단일칩 컴퓨터의 결함허용 스케쥴링 성능 분석

김 성 수[†]

요 약

본 논문에서는 RESO(REcomputation with Shifted Operands)와 같은 시간 결함허용 기법을 이용한 단일칩 컴퓨터의 결함허용성을 평가하기 위한 분석 및 시뮬레이션 모델을 제안한다. 단일칩으로 들어오는 모든 작업은 이중화 처리된다고 가정하고 1차 작업과 2차 작업의 효율적인 처리를 위한 세가지 스케쥴링 방법들을 제안하고 분석한다. 고안된 스케쥴링 방법들은 결함과 결함허용으로 인한 응답시간 지연이 시스템의 비용에 미치는 영향을 단일칩의 부하와 결함발생율에 따라서 평가한다. 제안된 모델을 사용하면 비용, 단일칩의 부하 및 결함발생율과 같은 실험 파라미터에 기초한 최적 지연(k)를 가지는 결함허용 스케쥴을 구할 수 있다.

Performance Analysis of Fault-Tolerant Scheduling in a Uniprocessor Computer

Sungsoo Kim[†]

ABSTRACT

In this paper, we present analytical and simulation models for evaluating the operation of a uniprocessor computer which utilizes a time redundant approach (such as recomputation by shifted operands) for fault-tolerant computing. In the proposed approach, all incoming jobs to the uniprocessor are duplicated, thus two versions of each job must be processed. Three methods for appropriately scheduling the primary and secondary versions of the jobs are proposed and analyzed. The proposed scheduling methods take into account the load and the fault rate of the uniprocessor to evaluate two figures of merit for cost and profit with respect to a delay in response time due to faults and fault tolerance. Our model utilizes a fault-tolerant schedule according to which it is possible to find an optimal delay (given by k) based on empiric parameters such as cost, the load and the fault rate of the uniprocessor.

1. Introduction

To meet the increasing demand of system reliability and availability, fault-tolerant techniques have been widely employed in today's computers [1]. One of the arrangements commonly employed for achieving fault tolerance consists of providing some form of redundancy in the system. Redundancy can be applied in either the hardware (for example by using a duplex or a higher modul replication) or the software (as the N version programming redu-

※ 본 연구는 1997년도 정보통신연구관리단 대학기초연구지원사업에 의한 결과임.

† 정 회 원 : 아주대학교 정보통신대학 정보및컴퓨터공학부

논문접수 : 1998년 2월 19일, 심사완료 : 1998년 4월 8일

ndancy of [2]). This type of redundancy is referred to as *space redundancy*. As in a redundant system, replication of at least twice the hardware or software components is required, then the cost of manufacturing fault-tolerant computing systems in VLSI can be expected to increase substantially [3]. Another interesting method for attaining fault-tolerant computing is time redundancy [4]; an example of this type of approach is the recomputation with shifted operands (RESO) [5,6], in which fault tolerance is achieved by limiting the modular replication of space redundancy, but at least doubling the time required for the basic computation step.

In a multiprocessor system, processors provide some *natural form* of space redundancy. Studies have shown that demand for system resources varies in a stochastic manner, in most cases a so-called *spare capacity* exists to implement some form of space redundancy [7, 8,9]. In this type of system, the idle processors (referred to as functional spares) [7,9] can be used to process a secondary version of some of the jobs executing on the active processors. Thus, fault tolerance can be achieved at very little additional cost.

Time-redundancy techniques, such as RESO, have not been very popular partly due of the perceived time overhead associated with this type of techniques [10]. In this paper, we consider time redundancy in a uniprocessor computer and the main focus is on the effect of fault tolerance on the response time of the system. Examples of time-redundant approaches for application to arithmetic computation can be found in [11].

In the proposed approach, all incoming jobs to the uniprocessor are duplicated, thus two versions of each job will be processed: the *primary* version and the *secondary* version. A discrepancy (disagreement) in the results produced by the computation of the two versions of the same job indicates that a fault may have occurred [12]. This approach can be used for detecting transient faults (an example of a transient fault is the faulty behavior of a VLSI chip due to α particles). For detecting permanent faults, the secondary version of a job must be computed under a well-defined transformation (such as the shifted operands of RESO) [13]. The interested reader should refer to [1,10] for a detailed treatment of these methods.

One of the issues addressed in this paper, is *scheduling*. There are several methods for appropriately scheduling the primary and secondary versions of these jobs. By choosing an appropriate scheduling method, the computation of the secondary version of the jobs may not significantly affect the response time of the first version in a uniprocessor system. For example, the response time will not be increased by executing the secondary version of the job only when the uniprocessor is idle; however, comparison of the results is delayed, thus decreasing the reliable throughput of the system. An opposite method consists of scheduling alternatively the *primary* version and the *secondary* version of every job; this method will provide the highest reliable throughput, but the response for the first versions of other jobs is delayed. In a real-time system, both reliable results and response time must be satisfied; this may preclude the use of some scheduling methods for some applications.

