*Proc. of the Fifth European Dependable Computing Conference, Apr. 2005*

# Energy Efficient Configuration for QoS in Reliable Parallel Servers

Dakai Zhu[1]*, Rami Melhem[2], and Daniel Mossé[2]

[1] University of Texas at San Antonio, San Antonio, TX, 78249, USA,
[2] University of Pittsburgh, Pittsburgh, PA, 15260, USA,

**Abstract.** Redundancy is the traditional technique used to increase system reliability. With modern technology, in addition to being used as temporal redundancy, slack time can also be used by energy management schemes to scale down system processing speed and supply voltage to save energy. In this paper, we consider a system that consists of multiple servers for providing reliable service. Assuming that servers have self-detection mechanisms to detect faults, we first propose an efficient parallel recovery scheme that processes service requests in parallel to increase the number of faults that can be tolerated and thus the system reliability. Then, for a given request arrival rate, we explore the optimal number of active severs needed for minimizing system energy consumption while achieving $k$-fault tolerance or for maximizing the number of faults to be tolerated with limited energy budget. Analytical results are presented to show the trade-off between the energy savings and the number of faults being tolerated.

## 1 Introduction

The performance of modern computing systems has increased at the expense of drastically increased power consumption. For large systems that consist of multiple processing units (e.g., complex satellite and surveillance systems, data warehouses or web server farms), the increased power consumption causes heat dissipation problems and requires more expensive packaging and cooling technologies. If the generated heat cannot be properly removed, it will increase the temperature and thus decrease system reliability.

Traditionally, energy management has focused on portable and handheld devices that have limited energy budget to extend their operation time. However, the energy management for servers in data centers, where heat generated and cooling costs are big problems, have caught people's attention recently. In [1], Bohrer *et al.* presented a case of managing power consumption in web servers. Elnozahy *et al.* evaluated a few policies that combine dynamic voltage scaling (DVS) [24, 25] on individual server and turning on/off servers for cluster-wide power management in server farms [5, 14]. Sharma *et al.* investigated adaptive algorithms for voltage scaling in QoS-enabled web servers to minimize energy consumption subject to service delay constraints [19]. Although fault tolerance

* Work was done while the author was a Ph.D student in University of Pittsburgh.

through redundancy [11, 13, 16, 20] has also been well studied, there is relatively less work addressing the problem of combining fault tolerance and energy management [26, 27]. For systems where both lower levels of energy consumption and higher levels of reliability are important, managing the system reliability and energy consumption together is desired.

Modular redundancy and temporal redundancy have been explored for fault tolerance. Modular redundancy detects and/or masks fault(s) by executing an application on several processing units in parallel and temporal redundancy can be used to re-execute an application to increase system reliability [16]. To efficiently use temporal redundancy, checkpointing techniques have been proposed by inserting checkpoints within an application and rolling back to the last checkpoint when there is a fault [11, 13]. In addition to being used for temporal redundancy, slack time can also be used by DVS techniques to scale down system processing speed and supply voltage to save energy [24, 25]. Therefore, there is an interesting trade-off between system reliability and energy savings.

For independent periodic tasks, using the primary/backup model, Unsal *et al.* proposed an energy-aware software-based fault tolerance scheme which postpones as much as possible the execution of backup tasks to minimize the overlap of primary and backup execution and thus to minimize energy consumption [23]. For Duplex systems, the optimal number of checkpoints, uniformly or non-uniformly distributed, to achieve minimum energy consumption was explored in [15]. Elnozahy *et al.* proposed an *Optimistic-TMR* (OTMR) scheme to reduce the energy consumption for traditional TMR systems by allowing one processing unit to slow down provided that it can catch up and finish the computation before the deadline if there is a fault [6]. The optimal frequency setting for OTMR is further explored in [28]. Combined with voltage scaling techniques, an adaptive checkpointing scheme was proposed to tolerate a fixed number of transient faults and save energy for serial applications [26]. The work was further extended to periodic real-time tasks in [27].

In this paper, we consider the execution of event-driven applications on parallel servers. Assuming that self-detection mechanisms are deployed in servers to detect faults, for a given system load (i.e., the number of requests in a fixed interval), we explore the optimal number of active servers needed for minimizing system energy consumption while achieving $k$-fault tolerance. We also explore maximizing the number of faults to be tolerated with limited energy budget. An efficient parallel recovery scheme is proposed, which processes service requests in parallel to increase the number of faults that can be tolerated within the interval considered and thus system *performability* (defined as the probability of finishing an application correctly within its deadline in the presence of faults [10]).

This paper is organized as follows: the energy model and the application and problem description are presented in Section 2. The recovery schemes are discussed in Section 3. Section 4 presents two schemes to find the optimal number of active servers needed for energy minimization and performability maximization, respectively. The analysis results are presented and discussed in Section 5 and Section 6 concludes the paper.

## 2 Models and Problem Description

### 2.1 Power Model

The power in a server is mainly consumed by its processor, memory and the underlying circuits. For CMOS based variable frequency processors, power consumption is dominated by dynamic power dissipation, which is cubicly related to the supply voltage and the processing speed [2]. As for memory, it can be put into different power states with different response times [12]. For servers that employ variable frequency processors [7, 8] and low power memory [17], the power consumption can be adjusted to satisfy different performance requirements. Although dynamic power dominates in most components, the static leakage power increases much faster than dynamic power with technology advancements and thus cannot be ignored [21, 22].

To incorporate all power consuming components in a server and keep the power model simple, we assume that a server has three different states: *active, sleep* and *off*. The system is in the *active state* when it is serving a request. All static power is consumed in the active state. However, a request may be processed at different frequencies and consume different dynamic power. The *sleep state* is a power saving state that removes all dynamic power and most of the static power. Servers in sleep state can react quickly (e.g., in a few cycles) to new requests and the time to transit from sleep state to active state is assumed to be negligible. A server is assumed to consume no power in the *off state*.

