removed from the schedule (third schedule of Figure 6). If the

backups in the schedule are to be activated along with the primaries allocated to this processor, then the resulting schedule is shown as the fourth schedule. Hence, after time 0, contingency schedule S_{P_2} consists of primaries 4, 1, 3, and 2 in that order.

At time 20, we are notified that backup 6 will not need to be activated. There is nothing that we can do to modify the schedule prior to this notification point; however, we can exploit the absence of 6 from the schedule by changing the order of primary execution to allow primary 2 to be completed sooner (in order to improve the quality of the schedule as specified by the cost functions).

Finally, at time 40, we are notified that backup 7 will not activate. We proceed as done previously.

The overall default schedule is shown as the final schedule in Figure 6. This is the primary schedule that will run in the usual case of no failure. If at any time the failure of a primary elsewhere in the system requires a backup on this processor to be activated, we will simply switch to the appropriate contingency schedule.

7. VOLTAGE SCALING

Circuit delays in processors are a strong function of their supply voltage:

$$\delta(v) = \frac{C_L v T^{\mu}(v)}{K(v - v_T)^{\alpha}}, \tag{13}$$

where v is the supply voltage, T is the absolute temperature, K is a constant, v_T is the threshold voltage, C_L is the effective switching capacitance, and α and μ are constants (typically around 1.2) [Liao et al. 2005]. Hence, by reducing the supply voltage, the speed of execution can be reduced and vice versa. This slowdown factor can be written as

$$slow(v) = \frac{v}{v_h}\left(\frac{v_h - v_T}{v - v_T}\right)^{\alpha}\left(\frac{T(v)}{T(v_h)}\right)^{\mu}. \tag{14}$$

Voltage scaling has become important, because in recent years, the energy consumption of processors has emerged as a major concern. Energy consumption causes heating that can increase the failure rate and strains energy resources such as batteries. Running a processor at lower voltage dramatically reduces its energy consumption. In real-time systems, we can scale voltage to ensure that tasks are completed just before their deadlines. Time in the schedule that is released when tasks complete ahead of their WCETs can be reclaimed by running the processor at a lower voltage. There is a substantial literature in this area: see, for example, Unsal and Koren [2003] for a review.

By running the processor at a slightly greater speed than is required to just meet task deadlines, we can build up a reserve of time that can then be used to execute tasks that have been affected by failure. This time reserve can be exploited by the scheduling algorithms that have been described earlier. An approach is presented along the lines of Santos et al. [2009].

For example, consider using this approach to deal with transient faults in a system that schedules tasks according to the RM algorithm. Recall from Section 4 that the response time of a task is obtained by adding its own execution time to the amount of time that the task has to wait for higher-priority tasks to finish executing on the processor.

Consider Equation (1), slightly modified by the addition of a t_i term to the right-hand side as follows:

$$R_i = \begin{cases} e_i + t_i & \text{if } i = 1 \\ e_i + t_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{P_j} \right\rceil e_j & \text{otherwise.} \end{cases} \tag{15}$$

If we satisfy the condition that $R_i \leq P_i$ for all tasks T_i, then it means that each task has enough time to be executed, to spend time waiting for higher-priority tasks to execute, *and* a reserve time of t_i, which can be used in recovering from failure. In other words, t_i is the time by which any iteration of periodic task T_i can be delayed in its execution as a result of having to execute backup copies or take other recovery action. (An alternative formulation is to use Equation (3) in Section 4.1, in which case each task is allowed to suffer one transient failure and still meet deadlines).

Since all tasks must meet their deadlines, it follows that $t_{min} = \min_i\{t_i\}$ is the system-wide time redundancy available. This time can be used to execute backup copies of failing tasks or to retry failed versions. Note that since the execution of backup copies is a rare occurrence, we can assume that this will be done at the highest supply voltage possible—that is, at the highest speed setting of processor.

e_i can be controlled by suitably selecting the supply voltage (within limits): we therefore have a tradeoff between the energy consumption and the time redundancy available for fault tolerance.

Consider the following simple example to illustrate this. Suppose that we have a two-task system, with the execution time and period being $(e_1, P_1) = (1, 5)$ and $(e_2, P_2) = (3, 12)$, where the execution times are specified at the highest speed of the processor. Further assume that we plan to run both tasks at the same slowdown factor, s. (Note that this is just to simplify our discussion: the slowdown factor cannot only be different for different tasks, it can also change during the course of execution of a task. Furthermore, s may in practical instances only be able to take one of a set of discrete values, since the supply voltage may not be continuously tunable. None of these facts changes the basis for this approach.)