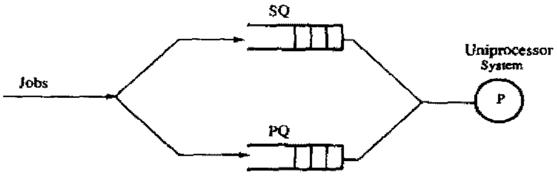
The goal of this paper is to develop analytical and simulation models to determine an appropriate scheduling method for a uniprocessor which utilizes time redundancy. The proposed model takes into account the load of the system, as well as the cost of a delayed response time and the cost of undetected faults. This paper is organized as follows.

Section 2 introduces the preliminaries and a brief review of RESO. Section 3 presents the analysis of the proposed scheduling methods. The model for the simulation approach is described in section 4. The parametric analysis of the models is presented in section 5. Discussion and conclusions are given in section 6.

2. Review and Preliminaries

A simple approach to fault tolerance by time redundancy is to recompute at least twice and compare the results of the computations [5]. This method is effective in determining whether the computation results are correct in the presence of faults which have a short duration (such as transient faults). A limitation of this method is that it starts to lose its effectiveness in the presence of faults which either are permanent, or have long duration. An example of an approach which utilizes time redundancy, is RESO [5,6,11]. RESO does not rely on a simple recomputation: a function is recomputed with shifted operands, then the result is shifted back for comparison purposes. RESO can detect (and possibly correct) many permanent faults which are not detected by simple recomputation methods. A comprehensive treatment on the fault coverage of RESO is given in [10]. The principles of RESO can be also generalized to different versions of software programs and algorithms.

In this paper, a uniprocessor computer system is analyzed. It is assumed that the system can be modeled as a classical *M*/*M*/1 queueing system in which the interarrival and service times of the jobs are mutually independent, exponentially distributed random variables with mean $1/\lambda$, and $1/\mu$, respectively. In general, for this system it is assumed that there is one job queue *Q* of infinite length. If $\mu > \lambda$, then the job queue may become empty: when the processor finishes the current job, then the processor is said to be idle. Also, if $\rho = \frac{\lambda}{\mu} < 0.5$, then the total idle time is greater than the processing time. Therefore, during the idle time the processor can be used to recompute a second version of the just completed job. For fault-tolerant computing purposes, by carefully scheduling the recomputation of the secondary version of a job, fault detection can be obtained using time redundancy at no significant increase in processing time. In this paper, the term "job" is used in a general context to identify a computation process which may consist of a single step as in [11] or a series of steps as in [5].



(Fig. 1) Computation by time redundancy in the uniprocessor system

The basic arrangement for fault-tolerant computing using time redundancy in a uniprocessor is depicted in Figure 1. Let *PQ* and *SQ* be the two job queues of infinite length. *PQ* and *SQ* are the primary and secondary queue, respectively. Whenever a job arrives, the job is sent to *PQ* and *SQ* simultaneously. The jobs in *PQ* are the original versions of the jobs (hereafter referred to as *the primary versions of the jobs*): the jobs in *SQ* are the original jobs with some additional operation (such as shifted operands or change of ordering), they are referred to as *the secondary versions of the jobs.* Usually the attached operation accounts for a negligible overhead compared with the original job, thus the processing time for the secondary version can be considered to be the same as for the primary version. However, the primary version of a job J_i has a higher priority than the secondary version of the same job. The result of the primary version of the ith job (i.e. J_i) is stored into a specific area of a memory buffer, whenever the result of the secondary version of J_i is available, the system compares the two results: if they are the same, then a *confirm* signal is generated, else a *discard* signal is provided to the processor for initiating further processing (such as a further execution of the job).

The response time of the primary version of J_i and the response time of the secondary version of J_i are denoted as $T_p(i)$ and $T_s(i)$, respectively. Because of the processing priority in the processor, $T_p(i) < T_s(i)$. For different applications, the computation result for J_i can be made available at a later time. This time is generally referred to as the *delivery time* (and denoted as $T_d(i)$). For example, in a highly reliable system, the result of the computation can't be made available until the secondary version of J_i has completed, the computation results of both primary and secondary versions of J_i have been compared and the *confirm* signal has been issued: in this case, the delivery time is not earlier than the response time of the secondary version of the job, i.e. $T_d(i) = T_s(i)$. The *checking time* (denoted as $T_c(i)$), is the time at which the result of the secondary version of J_i is available and the comparison is performed. Depending on the scheduling policy, the checking time can be either at the same time, or later than the delivery time. In this last scenario, the delivery time is the response time of the primary version of J_i, i.e. $T_d(i) = T_p(i)$, while the checking time is the response time T_s of the secondary version of J_i. An appropriate signal is then generated. The *discard* signal indicates that the result of the computation is not reliable (disagreement has occurred), hence the system should adopt some additional actions to verify the correctness of the primary result (obviously at an extra cost). The longer the elapsed time between T_d and T_c, the more expensive is the penalty. An earlier delivery of results may provide some benefits for the system depending on the application.

As an example, consider a real-time system, in which the sensors send data (for example, a temperature variation) to a controller which is configured as slave of a processor. The controller notifies the processor that a temperature variation has occurred. A job is created in the processor and a notification of the temperature variation is issued. The results of the computation are sent from the processor to the controller. The earlier the controller gets the notification, the sooner it can act as required (for example, it can change the fuel injection to correct the temperature). The cost of the operation is therefore decreased; however, if the processor generates an erroneous response to the notification, the controller may perform the wrong action (such as increasing the temperature assuming no notification is provided from the processor, such that the controller keeps the fuel at the previous level). In this case, further processing must be performed if a longer time elapses prior to generating the *confirm* signal.