Considering the almost linear relation between processing frequency and supply voltage [2], voltage scaling techniques reduce the supply voltage for lower frequencies [24, 25]. In what follows, we use *frequency scaling* to stand for changing both processing frequency and supply voltage. Thus, the power consumption of a server at processing frequency $f$ can be modeled as [28]:

$$P(f) = P_s + \hbar(P_{ind} + P_d) = P_s + \hbar(P_{ind} + C_{ef} f^m) \tag{1}$$

where $P_s$ is the sleep power; $P_{ind}$ and $P_d$ are the active powers that are frequency-independent and frequency-dependent, respectively. $\hbar$ equals 1 if a server is active and 0 otherwise. $C_{ef}$ and $m$ are system dependent constants. The maximum frequency-dependent active power corresponds to the maximum processing frequency $f_{max}$ and is given by $P_d^{max} = C_{ef} f_{max}^m$. For convenience, the values of $P_s$ and $P_{ind}$ are assumed to be $\alpha P_d^{max}$ and $\beta P_d^{max}$, respectively. Moreover, we assume that continuous frequency is used. For systems that have discrete frequencies, two adjacent frequencies can be used to emulate any frequency as discussed in [9].

Notice that, less frequency-dependent energy is consumed at lower frequencies; however, it takes more time to process a request and thus more sleep and frequency-independent energy is consumed. Therefore, due to the sleep power and frequency-independent active power, there is an energy efficient processing frequency at which the energy consumption to process a request is minimized [28]. Since the overhead of turning on/off a server is large [1], we assume in this paper that the deployed servers are always on and the sleep power $P_s$ is not manageable (i.e., always consumed). Thus, the *energy efficient frequency* can be easily found as:

$$f_{ee} = \sqrt[m]{\frac{\beta}{m-1}} \cdot f_{max} \tag{2}$$

If $f_{ee} > f_{max}$, that is, $\beta > m - 1$, all requests should be processed at the maximum frequency $f_{max}$ to minimize their energy consumption and no frequency scaling is necessary. Notice that $f_{ee}$ is solely determined by the system's power characteristics and is independent of requests to be processed. Given that $f_{low}$ is the lowest supported processing frequency, we define the minimum energy efficient frequency as $f_{min} = max\{f_{low}, f_{ee}\}$. That is, we may be forced to run at a frequency higher than $f_{ee}$ to meet an application's deadline or to comply with the lowest frequency limitation. However, for energy efficiency, we should never run at a frequency below $f_{ee}$. For simplicity, we assume that $f_{ee} \geq f_{low}$, that is, $f_{ee} = \kappa f_{max}$, where $\kappa = \frac{f_{ee}}{f_{max}}$.

### 2.2 Application Model and Problem Description

In general, the system load of an event-driven application is specified by service request[3] arrival rates. That is, the number of requests within a given interval. Although the service time for each individual request may vary, we can employ the law of large numbers and use a mean service time for all requests, which can be justified in the case of high performance servers where the number of requests is large and each individual request has relatively short service time [19]. That is, we assume that requests have the same size and need $C$ cycles to be processed. For the case of large variations in request size, checkpointing techniques can be employed to break requests into smaller sections of the same size [15].

Given that we are considering variable frequency processors, the number of cycles needed to process a request may also depend on the processing frequency [18]. However, with a reasonable size cache, $C$ has been shown to have very small variations with different frequencies [15]. For simplicity, we assume that $C$ is a constant[4] and is the mean number of cycles needed to process a request at the maximum frequency $f_{max}$. Without loss of generality, the service time needed for each request at $f_{max}$ is assumed to be $c = \frac{C}{f_{max}} = 1$ time unit. Moreover, to ensure responsiveness, we consider time intervals of length equal to $D$ time units. All requests arriving in an interval will be processed during the next interval. That is, the response time for each request is no more than $2D$.

During the processing of a request, a fault may occur. To simplify the discussion, we limit our analysis to the case where faults are detected through a *self-detection mechanism* on each server [16]. Since *transient* and *intermittent* faults occur much more frequently than *permanent* faults [3], in this paper, we

[3] Without causing confusion, we use events and service requests interchangeably.

[4] Notice that, this is a conservative model. With fixed memory cycle time, the number of CPU cycles needed to execute a task actually decreases with reduced frequencies and the execution time will be less than the modeled time.

focus on transient and intermittent faults and assume that such faults can be recovered by re-processing the faulty request.

For a system that consists of $M$ servers, due to energy consideration, suppose that $p$ ($p \leq M$) servers are used to implement a *k-fault tolerant system*, which is defined as a system that can tolerate $k$ faults within any interval $D$ under all circumstances. Let $w$ be the number of requests arriving within an interval $D$. Recall that the processing of one request needs one time unit. Hence, $n = \lceil \frac{w}{p} \rceil$ time units are needed to process all the requests. Define a *section* as the execution of one request on one server. If faults occur during the processing of one request, the request becomes faulty and a *recovery section* of one time unit is needed to re-process the faulty request. To tolerate $k$ faults in the worst case, a number of time units, $b$, have to be reserved as *backup slots*, where each backup slot has $p$ parallel recovery sections. For a faulty request, the processing during a recovery section may also encounter faults. If all the recovery sections that process a given faulty request fail, then we say that there is a *recovery failure*.

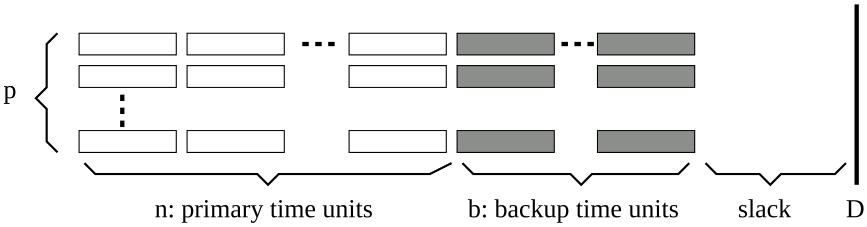


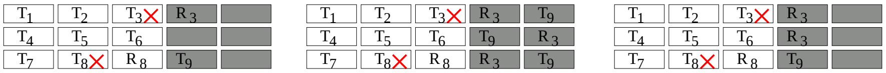
**Fig. 1.** To achieve a $k$-fault tolerant system, $p$ servers are used to process $w$ requests within a time interval of $D$. Here, $b$ time units are reserved as backup slots.

The schedule for processing all requests within the interval of $D$ is shown in Figure 1. In the figure, each white rectangle represents a section that is used to process one request on a server and the shadowed rectangles represent the recovery sections reserved for processing the faulty requests. For ease of presentation, the first $n$ time units are referred to as *primary time units* and all white rectangles are referred as *primary execution.* After scheduling the primary time units and backup slots, the amount of slack left is $D - (n + b)$, which can be used to scale down the processing frequency of servers and save energy.

For a given request arrival rate and a fixed time interval in an event-driven system that consists of $M$ servers, we focus on exploring the optimal number of active servers needed to minimize energy consumption while achieving a $k$-fault tolerant system or to maximize the number of faults that can be tolerated with limited energy budget.