The schedulability equations now become

$$s + t_{\min} \leq 5; \tag{16}$$

$$3s + 3 + t_{\min} \leq 12. \tag{17}$$

From this, we have $t_{\min} = \min\{5-s, 12-4s\}$. By adjusting the slowdown, we can control the amount of time redundancy available and obtain a trade-off between the available fault tolerance and the energy consumption.

Many variations on this basic theme are possible. For example, as mentioned previously, only discrete frequencies may be available. Furthermore, there can be an overhead every time the voltage and clock frequency are adjusted. In addition, the failure rate is linked to the voltage level, which may affect the voltage level chosen [Zhu et al. 2004; Pop et al. 2007].

8. DISCUSSION

Real-time systems are becoming ever more widely used in life-critical applications, and the need for fault-tolerant scheduling can only grow in the years ahead.

In this article, some techniques for fault-tolerant scheduling in homogeneous systems have been outlined. All of these algorithms have one objective: to guarantee that, despite a certain maximum number of faults, the tasks will have enough hardware and time redundancy to meet their deadlines. Many interesting challenges in this field remain.

While the focus has been on homogeneous systems, these techniques can also be used, with some modifications, in heterogeneous systems as well. In a heterogeneous system, tasks can have different execution times on different processors. Some processors may have specialized hardware (like fast array-handling capabilities) that render them more suited to some applications than to others. One can define a matrix of execution times of tasks to processors and make task assignment decisions (for both primary and backup copies) based on this information.

If nodes are heterogeneous, they are also likely to differ in their failure characteristics, both in the rate with which they fail and also the temporal characteristics of their transient failure. Algorithms to take such characteristics explicitly into account while assigning primary and backup copies are worth developing.

Another area that deserves greater attention is the impact of the task assignment and schedule itself on reliability. As the processor workload increases, its static and dynamic power consumption both increase, thereby increasing its temperature. With an increase in operating temperature comes an increase in the failure rate. Such an effect is likely to be exacerbated by the move into ever finer feature sizes. Thermal-aware fault-tolerant scheduling has not yet attracted the attention that it deserves.

Fault-tolerant scheduling algorithms have concentrated on the processor, ignoring other aspects of the system. This is especially the case with voltage scaling. However, the processor is but one part of a complex that also includes memory, the interconnection network, and I/O devices. Taking account of these dimensions can alter the framework in which the scheduler operates. For example, the size of the memory footprint of each task can constrain the task assignment procedure as well as limit the range of options available to energy management algorithms (which have to reduce the sum of the static and dynamic energy consumption).

To summarize, although there is already a significant body of work in the area of fault-tolerant scheduling of real-time systems, many interesting problems remain.

APPENDIX: TABLE OF SYMBOLS AND ACRONYMS

$\alpha(t)$	Work brought in by a task at time t
B_i	Backup copy of T_i
D_i	Absolute deadline of T_i
$\delta^{\mathcal{W}}(\mathcal{T}, t)$	Max additional unfinished work at time t due to task set $\mathcal{T}$ being struck by $\mathcal{W}$ failures
e_i	WCET of T_i
$\zeta(i)$	ith task whose primary has been assigned to processor P
f_i	Finishing time of T_i
LCM	Least Common Multiple
$\nu_{B_{\psi(i)}}$	Notification time for backup of $\psi(i)$ on processor P
ϕ_i	Number of faults during execution of T_i
π_i	Primary copy of T_i
PB	Primary/Backup
P_i	Period of task i (if periodic)
r_i	Release time of T_i
R_i	Response time of T_i
S_{p_i}	ith primary schedule
T_i	Task i
slack(a,b)	Slack in the schedule over $[a, b]$
t_{rec}	Transient fault recovery time
TMR	Triple Modular Redundancy
τ_F	Minimum interval between successive transient failures
$\mathcal{W}(\mathcal{T}, t)$	Unfinished work at time t due to task set $\mathcal{T}$
WCET	Worst-case execution time
ξ_i	WCET of T_i backup
$\psi(i)$	ith task whose backup has been assigned to processor P

ACKNOWLEDGMENTS

The author would like to thank the referees for their careful reading of the manuscript and their useful suggestions.

REFERENCES

K. Ahn, J. Kim, and S. Hong. 1997. Fault-Tolerant Real-Time Scheduling Using Passive Replicas. In *Proceedings of the Pacific Rim International Symposium on Fault-Tolerance.* 98–103.