There are two extreme cases: (Case 1) the processor is always fault-free, then the result of the computation should be made available to the controller as early as possible: (Case 2) the processor can be affected by faults, then the result that is confirmed by computing twice the same job, is made available to the controller. Common operation of the system is between these two extremes, i.e. the processor is basically reliable, but it may occasionally be affected by faults (such as transients).

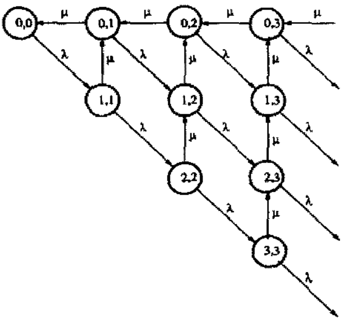
There are three scheduling methods which are studied in this paper, they correspond to the above cases as follows: (1) The processor does not execute the secondary version of the jobs until *PQ* is empty. If the secondary version of jobs can be preemptive, the response time for the primary version of jobs (which is also the deliver time) is the same as usual, i.e. as in a queueing model with one job. However, the response time for the secondary version of the job (i.e. the time to generate either the *confirm* or *discard* signal), is the longest among the three proposed scheduling methods. Therefore, this scheduling method is suitable for a processor with a low fault rate. (2) The processor executes the primary and the secondary versions of a job alternatively and then it delivers the results of the computation after the response time of the secondary job (provided the time for the comparison operation is negligible): in this case, the response time for *SQ* is the shortest, however the delivery time is the longest. (3) The processor does not execute the secondary version of the job J_i until it is either k jobs behind the primary version of J_i, or *PQ* is empty, and the system delivers the computation result of the primary version of J_i to the outside world. In this case, the delivery and the response times of the primary and the secondary versions of a job have a value between those for methods 1 and 2 described above.

3. Analysis of the Scheduling Methods

In this section, the response time of the primary and secondary versions of the jobs is studied for the three different scheduling methods [14] described in the previous section.

Scheduling Method 1: In this case, the secondary version of a job is processed only if the processor is idle due to lack of jobs in *PQ*. The following strategies can be used: the secondary version of a job is executed according to either a preemptive or non-preemptive strategy. As mentioned above, this scheduling model can be applied to a uniprocessor with a low fault rate: thus, the preemptive strategy is more suitable because it will interfere the least with the execution of the primary versions of the jobs.

The state-transition-rate diagram for the preemptive strategy is shown in Figure 2. Vertex (i,j), $(i \neq j)$ located in row i and column j of the state diagram, represents the state of the system S in which i primary versions of the jobs are either being processed, or waiting in *PQ* for service and j secondary versions of the jobs are waiting in *SQ*. Vertex $(0,j)$ (i.e. in the first row and column j of the diagram), represents the state of S in which no primary version of a job is currently being processed or waiting in *PQ* and j secondary versions of the jobs are either being processed or waiting in *SQ*. It is easy to observe that by using a preemptive strategy for the RESO computation, the primary versions of the jobs are served exactly as they would be in the original $M/M/1$ queueing system, because the secondary versions of the jobs can only be processed if *PQ* is empty, otherwise the secondary version of a job will be preempted.

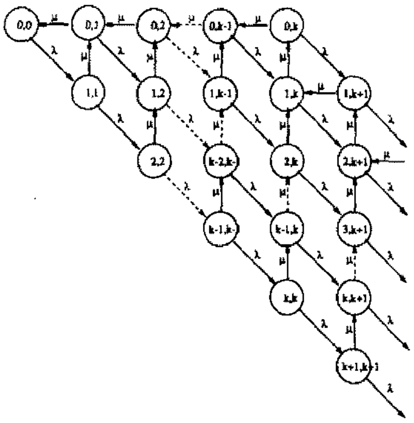


(Fig. 2) Markov chain for Model 1

Let $p_{i,j}$ be the probability that *PQ* has i jobs (including the one which is being executed) and *SQ* has j jobs. At equilibrium, the difference equations for the diagram of Figure 2 and their solutions can be regarded as a special case of Algorithm 1 for Scheduling Method 3, when $k=\infty$ (see below). Therefore, the analysis is omitted.

Scheduling Method 2: In this case, the processor executes the primary and secondary versions of a job alternatively. Let $\rho' = 2\rho = 2\frac{\lambda}{\mu}$; this scheduling model can be solved directly from the *M/M/*1 system model. The analysis is trivial, however it can be used for comparing the proposed scheduling models.

Scheduling Method 3: In this case, the secondary version of a job J_i will be processed if the primary version of J_i is k jobs ahead of it, i.e. the number of jobs in *SQ* is less than or equal to the number of jobs in *PQ* plus k. If $k=1$, this is the same scenario as Scheduling Method 2 (note that the delivery time may be different) and if $k=\infty$, this corresponds to Scheduling Method 1.