## 3 Recovery with Parallel Backup Slots

In this section, we calculate the worst case maximum number of faults that can be tolerated during the processing of $w$ requests by $p$ servers with $b$ backup slots. The addition of one more fault could cause an additional faulty request that can not be recovered and thus leads to a system failure. As a first step, we assume that the number of requests $w$ is a multiple of $p$ (i.e., $w = n \cdot p$, $n \geq 1$). The case of $w$ being not a multiple of $p$ will be discussed in Section 3.4. For different strategies of using backup slots, we consider three recovery schemes: *restricted serial recovery*, *parallel recovery* and *adaptive parallel recovery*.



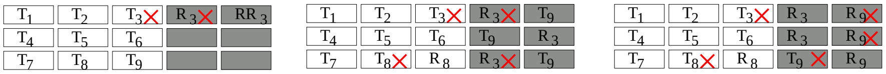
a. Restricted serial recovery b. Parallel recovery c. Adaptive parallel recovery

**Fig. 2.** Different recovery schemes.

Consider the example shown in Figure 2 where 9 requests are processed on three servers. The requests are labeled $T_1$ to $T_9$ and there are two backup slots (i.e., six recovery sections). Suppose that requests $T_3$ and $T_8$ become faulty on the top server during the third time unit and the bottom server during the second time unit, respectively. Request $T_8$ is recovered immediately during the third time unit ($R_8$) and the processing of request $T_9$ is postponed. Therefore, before using backup slots, there are two requests to be processed/re-processed; the original request $T_9$ and the recovery request $R_3$.

### 3.1 Restricted Serial Recovery

The restricted serial recovery scheme limits the re-processing of a faulty request to the *same* server. For example, Figure 2a shows that $T_3$ is recovered by $R_3$ on the top server while $T_8$ is recovered by $R_8$ on the bottom server.



a. Restricted serial recovery b. Parallel recovery c. Adaptive parallel recovery

**Fig. 3.** The maximum number of faults that can be tolerated by different recovery schemes in the worst case.

It is easy to see that, with $b$ backup slots, the restricted serial recovery scheme can only recover from $b$ faults in the worst case (either during primary or backup execution). For example, as shown in Figure 3a, if there is a fault that causes request $T_3$ to be faulty during primary execution, we can only tolerate one more fault in the worst case when the fault causes $T_3$'s recovery, $R_3$, to be faulty. One additional fault could cause the second recovery $RR_3$ of request $T_3$ to be faulty and lead to system failure since the recovery of the faulty requests is restricted to the same server.

### 3.2 Parallel Recovery

If faulty requests can be re-processed on multiple servers in parallel, we can allocate multiple recovery sections to recover one faulty request concurrently. The parallel recovery scheme considers all recovery sections at the beginning of backup slots and equally allocates them to the remaining requests. For the above example, there are 6 recovery sections in total and each of the remaining requests $R_3$ and $T_9$ gets three recovery sections. The schedule is shown in Figure 2b.

Suppose that there are $i$ faults during primary execution and $i$ requests remain to be processed/re-processed at the beginning of the backup slots. With $b{\cdot}p$ recovery sections in total, each remaining request will get at least $\lfloor \frac{b\cdot p}{i} \rfloor$ recovery sections. That is, at most $\lfloor \frac{b\cdot p}{i} \rfloor - 1$ additional faults can be tolerated. Therefore, when there are $i$ faults during primary execution, the number of additional faults during the backup execution that can be tolerated by parallel recovery is:

$$PR(b, p, i) = \left\lfloor \frac{b \cdot p}{i} \right\rfloor - 1 \tag{3}$$

Let $PR_{b,p}$ represents the maximum number of faults that can be tolerated by $p$ servers with $b$ backup slots in the worst case. Hence:

$$PR_{b,p} = \min_{1 \le i \le \min\{b\cdot p, n\cdot p\}} \{i + PR(b, p, i)\} \tag{4}$$

Notice that, $w$ $(= n{\cdot}p)$ is the maximum number of faults that could occur during the $n$ primary time units. That is, $i \le n{\cdot}p$. Furthermore, we have $i \le b{\cdot}p$ because it is not feasible for $b{\cdot}p$ recovery sections to recover more than $b{\cdot}p$ faulty requests. Algebraic manipulations show that the value of $PR_{b,p}$ is obtained when:

$$i = \min \left\{ n \cdot p, \left\lfloor \sqrt{b \cdot p} \right\rfloor + u \right\}. \tag{5}$$

where $u$ equals 0 or 1 depending on the floor operation in Equation 3. For the example in Figure 2, we have $PR_{2,3} = 4$ when $i = 2$ (illustrated in Figure 3b) or $i = 3$. That is, for the case shown in Figure 3b, two more faults can be tolerated in the worst case and we can achieve a 4-fault tolerant system. One additional fault could cause the third recovery section for $R_3$ to be faulty and lead to a system failure. Notice that, although $T_9$ is processed successfully during the first backup slot, the other two recovery sections in the second backup slot that are allocated to $T_9$ can not be used by $R_3$ due to the fixed recovery schedule.

### 3.3 Adaptive Parallel Recovery

Instead of considering all recovery sections together, we can use *one* backup slot *at a time* and adaptively allocate the recovery sections to improve the performance and tolerate more faults. For example, as shown in Figure 2c, we first use the three recovery sections in the first backup slot to process/re-process the remaining two requests. The recovery $R_3$ is processed on two servers and request $T_9$ on one server. If the server that processes $T_9$ happens to encounter a fault, the recovery $R_9$ can be processed using all recovery sections in the second backup slot on all three servers, thus allowing two additional faults as shown in Figure 3c. Therefore, a 5-fault tolerant system is achieved. Compared to the simple parallel recovery scheme, one more fault could be tolerated.

In general, suppose that there are $i$ requests remaining to be processed/re-processed before using backup slots. Since there are $p$ recovery sections within one backup slot, we can use the first backup slot to process up to $p$ remaining requests. If $i > p$, the remaining requests and any new faulty requests during the first backup slot will be processed on the following $b-1$ backup slots. If $i \leq p$, requests are processed redundantly using a round-robin scheduler. In other words, $p - i \left\lfloor \frac{p}{i} \right\rfloor$ requests are processed with the redundancy of $\left\lfloor \frac{p}{i} \right\rfloor + 1$ and the other requests are processed with the redundancy of $\left\lfloor \frac{p}{i} \right\rfloor$.