R. Al-Omari, G. Manimaran, and A. K. Somani. 2001. An Efficient Backup-Overloading for Fault-Tolerant Scheduling of Real-Time Tasks. In *Proceedings of the International Parallel Processing Symposium.* 1291–1295.

A. A. Bertossi, L. V. Mancini, and A. Menapace. 2006. Scheduling Hard-Real-Time Tasks with Backup Phasing Delay. In *Proceedings of the IEEE Symposium on Distributed Simulation and Real-Time Applications (DS-RT).*

A. A. Bertossi, L. V. Mancini, and F. Rossini. 1999. Fault-Tolerant Rate-Monotonic First-Fit Scheduling in Hard-Real-Time Systems. *IEEE Transactions on Parallel and Distributed Systems* 10, 9 (September 1999), 934–945.

A. Burns, R. Davis, and S. Punnekat. 1996. Feasibility Analysis of Fault-Tolerant Real-Time Task Sets. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems (EUROWRTS).* 29–33.

M. Caccamo and M. Buttazzo. 1998. Optimal Scheduling for Fault-Tolerant and Firm Real-Time Systems. In *Proceedings of the IEEE Conference on Real-Time Computing Systems and Applications (RTCSA).*

A. Cheng. 2002. *Real-Time Systems: Scheduling, Analysis and Verification*. Wiley-Interscience.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2004. *Introduction to Algorithms*. MIT Press.

M. S. Garey and D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman.

S. Ghosh, R. Melhem, and D. Mosse. 1972. Fault-Tolerance Through Scheduling of Aperiodic Tasks in Hard Real-Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems* 8, 3 (March 1997), 272–283.

O. Gonzalez, H. Shrikumar, J. A. Stankovic, and K. Ramamritham. 1997. Adaptive Fault Tolerance and Graceful Degradation Under Dynamic Hard Real-time Scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium*. 79–89.

R. L. Graham. 1969. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal of Applied Mathematics* 17, 2 (March 1969), 416–429.

C.-C. Han, K. G. Shin, and J. Wu. 2003. A Fault-Tolerant Scheduling Algorithm for Real-Time Periodic Tasks with Possible Software Faults. *IEEE Transactions on Computers* 52, 3 (March 2003), 363–372.

F. S. Hillier and G. J. Lieberman. 2001. *Introduction to Operations Research*. McGraw-Hill.

B. Johnson. 1989. *The Design and Analysis of Fault-Tolerant Digital Systems*. Addison-Wesley.

M. Joseph and P. Pandya. 1986. Finding Response Times in a Real-Time System. *Computer Journal* 29, 5 (October 1986), 390–395.

H. Kopetz. 1997. *Real-Time Systems*. Kluwer Academic Publishers.

H. Kopetz and G. Bauer. 2003. The Time-Triggered Architecture. *Proceedings of the IEEE* 91, 1 (January 2003), 112–126.

H. Kopetz and D. Millinger. 1999. The Transparent Implementation of Fault Tolerance in the Time-Triggered Architecture. In *Dependable Computing for Critical Applications*, A. Avizienis, H. Kopetz, and J. C. Laprie (Eds.), 192–205.

H. Kopetz and W. Ochsenreiter. 1987. Clock Synchronization in Distributed Real-Time Systems. *IEEE Transactions on Computers* C-36, 933–940.

I. Koren and C. M. Krishna. 2007. *Fault-Tolerant Systems*. Morgan Kaufmann.

C. M. Krishna and K. G. Shin. 1986. Scheduling Tasks with a Quick Recovery from Failure. *IEEE Transactions on Computers* C-35, 5 (May 1986), 448–455.

C. M. Krishna and K. G. Shin. 1987. Performance Measures for Control Computers. *IEEE Transactions on Automatic Control* AC-32, 6, 467–473.

C. M. Krishna and K. G. Shin. 1997. *Real-Time Systems*. McGraw-Hill.

J. P. Lehoczky, L. Sha, and J. K. Strosnider. 1987. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. In *Proceedings of the IEEE Real-Time Systems Symposium.* 261–270.

W. Liao, L. He, and K. M. Lepak. 2005. Temperature and Supply Voltage Aware Performance and Power Modeling at Microarchitecture Level. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24, 7 (July 2005), 1042–1053.

F. Liberato, R. Melhem, and D. Mosse. 2000. Tolerance to Multiple Transient Faults in Hard Real-Time Systems. *IEEE Transactions on Computers* 49, 9 (September 2000), 906–914.

A. L. Liestman and R. H. Campbell. 1986. A Fault-Tolerant Scheduling Problem. *IEEE Transactions on Software Engineering* 12, 11 (November 1986), 1089–1095.