(Fig. 3) Markov chain for Model 3

If the preemptive strategy is adopted, the state-transition-rate diagram is given in Figure 3. In Figure 3, the notation of vertex (i,j) and $\lambda(\mu)$ are the same as in Figure 2. Let $p_{i,j}$ be the probability that *PQ* has i jobs and *SQ* has j jobs. The equilibrium difference equations for the state diagram of Figure 3 are as follows:

$$p_{i-1,i-1}\lambda = p_{i,i}(\lambda+\mu) \tag{1}$$

$$p_{0,0}\lambda = p_{0,1}\mu \tag{2}$$

$$p_{0,j}(\lambda+\mu) = (p_{1,j}+p_{0,j+1})\mu \quad \text{for } 1\le j\le k \tag{3}$$

$$p_{0,k}(\lambda+\mu) = p_{1,k}\mu \tag{4}$$

$$p_{i,j}(\lambda+\mu) = p_{i-1,j-1}\lambda + p_{i+1,j}\mu \quad \text{for } 1\le i< j\le i+k-2 \tag{5}$$

$$p_{i,i+k-1}(\lambda+\mu) = p_{i-1,i+k-2}\lambda + (p_{i+1,i+k-1}+p_{i,i+k})\mu \quad \text{for } i\ge 1 \tag{6}$$

$$p_{i,i+k}(\lambda+\mu) = p_{i-1,i+k-1}\lambda + p_{i+1,i+k}\mu \quad \text{for } i\ge 1 \tag{7}$$

Let $\rho = \frac{\lambda}{\mu}$ and $\theta = \frac{\lambda}{(\lambda+\mu)}$.

From(4),

$$p_{1,k} = (1+\rho)p_{0,k} \tag{8}$$

From (1), (2), (3) and (5),

$$p_{i,i} = \theta^i p_{0,0} \tag{9}$$

$$p_{0,1} = \rho p_{0,0} \tag{10}$$

$$p_{0,j} = (1+\rho)p_{0,j-1} - p_{1,j-1} \quad \text{for } 2\le j\le k \tag{11}$$

$$p_{i,j}=\theta p_{i-1,j-1}+(1-\theta)p_{i+1,j} \text{ for } 1\le i<j\le i+k-2 \quad (12)$$

From (6) and (7),

$$p_{i,i+k}=(1+\rho)p_{i,i+k-1}-\rho p_{i-1,i+k-2}-p_{i+1,i+k-1} \text{ for } i\ge 1 \quad (13)$$

$$p_{i+1,i+k}=(1+\rho)p_{i,i+k}-\rho p_{i-1,i+k-1} \text{ for } i\ge 1 \quad (14)$$

Thus, the probability for vertex (i,j), where $0\le i\le j<i+k-1$ and $i=0$, $j=k$ and $i=1$, $j=k$, can be calculated from (8)-(14). Hence, all $p_{i,j}$, $0\le i\le j\le i+k-2$ have been calculated; the remaining probabilities $p_{i,i+k-1}$ and $p_{i,i+k}$, (for $i\ge1$), can be derived from (13) and (14).

To illustrate this process an example for k=3 will be analyzed in detail. The state-transition-rate diagram for $k=3$ is shown in Figure 4. First, all $p_{i,j}$ $(0\le i\le j\le i+1)$ are calculated by using (8)-(12). The index located beside each vertex of Figure 4, shows the number in the computation sequence. For example,

step 2 : from (9) $p_{1,1}=\theta p_{0,0}$

step 3 : from (10) $p_{0,1}=\rho p_{0,0}$

step 4 : from (9) $p_{2,2}=\theta p_{1,1}=\theta^2 p_{0,0}$

step 5 : from (12) $p_{1,2}=\theta p_{0,1}+(1-\theta)p_{2,2}$

step 6 : from (11) $p_{0,2}=(1+\rho)p_{0,1}-p_{1,1}$

step 7 : from (9) $p_{3,3}=\theta p_{2,2}$

step 8 : from (12) $p_{2,3}=\theta p_{1,2}+(1-\theta)p_{3,3}$

step 9 : from (11) $p_{0,3}=(1+\rho)p_{0,2}-p_{1,2}$

step 10 : from (8) $p_{1,3}=(1+\rho)p_{0,3}$

step 11 : from (9) $p_{4,4}=\theta p_{3,3}$

step 12 : from (12) $p_{3,4}=\theta p_{2,3}+(1-\theta)p_{4,4}$

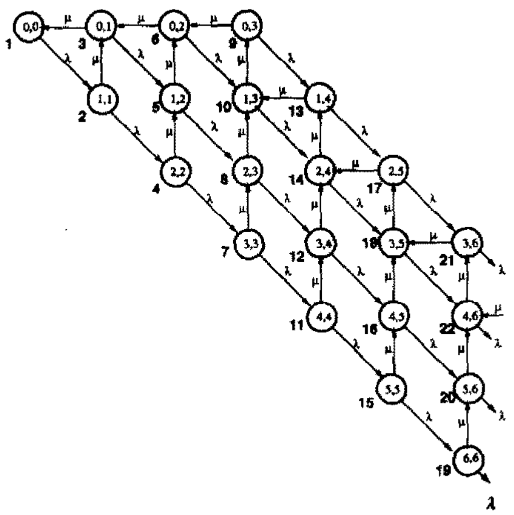
step 13 : from (13)
$p_{1,4}=(1+\rho)p_{1,3}-\rho p_{0,2}-p_{2,3}$

step 14 : from $p_{2,4}=(1+\rho)p_{1,4}-\rho p_{0,3}$

Therefore, all the probabilities can be computed (as for the required precision) along the sequence labeled in Figure 4. The algorithm for the solution of the Markov chain of Figure 3 is as follows:

Algorithm 1: The Solution of the Markov chain of Figure 3

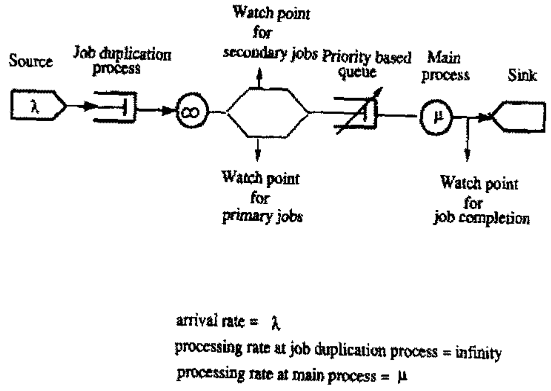
```
Step 1. Determine the computation sequence
        and store it into a queue Q (* a
        sequence is illustrated in Figure 4 *)
Step 2. Compute the probabilities according
        to the order of step 1
    p [0,0] := 1;
    FOR order := 1 TO order_number DO
        Take out the row and column
        numbers (i, j) from Q.
        (* calculate the probabilities by the
         formula ( ) - ( ), according to the
         relations between i, j and k *)
        p [i, j] := calculate_prob (i, j);
        (* ε is very small number that
         depends on precision *)
        IF p [i, j] < ε THEN BREAK;
Step 3. Calculate the sum of the probabilities,
        sum_of_p inclusive of p [0,0].
Step 4.
    FOR all i and j DO
        p [i, j] := p [i, j] / sum_of_p;
```



(Fig. 4) An example of Markov chain for Model 3 (k=3)

4. Simulation Model and Approach

In the last section, we have studied the parameters of the proposed policies by analytical methods. However, an analytical method suffers from the drawback that although in practice the two versions of a job should have the same processing time by assumption in the Markov chain, they are assumed to have the same distribution function. To estimate the difference, a discrete event simulation model that is capable of handling priority-based queues, is proposed for verifying the analytical results. In a typical simulation model, the incoming jobs are proce-ssed on a First-Come-First-Served basis; in the proposed simulation model, the simulation system is capable of selecting the appropriate versions of the jobs based on the chosen scheduling method (as presented in the previous section).



arrival rate = λ
processing rate at job duplication process = infinity
processing rate at main process = μ

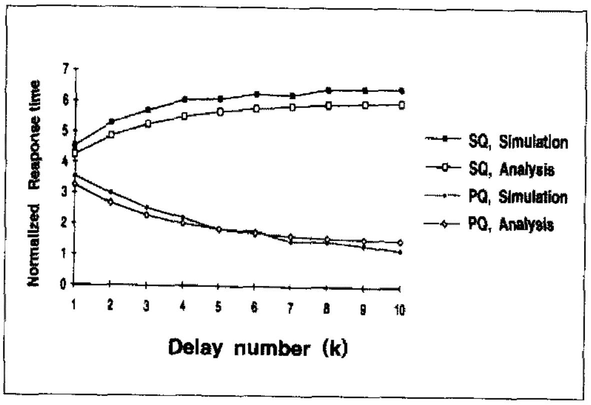
(Fig. 5) Simulation model for job scheduling

Figure 5 depicts the basic structure of the simulation model. Besides the priority-based job selection model, the other additional features which set apart this model from conventional models, are the watchpoint functions and the two versions of the simulation entity, referred to as *subprocesses*; these correspond to the primary and secondary versions of each job. The subprocesses are used to initiate two jobs at the same time. The watchpoint functions are used for collecting information from the system (such as initiation or completion time, and response time distribution).

The system is modeled using an open system model (with one source and one sink). The mean arrival rate is given by λ and the first process (with zero processing time) is used for generating the two versions of the subprocesses for each arriving job (the first is the primary version and the second is secondary version). The secondary version of a job has a priority order in the queue (not based on the order of arrival) that complies with one of the scheduling policies of the previous section.

In the simulation process, the source generates new jobs at a mean rate of λ; the first process duplicates each job by creating two version of subprocesses (one with high priority and one with low priority). Each subprocess has two identification parameters: its priority and its cumulative serial number. When the two subprocesses are created, they enter the second queue from which the second process executes them (according to its scheduling policy). The jobs are not rearranged, because changes in the order of the jobs in the queue are not allowed (unless for the removal of the job to be processed next), hence the selection of the next job to be executed is simplified.

For the second queue, an utility function is used to find the serial number of the primary version (*PVJ*) and the secondary version of a job (*SVJ*). By comparing the serial numbers, the process can decide which job should be executed next. The comparison consists of subtracting *SVJ* from *PVJ* and comparing the result with the value of the delay number k. The results of the analytical and simulation models are illustrated in Figure 6 for the normalized response time; the difference is less than 5%, thus suggesting that the analytical model fits an experimental evaluation in a satisfactory manner. The normalized response time is defined as the response time for $\mu=1$.