Assuming that $z$ requests need to be processed/re-processed after the first backup slot, then the same recovery algorithm that is used in the first backup slot to process $i$ requests is used in the second backup slot to process $z$ requests; and the process is repeated for all $b$ backup slots.

With the adaptive parallel recovery scheme, suppose that $APR_{b,p}$ is the worst case maximum number of faults that can be tolerated using $b$ backup slots on $p$ servers. We have:

$$APR_{b,p} = \min_{1 \leq i \leq \min\{b \cdot p, n \cdot p\}} \{i + APR(b, p, i)\} \tag{6}$$

where $i$ is the number of faults during the primary execution and $APR(b, p, i)$ is the maximum number of additional faults that can be tolerated during $b$ backup slots in the worst case distribution of the faults.

In Equation 6, $APR_{b,p}$ is calculated by considering different number of faults, $i$, occurred in the primary execution and estimating the corresponding number of faults allowed in the worst case in backup slots, $APR(b, p, i)$, and then taking the minimum over all values of $i$. Notice that at most $w = n \cdot p$ faults can occur during the primary execution of $w$ requests and at most $b \cdot p$ faults can be recovered with $b$ backup slots. That is $i \leq \min\{n \cdot p, b \cdot p\}$. Hence, $APR(b, p, i)$ can be found iteratively as shown below:

$$APR(1, p, i) = \left\lfloor \frac{p}{i} \right\rfloor - 1 \tag{7}$$

$$APR(b, p, i) = \min_{x(i) \leq J \leq y(i)} \{J + APR(b-1, p, z(i, J))\} \tag{8}$$

When $b = 1$ (i.e., $i \leq p$), Equation 7 says that the maximum number of additional faults that can be tolerated in the worst case is $\left\lfloor \frac{p}{i} \right\rfloor - 1$. That is, one more fault could cause a recovery failure that leads to a system failure since at least one request is recovered with redundancy $\left\lfloor \frac{p}{i} \right\rfloor$.

For the case of $b > 1$, in Equation 8, $J$ is the number of faults during the first backup slot and $z(i, J)$ is the number of requests that still need to be processed during the remaining $b-1$ backup slots. We search all possible values of $J$ and the minimum value of $J + APR(b-1, p, z(i, J))$ is the worst case maximum number of additional faults that can be tolerated during $b$ backup slots.

The bounds on $J$, $x(i)$ and $y(i)$, depend on $i$, the number of requests that need to be processed during $b$ backup slots. When $i > p$, we have enough requests to be processed and the first backup slot is used to process $p$ requests (each on one server). When $J$ ($0 \leq J \leq p$) faults happen during the first backup slot and the total number of requests that remain to be processed during the remaining $b-1$ backup slots is $z(i, J) = i - p + J$. Since we should have $z(i, J) \leq (b-1)p$, then $J$ should not be larger than $b \cdot p - i$. That is, when $i > p$, we have $x(i) = 0$, $y(i) = \min\{p, b \cdot p - i\}$ and $z(i, J) = i - p + J$.

When $i \leq p$, all requests are processed during the first backup slot with the least redundancy being $\lfloor \frac{p}{i} \rfloor$. To get the maximum number of faults that can be tolerated, at least one recovery failure is needed during the first backup slot such that the remaining $b - 1$ backup slots can be utilized. Thus, the lower bound for $J$, the number of faults during the first backup slot, is $x(i) = \lfloor \frac{p}{i} \rfloor$. Therefore, $\lfloor \frac{p}{i} \rfloor = x(i) \leq J \leq y(i) = p$. When there are $J$ faults during the first backup slot, the maximum number of recovery failures in the worst case is $z(i, J)$, which is also the number of requests that need to be processed during the remaining $b - 1$ backup slots. From the adaptive parallel recovery scheme, it is not hard to get $z(i, J) = \left\lfloor \frac{J}{\lfloor p/i \rfloor} \right\rfloor$ when $\lfloor \frac{p}{i} \rfloor \leq J \leq (i - p + i\lfloor \frac{p}{i} \rfloor)\lfloor \frac{p}{i} \rfloor$ and $z(i, J) = (i - p + i\lfloor \frac{p}{i} \rfloor) + \left\lfloor \frac{J - (i-p+i\lfloor \frac{p}{i} \rfloor)\lfloor \frac{p}{i} \rfloor}{\lfloor p/i \rfloor + 1} \right\rfloor$ when $(i - p + i\lfloor \frac{p}{i} \rfloor)\lfloor \frac{p}{i} \rfloor < J \leq p$.

For the example in Figure 2, applying Equations 7 and 8, we get $APR(2, 3, 1) = 5$. That is, if there is only one fault during the primary execution, it can tolerate up to 5 faults since all 6 recovery sections will be redundant. Similarly, $APR(2, 3, 2) = 3$ (illustrated in Figure 3c), $APR(2, 3, 3) = 2$, $APR(2, 3, 4) = 1$, $APR(2, 3, 5) = 0$ and $APR(2, 3, 6) = 0$. Thus, from Equation 6, $APR_{2,3} = \min_{i=1}^{6}\{i + APR(2, 3, i)\} = 5$.

### 3.4 Arbitrary Number of Requests

We have discussed the case where the number of requests, $w$, in an interval is a multiple of $p$, the number of working servers. Next, we focus on extending the results to the case where $w$ is *not* a multiple of $p$.

Without loss of generality, suppose that $w = n \cdot p + d$, where $n \geq 1$ and $0 < d < p$. Thus, processing all requests will need $(n + 1)$ primary time units. However, the last primary time unit is not fully scheduled with requests. If we consider the last primary time unit as a backup slot, there will be *at least* $d$ requests that need to be processed after finishing the execution in the first $n$ time units.

Therefore, similar to Equations 3 and 6, the worst case maximum number of faults that can be tolerated with $b$ backup slots can be obtained as:

$$PR_{b+1,p} = \min_{d \leq i \leq \min\{w,(b+1)\cdot p\}} \{i + PR(b + 1, p, i)\} \tag{9}$$

$$APR_{b+1,p} = \min_{d \leq i \leq \min\{w,(b+1)\cdot p\}} \{i + APR(b + 1, p, i)\} \tag{10}$$

where $i$ is the number of requests to be processed/re-processed on $b + 1$ backup slots. $PR(b + 1, p, i)$ and $APR(b + 1, p, i)$ are defined as in Equations 3 and 8, respectively. That is, we pretend to have $b + 1$ backup slots and treat the last $d$ requests that are not scheduled within the first $n$ time units as faulty requests. Therefore, the minimum number of faulty requests to be processed/re-processed is $d$ and the maximum number of faulty requests is $\min\{w, (b + 1) \cdot p\}$, which are shown as the range of $i$ in Equations 9 and 10.