C. L. Liu and J. W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM* 20, 1 (January 1973), 40–61.

J. W. S. Liu. 2000. *Real-Time Systems*. Wiley.

C. Siva Ram Murthy and G. Manimaran. 2001. *Resource Management in Real-Time Systems and Networks*. MIT Press.

G. Manimaran and C. Siva Ram Murthy. 1998. A Fault-Tolerant Dynamic Scheduling Algorithm for Multiprocessor Real-Time Systems and Its Analysis. *IEEE Transactions on Parallel and Distributed Processing Systems* 9, 11 (November 1998), 1137–1152.

M. Naedele. 1999. Fault-Tolerant Real-Time Scheduling Under Real-Time Constraints. In *Proceedings of the International Workshop on Real-Time Computing Systems and Applications (RTCSA)*. 392–395.

N. Nissanke. 1997. *Realtime Systems*. Prentice Hall.

Y. Oh and S. H. Son. 1992. An Algorithm for Real-Time Fault-Tolerant Scheduling in Multiprocessor Systems. In *Proceedings of the Euromicro Workshop on Real-Time Systems*. 190–195.

M. Pandya and M. Malek. 1998. Minimum Achievable Utilization for Fault-Tolerant Processing of Periodic Tasks. *IEEE Transactions on Computers* 47, 10 (October 1998), 1102–1112.

E. L. Petersen. 1997. Predictions and Observations of SEU Rates in Space. *IEEE Transactions on Nuclear Science* 44, 6 (December 1997), 2174–2187.

S. Poledna, A. Burns, A. Wellings, and P. Barretta. 2000. Replica Determinism and Flexible Scheduling in Hard Real-Time Dependable Systems. *IEEE Transactions on Computers* 49, 2 (February 2000), 100–111.

P. Pop, K. H. Poulsen, V. Izosimov, and P. Eles. 2007. Scheduling and Voltage Scaling for Energy/Reliability Trade-Offs in Fault-Tolerant Time-Triggered Embedded Systems. *CODES+ISSS*. 233–238.

D. K. Pradhan. 1996. *Fault-Tolerant Computer System Design*. Prentice Hall.

K. Ramamritham, J. A. Stankovic, and P.-F. Shiah. 1994. Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. *Proceedings of the IEEE* 82, 1 (January 1994), 55–67.

R. M. Santos, J. Santos, and J. D. Orozco. 2009. Power Saving and Fault-Tolerance in Real-Time Critical Embedded Systems. *Journal of Systems Architecture* 55, 90–101.

K. G. Shin and X. Cui. 1995. Computing Time Delay and Its Effects on Real-Time Control Systems. *IEEE Transactions on Control Systems Technology* 3, 2 (June 1995), 218–224.

K. G. Shin and C. M. Krishna. 1987. Performance Measures for Control Computers. *IEEE Transactions on Automatic Control* AC-32, 6 (June 1987), 467–473.

D. Siewiorek and R. Swarz. 1999. *Reliable Computer Systems: Design and Evaluation*. A. K. Peters.

N. Speirs and P. Barrett. 1989. Using Passive Replicates in Delta-4 to Provide Dependable Distributed Computing. In *Proceedings of the Fault-Tolerant Computing Symposium (FTCS-19)*. 184–190.

J. A. Stankovic and K. Ramamritham. 1989. The Spring Kernel: A New Paradigm for Real-Time Operating Systems. *ACM Operating Systems Review* 23, 3 (July 1989), 54–71.

T. Tsuchiya, Y. Kakuda, and T. Kikuno. 1995a. Fault-Tolerant Scheduling Algorithm for Distributed Real-Time Systems. 1995. In *Proceedings of the 3rd Workshop on Parallel and Distributed Real-Time Systems*. 99–103.

T. Tsuchiya, Y. Kakuda, and T. Kikuno. 1995b. A New Fault-Tolerant Scheduling Technique for Real-Time Multiprocessor Systems. In *Proceedings of the International Workshop on Real-Time Computing Systems and Applications (RTCSA)*. 197–202.

O. S. Unsal and I. Koren. 2003. System-Level Power-Aware Design Techniques in Real-Time Systems. *Proceedings of the IEEE* 91, 7 (July 2003), 1055–1069.

D. Zhu, R. Melhem, and D. Mosse. 2004. The Effects of Energy Management on Reliability in Real-Time Embedded Systems. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. 35–40.

Received May 2008; revised September 2012; accepted September 2013