(Fig. 6) Comparison between simulation and analysis ($\rho=0.30$)

5. Parametric Analysis

In this section, we will study the effect of the scheduling methods (as measured by the delay number k), the fault rate of the processor and the work load on the cost of the system. In many applications (such as for real-time control), both T_d and T_c will affect the operation of the system (and consequently its cost) in the presence of faults. If k is larger, the delivery time T_d will be easier and the checking time will be later (if $T_d=T_p$ and $T_c=T_s$), and vice versa. Therefore, an optimal scheduling method may exist for a particular system.

Using Algorithm 1, we have calculated the response time of the primary and the secondary versions of the jobs under different ρ and k, where $\rho=\frac{\lambda}{\mu}$ and k is the delay number of the jobs in the secondary queue. If $k=1$, then this corresponds to Case 2, i.e. the jobs in the primary and secondary queues will be processed alternatively (and they may have different delivery time depending on the characteristics of the system). If k is a very large number, this corresponds to Case 1, i.e. the jobs in the secondary queue will not be processed until the primary queue is empty. Some results on the average queue length for the primary and secondary versions of the jobs are shown in Table 1 (where the subscript of PQ and SQ denotes the value of k). It is assumed that $\mu=1$, i.e. $\rho=\lambda$ for simplicity.

〈Table 1〉 Average queue length for the primary' and secondary versions of the jobs

ρ	PQ_1	SQ_1	PQ_2	SQ_2	PQ_3	SQ_3	PQ_5	SQ_5	PQ_{10}	SQ_{10}
0.05	0.06	0.11	0.05	0.11	0.05	0.11	0.05	0.11	0.05	0.11
0.10	0.14	0.24	0.12	0.26	0.11	0.26	0.11	0.26	0.11	0.26
0.15	0.25	0.40	0.20	0.44	0.19	0.46	0.18	0.47	0.18	0.47
0.20	0.40	0.60	0.32	0.68	0.28	0.72	0.26	0.74	0.25	0.75
0.25	0.63	0.88	0.50	1.00	0.43	1.07	0.37	1.13	0.34	1.16
0.30	0.98	1.28	0.80	1.46	0.68	1.57	0.55	1.70	0.45	1.80
0.35	1.58	1.93	1.33	2.17	1.15	2.35	0.91	2.59	0.65	2.85
0.40	2.80	3.20	2.48	3.52	2.22	3.79	1.80	4.20	1.20	4.80
0.45	6.53	6.98	6.12	7.38	5.75	7.75	5.09	8.41	3.82	9.68

As expected, for large values of k then is little difference between the time of PQ and the time of SQ if ρ is small ($\rho<0.2$), but there are significant differences provided ρ is large ($\rho>0.3$). $k=1$ is used as a benchmark for comparison purpose; thus, when k increases, the PQ time decreases, while the SQ time increases, i.e. the job delivery time is anticipated, and the verification time in the time-redundant approach is postponed for larger values of k. Usually, an earlier delivery time provides only for a limited "profit" (i.e. a decrease of the cost), as for example, the gain may be proportional to the earlier delivery time. However as indicated by T.W.Williams in [15],

> *If it costs 0.30 dollars to detect a fault at the chip level, then it would cost 3 dollars to detect that same fault when it was embedded at the board level; 30 dollars when it is embedded at the system level; and 300 dollars when it is embedded at the system level but has to be found in the field.*

then if the processor has a fault, the postponement of the verification of the two versions of a job will cause a rapid increase in penalty, i.e. a loss of "profit" (a corresponding increase in cost). This increase in cost is proportional to a polynomial factor (denoted by p) of the delay incurred in the verification process. Thus, we can use the following formula [12,15] to calculate the *Net_profit* of a system with RESO under different values of k,

$$Net\_profit = gain - loss \qquad (15)$$

where,

$$gain = advanced\_delivery\_time \times factor\_of\_gain \times (1 - fault\_rate) \qquad (16)$$

$$loss = (delayed\_verification\_time)^{p} \times (factor\_of\_loss) \times fault\_rate \qquad (17)$$

where

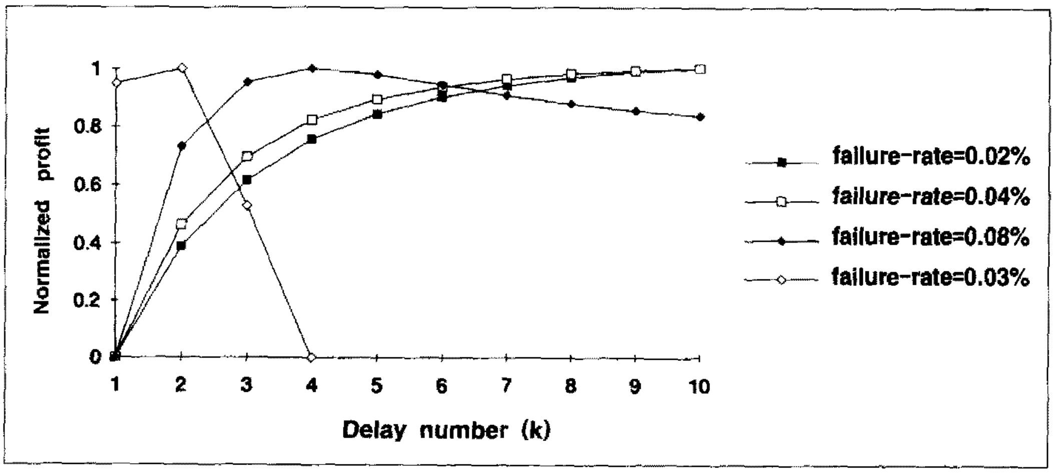
$$advanced\_delivery\_time = \max(S_{PQ}) - PQ_{time\_k=i} = PQ_{time\_k=1} - PQ_{time\_k=i} \qquad (18)$$