### 3.5 Maximum Number of Tolerated Faults

To illustrate the performance of different recovery schemes, we calculate the worst case maximum number of faults that can be recovered by $p$ servers with $b$ backup slots under different recovery schemes. Recall that, for the restricted serial recovery scheme, the number of faults that can be tolerated in the worst case is the number of available backup slots $b$ and is independent of the number of servers that work in parallel.

**Table 1.** The worst case maximum number of faults that can be tolerated by $p$ servers with $b$ backup slots.

| $b$ | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $p = 4$ | parallel | 3 | 4 | 6 | 7 | 8 | 8 | 9 | 10 | 11 | 11 | 12 | 12 | 13 | 14 | **14** | **16** |
| | adaptive | 3 | 6 | 10 | 14 | 18 | 22 | 26 | 30 | 34 | 38 | 42 | 46 | 50 | 54 | 58 | 78 |
| $p = 8$ | parallel | 4 | 7 | 8 | 10 | 11 | 12 | 14 | 15 | 16 | 16 | 17 | 18 | 19 | 20 | 20 | 24 |
| | adaptive | 4 | 10 | 17 | 24 | 31 | **39** | **47** | **55** | **63** | **71** | **79** | **87** | **95** | **103** | **111** | **151** |

Assuming that the number of requests $w$ is a multiple of $p$ and is more than the number of available recovery sections, Table 1 gives the worst case maximum number of faults that can be tolerated by a given number of servers with different numbers of backup slots under the parallel and adaptive parallel recovery schemes. From the table, we can see that the number of faults that can be tolerated by the parallel recovery scheme may be less than what can be tolerated by the restricted serial recovery scheme. For example, with $p = 4$, restricted serial recovery scheme can tolerated 15 and 20 faults when $b = 15$ and $b = 20$, respectively. However, parallel recovery can only tolerate 14 and 16 faults respectively. The reason comes from the unwise decision of fixing allocation of all recovery slots, especially for larger number of backup slots. When the number of backup slots equals 1, the two parallel recovery schemes have the same behavior and can tolerate the same number of faults.

From Table 1, we can also see that the adaptive parallel recovery scheme is much more efficient than the restricted serial recovery and the simple parallel recovery schemes, especially for higher levels of parallelism and larger number of backup slots. Interestingly, for the adaptive parallel recovery scheme, the number of faults that can be tolerated by $p$ servers increases linearly with the number of backup slots $b$ when $b$ is greater than a certain value that depends on $p$. For example, with $p = 8$, after $b$ is greater than 5, the number of faults that can be tolerated using adaptive parallel recovery scheme increases by 8 when $b$ is incremented. However, for $p = 4$, when $b > 2$, the number of faults increases by 4 when $b$ is incremented.

## 4 Optimal Number of Active Servers

In what follows, we consider two optimization problems. First, for a given performability goal (e.g., $k$-fault tolerance), what is the optimal number of active servers needed to minimize system energy consumption? Second, for a limited energy budget, what is the optimal number of active servers needed to maximize system performability (e.g., in terms of number of faults to be tolerated)? In either case, we assume that the number of available servers is $M$ and that after determining the optimal number of servers $p$, the remaining $M - p$ servers are turned off to save energy.

### 4.1 Minimize Energy with Fixed Performability Goal

To achieve a $k$-fault tolerant system, we may use different number of servers that consume different amount of energy. In the last section we have shown how to compute the maximum number of faults, $k$, that can be tolerated by $p$ servers with $b$ backup slots in the worst case. Here, we use the same analysis for the inverse problem. That is, finding the *least* number of backup slots, $b$, needed by $p$ servers to tolerate $k$ faults.

For a given recovery scheme, let $b$ be the number of backup slots needed by $p$ servers ($p \leq M$) to guarantee that any $k$ faults can be tolerated. If $b$ is more than the available slack units (i.e., $b > D - \left\lceil \frac{w}{p} \right\rceil$), it is not *feasible* for $p$ servers to tolerate $k$ faults during the processing of all requests within the interval considered. Suppose that $b \leq D - \left\lceil \frac{w}{p} \right\rceil$, the amount of remaining slack time on each server is $slack = D - \left\lceil \frac{w}{p} \right\rceil - b$. Expecting that no faults will occur (i.e., being optimistic), the slack can be used to scale down the primary execution of requests while the recoveries are executed at the maximum frequency $f_{max}$ if needed. Alternatively, expecting that all faults will occur (i.e., being pessimistic), we can use the slack to scale down the primary execution as well as all recovery execution to minimize the expected energy consumption.

Expecting that $k_e$ ($\leq k$) faults will occur (i.e., $k_e$*-pessimism*) and assuming that $b_e$ ($\leq b$) is the least number of backup slots needed to tolerate $k_e$ faults, the slack time is used to scale down the primary execution as well as the recovery execution during the first $b_e$ backup slots. The recovery execution during the remaining backup slots is executed at the maximum frequency $f_{max}$ if more than $k_e$ faults occur. Here, optimistic analysis corresponds to $k_e = 0$ and pessimistic analysis corresponds to $k_e = k$. Thus, the $k_e$*-pessimism expected energy consumption* is:

$$E(k_e) = p \cdot \left[ P_s D + (P_{ind} + C_{ef} f^m(k_e)) \frac{\lceil w/p \rceil + b_e}{f(k_e)} \right] \tag{11}$$

where

$$f(k_e) = \min \left\{ \frac{\lceil w/p \rceil + b_e}{D - (b - b_e)}, f_{ee} \right\} \tag{12}$$

$$\tag{13}$$

is the frequency to process all original requests and the recovery requests during the first $b_e$ backup slots. Recall that $f_{ee}$ is the minimum energy efficient frequency (see Section 2).

Searching through all feasible number of servers, we can get the optimal number of servers to minimize the expected energy consumption while tolerating $k$ faults during the processing of all requests within the interval of $D$. Notice that, finding the least number of backup slots $b$ to tolerate $k$ faults has a complexity of $\mathbf{O}(k)$ and checking the feasibility of all possible numbers of servers has a complexity of $\mathbf{O}(M)$. Therefore, the complexity of finding the optimal number of servers to minimize the expected energy consumption is $\mathbf{O}(kM)$.

### 4.2 Maximize Performability with Fixed Energy Budget

When the energy budget is limited, we may not be able to power up all $M$ servers at the maximum frequency. The more servers are employed, the lower the frequency at which the servers can run. Different numbers of active servers will run at different frequencies and thus lead to different maximum number of faults that can be tolerated within the interval considered. In this section, we consider the optimal number of servers to maximize the number of faults that can be tolerated with fixed energy budget.

Notice that, from the power model discussed in Section 2, it is the most energy efficient to scale down all the employed servers uniformly within the interval. With the length of the interval considered being $D$ and with limited energy budget, $E_{budget}$, the maximum power level that a system can consume is:

$$P_{budget} = \frac{E_{budget}}{D} \tag{14}$$

For active servers, the minimum power level is obtained when every server runs at the minimum energy efficient frequency $f_{ee}$. Thus, the minimum power level for $p$ servers is:

$$P_{min}(p) = p(P_s + P_{ind} + C_{ef} f_{ee}^m) = p(\alpha + \beta + \kappa^m) P_d^{max} \tag{15}$$

If $P_{min}(p) > P_{budget}$, $p$ servers are not feasible in terms of power consumption. Suppose that $P_{min}(p) \leq P_{budget}$, which means that the servers may run at a higher frequency than $f_{ee}$. Assuming that the frequency is $f_{budget}(p)$, we have:

$$f_{budget}(p) = \sqrt[m]{\frac{P_{budget}}{p \cdot P_d^{max}} - \alpha - \beta} \tag{16}$$

The total time needed for executing all requests at frequency $f_{budget}(p)$ is:

$$t_{primary} = \frac{\lceil w/p \rceil}{f_{budget}(p)} \tag{17}$$

If $t_{primary} > D$, $p$ servers cannot finish processing all requests within the interval considered under the energy budget. Suppose that $t_{primary} \leq D$. We have $D - t_{primary}$ units of slack time and the number of backup slots that can be scheduled at frequency $f_{budget}(p)$ is:

$$b_{budget}(p) = (D - t_{primary}) f_{budget}(p) = D \cdot f_{budget}(p) - \left\lceil \frac{w}{p} \right\rceil \tag{18}$$

From Section 3, the worst case maximum number of faults that can be tolerated by $p$ servers using restricted recovery scheme is $b_{budget}(p)$. For parallel recovery schemes, from Equations 9 and 10, the maximum number of faults that can be tolerated within the interval considered is either $PR_{p,b_{budget}(p)}$ (for the parallel recovery scheme) or $APR_{p,b_{budget}(p)}$ (for the adaptive parallel recovery scheme).

For a given recovery scheme, by searching all feasible numbers of servers, we can get the optimal number of servers that maximizes the worst case maximum number of faults to be tolerated within the interval $D$.

## 5 Analytical Results and Discussion

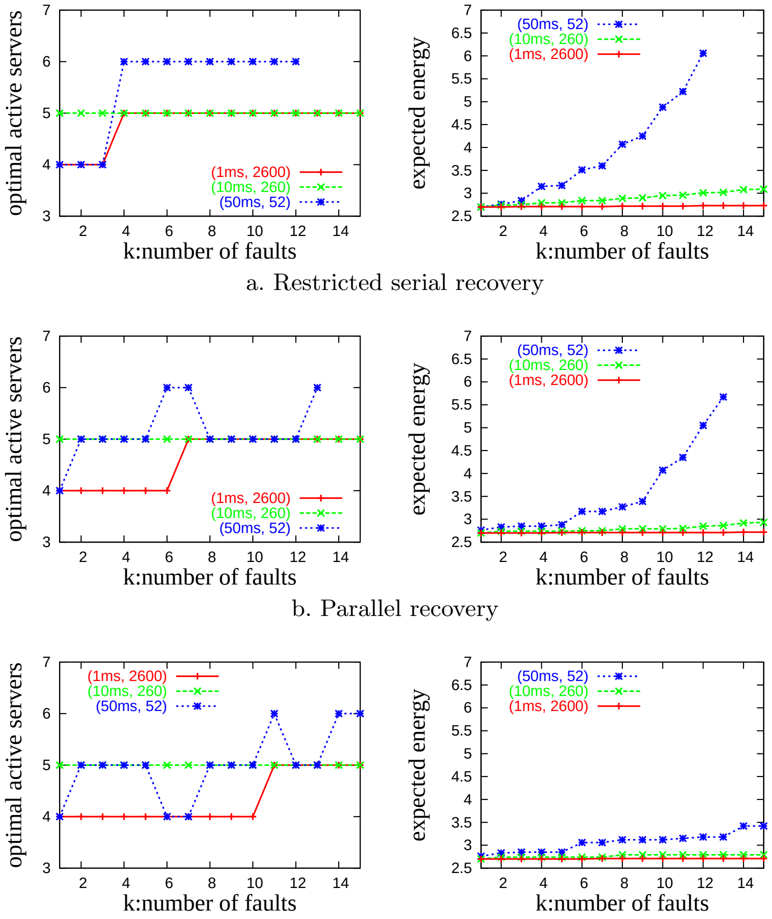
Generally, the exponent $m$ for frequency-dependent power is between 2 and 3 [2]. We use $m = 3$ in our analysis. The maximum frequency is assumed to be $f_{max} = 1$ and the maximum frequency-dependent power is $P_d^{max} = C_{ef} f_{max}^m = 1$. Considering that processor and memory power can be reduced by up to 98% of their active power when hibernating [4, 12], the values of $\alpha$ and $\beta$ are assumed to be 0.1 and 0.3 respectively. These values are justified by observing that the Intel Pentium M processor consumes $25W$ peak power with sleep power around $1W$ [4] and a RAMBUS memory chip consumes $300mW$ active power with sleep power of $3mW$ [17].

In our analysis, we focus on varying the size of requests, request arrival rate (i.e., system load), the number of faults to be tolerated ($k$) and the recovery schemes to see how they affect the optimal number of active servers. We consider a system that consists of 6 servers. The interval considered is 1 second (i.e., worst case response time is 2 seconds) and three different request sizes are considered: $1ms$, $10ms$ and $50ms$. The number of expected faults is assumed to be $k_e = \lfloor \frac{k}{2} \rfloor$.