$$delayed\_verification\_time = SQ_{time\_k=i} - \min(S_{SQ}) = SQ_{time\_k=i} - SQ_{time\_k=1} \qquad (19)$$

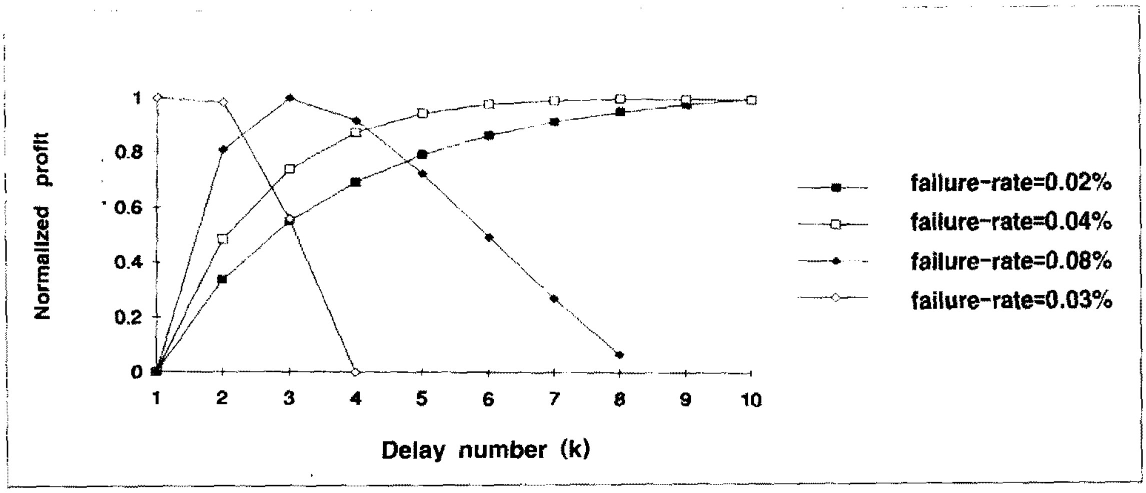
S_{PQ} (S_{SQ}) $=\{PQ_{time\_k=1}, PQ_{time\_k=2}, \cdots\}$ ($\{SQ_{time\_k=1}, SQ_{time\_k=2}, \cdots\}$) for $k=1,2,\cdots$ and PQ_{time} (SQ_{time}) stands for the normalized response time for the primary (and secondary) versions of the jobs (for a given value of k), respectively and

$$normalized\_profit = \frac{Net\_profit}{\max[Net\_profit_{k=1},\ Net\_profit_{k=2}, \cdots]} \qquad (20)$$

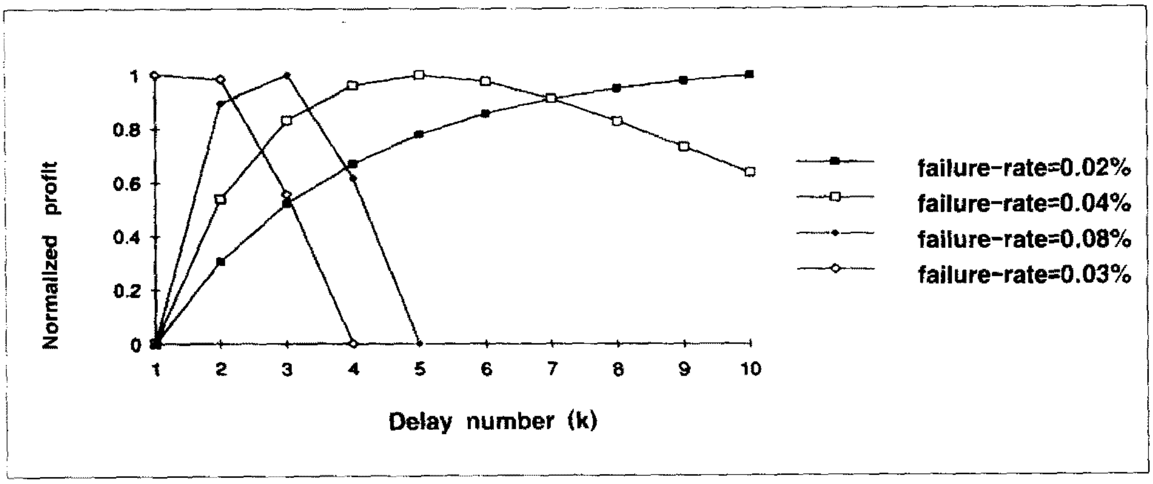
The value of p, factor_of_gain and factor_of_loss are dependent on the application. If $p=2$, $factor\_of\_gain=1$ and $factor\_of\_loss=500$, from Table 1 Figures 7, 8 and 9 are obtained for $\rho=0.30, 0.35$ and 0.40 respectively. As an example in Figure 8, if the fault rate is very low, $f=0.02\%$ per unitary time, then the verification process (the computation of the secondary