### 5.1 Optimal Number of Servers for Energy Minimization

Define *system load* as the ratio of the total number of requests arrived in one interval over the number of requests that can be handled by *one* server within one interval. With 6 servers, the maximum system load that can be handled is 6. To get enough slack for illustrating the variation of the optimal number of servers, we consider a system load of 2.6. Recall that the interval considered is 1 second, different request arrival rates are used for different request sizes to obtain the system load of 2.6.

The left figures in Figure 4abc show the optimal number of active servers used (the remaining servers are turned off for energy efficiency) to tolerate a given number of faults, $k$, under different recovery schemes. The two numbers in the legends stand for request size and request arrival rate (in terms of number of requests per second), respectively. From the figure, we can see that the optimal number of servers generally increases with the number of faults to be tolerated. However, due to the effect of sleep power, the optimal number of servers does not increase monotonically when the number of faults to be tolerated increases, especially for the case of large request size where more slack time is needed as temporal redundancy for the same number of backup slots. Moreover, for the case of request size being $50ms$, restricted serial recovery can only tolerate 12 faults and parallel recovery can tolerate 13 faults within the interval considered, while adaptive parallel recovery can tolerate at least 15 faults.



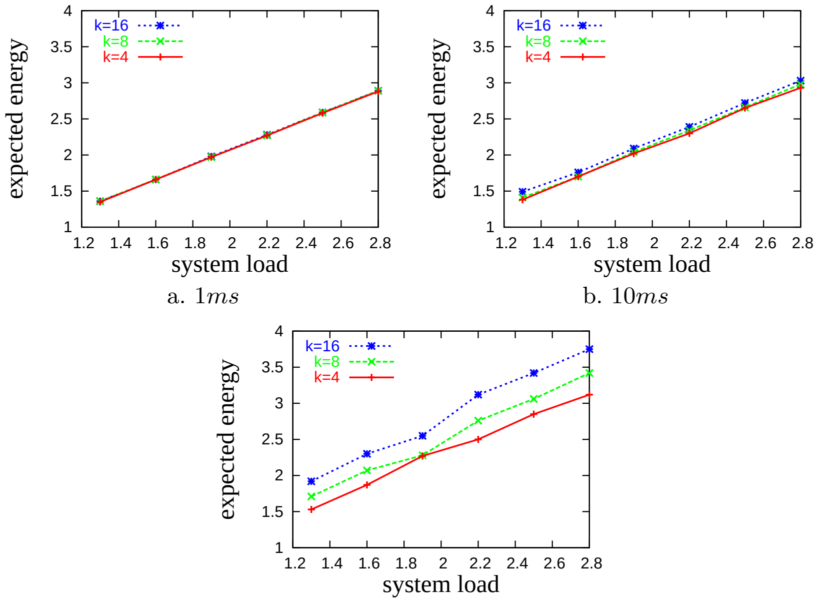
a. Restricted serial recovery

b. Parallel recovery

c. Adaptive parallel recovery

**Fig. 4.** The optimal number of active servers and the corresponding expected minimum energy consumption.

The right figures in Figure 4abc show the corresponding expected energy consumption when the optimal number of servers are employed. Recall that the normalized power is used. For each server, the maximum frequency-dependent power is $P_d^{max} = 1$, sleep power is $P_s = 0.1$ and frequency-independent power is $P_{ind} = 0.3$. From the figure, we can see that, when the request size is $1ms$, the minimum expected energy consumption is almost the same for different numbers of faults to be tolerated. The reason is that, to tolerate up to 15 faults, the amount of slack time used by the backup slots is almost negligible and the amount of slack time used for energy management is more or less the same when each backup slot is only $1ms$. However, when the request size is $50ms$, the size of one backup slot is also $50ms$ and the minimum expected energy consumption increases significantly when the number of faults to be tolerated increases. This comes from the fact that each additional backup slot needs relatively more slack time and less slack is left for energy management when the number of faults to be tolerated increases. Compared with restricted serial recovery and parallel recovery, to tolerate the same number of faults, the adaptive parallel recovery scheme needs fewer backup slots and leaves more slack for energy management. From the figure, we can also see that the adaptive parallel recovery scheme consumes the least amount of energy, especially for larger requests.



a. $1ms$

b. $10ms$

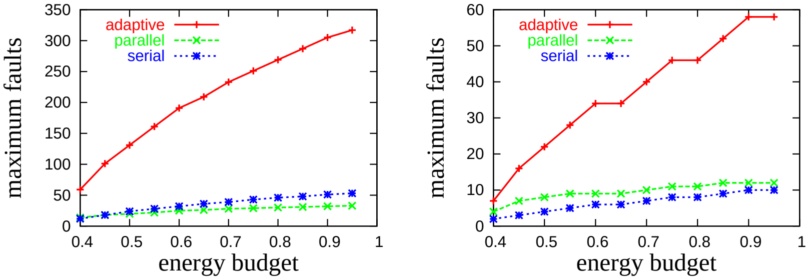
c. $50ms$

**Fig. 5.** The minimum expected energy consumption under different system load for different request sizes to tolerate given numbers of faults. The adaptive parallel recovery scheme is used and $k_e = \frac{k}{2}$.

For different sizes of requests under adaptive parallel recovery scheme, Figure 5 further shows the expected energy consumption to tolerate given numbers of faults under different system loads. For different request sizes, different request arrival rates are used to obtain a certain system load. When system load increases, more requests need to be processed within one interval and the expected energy consumption to tolerate given numbers (e.g., 4, 8 and 16) of faults increases. As before, when the request size is $1ms$, the expected energy consumption is almost the same to tolerate 4, 8 or 16 faults within the interval of 1 second. The difference in the expected energy consumption increases for larger size of requests.

### 5.2 Optimal Number of Servers for Performability Maximization

Assume that the maximum power, $P_{max}$, corresponds to running all servers with the maximum processing frequency $f_{max}$. When the energy budget for each interval is limited, we can only consume a fraction of $P_{max}$ when processing requests during a given interval. For different energy budgets (i.e., different fraction of $P_{max}$), Figure 6 shows the worst case maximum number of faults that can be tolerated when the optimal number of active servers are used. The optimal number of active servers increases when energy budget increases but we did not show the results due to space limitation. Here, we consider fixed system load of 2.6. From the figure, we can see that the number of faults that can be tolerated increases with increased energy budget. When the request size increases, there are less available backup slots due to the large slot size and fewer faults can be tolerated. When the number of backup slots is very large (e.g., for the case of $10ms$ with 260 requests/second), the same as shown in Section 3, parallel recovery performs worse than restricted serial recovery. Adaptive parallel recovery performs the best and can tolerate many more faults than the other two recovery schemes at the expense of more complex management of backup slots.



a. $10ms$ and 260 requests/second     b. $50ms$ and 52 requests/second

**Fig. 6.** The worst case maximum number of faults that can be tolerated with limited energy budget for different sizes of requests.

## 6 Conclusions

In this work, we consider an event-driven application and a system that consists of a fixed number of servers. To efficiently use slack time as temporal redundancy for providing reliable service, we first propose an adaptive scheme that recovers requests from faults in parallel. Furthermore, we show that this scheme leads to higher reliability than serial or non-adaptive parallel recovery schemes.

Assuming self-detection mechanisms in each server, we consider two problems that exhibit trade-offs between energy consumption and system performability.

The first problem is to determine the optimal number of servers that minimizes the expected energy consumption while guaranteeing $k$-fault tolerance. The second problem is to maximize the number of faults that can be tolerated with limited energy budget. As expected, our analysis results show that more energy is needed if more faults are to be tolerated. Due to static power consumption in servers, the optimal number of servers needed for $k$-fault tolerance does not increase monotonically when the number of faults to be tolerated increases. For the same number of faults, large requests will need more slack for recovery and thus is expected to consume more energy. Parallel recovery schemes with a fixed recovery schedule may perform worse than serial recovery. However, adding adaptivity to the parallel recovery process requires less slack to tolerate a given number of faults, leaving more slack for energy management and thus results in less energy being consumed.

When self-detection mechanisms are not available in the system considered, we can further combine modular redundancy and parallel recovery to obtain reliable service. In our future work, we will explore the optimal combination of modular redundancy and parallel recovery to minimize energy consumption for a given performability goal or to maximize performability for a given energy budget.

## References

1. P. Bohrer, E. N. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, C. McDowell, and R. Rajamony. *The case for power management in web servers*, chapter 1. Power Aware Computing. Plenum/Kluwer Publishers, 2002.
2. T. D. Burd and R. W. Brodersen. Energy efficient cmos microprocessor design. In *Proc. of The HICSS Conference*, Jan. 1995.
3. X. Castillo, S. McConnel, and D. Siewiorek. Derivation and calibration of a transient error reliability model. *IEEE Trans. on computers*, 31(7):658–671, 1982.
4. Intel Corp. Mobile pentium iii processor-m datasheet. Order Number: 298340-002, Oct 2001.
5. E. (Mootaz) Elnozahy, M. Kistler, and R. Rajamony. Energy-efficient server clusters. In *Proc. of Power Aware Computing Systems*, 2002.
6. E. (Mootaz) Elnozahy, R. Melhem, and D. Mossé. Energy-efficient duplex and tmr real-time systems. In *Proc. of The IEEE Real-Time Systems Symposium*, 2002.
7. http://developer.intel.com/design/intelxscale/.
8. http://www.transmeta.com.
9. T. Ishihara and H. Yauura. Voltage scheduling problem for dynamically variable voltage processors. In *Proc. of The 1998 International Symposium on Low Power Electronics and Design*, Aug. 1998.
10. K. M. Kavi, H. Y. Youn, and B. Shirazi. A performability model for soft real-time systems. In *Proc. of the Hawaii International Conference on System Sciences (HICSS)*, Jan. 1994.
11. R. Koo and S. Toueg. Checkpointing and rollback recovery for distributed systems. *IEEE Trans. on Software Engineering*, 13(1):23–31, 1987.
12. A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In *Proc. of the* $9^{th}$ *International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

13. H. Lee, H. Shin, and S. Min. Worst case timing requirement of real-time tasks with time redundancy. In *Proc. of Real-Time Computing Systems and Applications*, 1999.
14. C. Lefurgy, K. Rajamani, Freeman Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *IEEE Computer*, 36(12):39–48, 2003.
15. R. Melhem, D. Mossé, and E. (Mootaz) Elnozahy. The interplay of power management and fault recovery in real-time systems. *IEEE Trans. on Computers*, 53(2):217–231, 2004.
16. D. K. Pradhan. *Fault Tolerance Computing: Theory and Techniques*. Prentice Hall, 1986.
17. Rambus. Rdram. http://www.rambus.com/, 1999.
18. K. Seth, A. Anantaraman, F. Mueller, and E. Rotenberg. Fast: Frequency-aware static timing analysis. In *Proc. of the IEEE Real-Time System Symposium*, 2003.
19. V. Sharma, A. Thomas, T. Abdelzaher, K. Skadron, and Z. Lu. Power-aware qos management in web servers. In *Proc. of the* $24^{th}$ *IEEE Real-Time System Symposium*, Dec. 2003.
20. K. G. Shin and H. Kim. A time redundancy approach to tmr failures using fault-state likelihoods. *IEEE Trans. on Computers*, 43(10):1151 – 1162, 1994.
21. A. Sinha and A. P. Chandrakasan. Jouletrack - a web based tool for software energy profiling. In *Proc. of Design Automation Conference*, Jun 2001.
22. S. Thompson, P. Packan, and M. Bohr. Mos scaling: Transistor challenges for the 21st century. *Intel Technology Journal*, Q3, 1998.
23. O. S. Unsal, I. Koren, and C. M. Krishna. Towards energy-aware software-based fault tolerance in real-time systems. In *Proc. of The International Symposium on Low Power Electronics Design (ISLPED)*, Aug. 2002.
24. M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *Proc. of The First USENIX Symposium on Operating Systems Design and Implementation*, Nov. 1994.
25. F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proc. of The* $36^{th}$ *Annual Symposium on Foundations of Computer Science*, 1995.
26. Y. Zhang and K. Chakrabarty. Energy-aware adaptive checkpointing in embedded real-time systems. In *Proc. of IEEE/ACM Design, Automation and Test in Europe Conference(DATE)*, 2003.
27. Y. Zhang and K. Chakrabarty. Task feasibility analysis and dynamic voltage scaling in fault-tolerant real-time embedded systems. In *Proc. of IEEE/ACM Design, Automation and Test in Europe Conference(DATE)*, 2004.
28. D. Zhu, R. Melhem, D. Mossé, and E.(Mootaz) Elnozahy. Analysis of an energy efficient optimistic tmr scheme. In *Proc. of the* $10^{th}$ *International Conference on Parallel and Distributed Systems (ICPADS)*, Jul. 2004.