(Fig. 7) Relation between delay number and profit for different failure rates ($\rho=0.30$)



(Fig. 8) Relation between delay number and profit for different failure rates ($\rho=0.35$)



(Fig. 9) Relation between delay number and profit for different failure rates ($\rho=0.40$)

version of a job in RESO) is not very important. So it is possible to postpone. the secondary version until the primary queue is empty to maximize the profit; if the fault rate is high, for example $f=0.3\%$, the verification process is also very important and the secondary version should be execute immediately after completing the execution of the primary version. For intermediate values of f (i.e. $0.04\% \leq f \leq 0.08\%$), the largest profit is obtained provided the value of k is between 5 and 3. A similar analysis of the results is applicable to Figures 7 and 9, because for a different value of ρ, the delay number k varies for maximizing the profit.

6. Discussion and Conclusions

This paper has presented a time redundant approach in a uniprocessor in which only a very small amount of hardware is required. The proposed method utilizes two different queues in which a primary and a secondary versions of each incoming job are stored. Three methods for scheduling the redundant computation (as required to meet response time requirements in different applications for fault-tolerant computing) have been presented and analyzed. In the first scheduling method, the primary computation is not affected by the provided arrangement, i.e. there is no time delay. However, this approach is not suitable for a processor affected by frequent transient faults as comparison between the two versions of the same job may be performed after a long delay. The second method operates on a totally opposite schedule: the secondary computation is performed immediately after the primary version and the fault can be detected at an earlier stage. The third method utilizes a fault-tolerant schedule according to which it is possible to find an optimal delay (given by k) based on empiric parameters such as cost, the load and the fault rate of the uniprocessor. In this model, the processor does not execute the secondary version of a job until this job is either k jobs behind its primary version or the primary queue is empty. The proposed model takes into account the load of the system, the fault rate, the cost of a delayed response time and the cost of undetected faults.

References

[1] D.K. Pradhan, Fault-Tolerant Computer System Design, Englewood Cliffs, NJ: Prentice Hall, 1996.

[2] A. Avizienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault-Tolerance during Execution," in Proc. Compac 77, pp.149-155, Nov. 1977.

[3] D.P. Siewiorek, "Niche Successes to Ubiquitous Invisibility: Fault-Tolerant Computing, Past, Present, and Future," in Proc. IEEE FTCS-25(Special Issue), pp.26-34, June 1995.

[4] J-C. Laprie, "Dependable Computing: Concepts, Limits, Challenges," in Proc. IEEE FTCS -25(Special Issue), pp.42-57, June 1995.

[5] J.H. Patel and L.Y. Fung, "Concurrent Error Detection in ALU's by Recomputing with Shifted Operands," IEEE Transactions on Computers, Vol.C-31, No.7, pp.589-595, 1982.

[6] J.H. Patel and L.Y. Fung, "Concurrent Error Detection in Multiplier and Divide Arrays," IEEE Transactions on Computers, Vol.C-32, No.4, pp.417-422, 1983.

[7] A.T. Dahbura, K.K. Sabnani and W.J. Henry, "Spare Capacity as Means of Fault Detection and Diagnosis in Multiprocessor Systems," IEEE Transactions on Computers, Vol.38, No.6, pp.881-891, June 1989.

[8] M.M. Bae and B. Bose, "Spare Processor Allocation for Fault Tolerance in Torus-Based Multicomputers," in Proc. IEEE FTCS-26, pp.282-291, June 1996.

[9] K. Hashimoto, T. Tsuchiya and T. Kikuno, "A New Approach to Realizing Fault-Tolerant Multiprocessor Scheduling by Exploiting Implicit Redundancy," in Proc. IEEE FTCS-27, pp.174-183, June 1997.

[10] G.S. Sohi, M. Franklin and K.K. Saluja, "A Study of Time-Redundancy Fault Tolerance Techniques for High-Performance Pipelined Computers," in Proc IEEE FTCS-19, pp. 436-443, 1989.

[11] Y.M. Hsu and E. Swartzlander, "Time Redundant Error Correcting Adders and Multipliers," in Proc. IEEE Workshop on VLSI Systems, pp.247-256, 1992.

[12] S. Kim, "Cost Analysis of Fault Tolerance in a Uniprocessor Computer," in Proc. of the 8th KIPS Fall Conference, Vol.4, No.2, pp.119-123, Oct. 1997.

[13] J.A. Abraham, "Challenges in Fault Detection," in Proc. IEEE FTCS-25(Special Issue), pp. 96-114, June 1995.

[14] S. Kim, et al., "Scheduling Policies for Fault Tolerance in a VLSI Processor," in Proc. IEEE DFT'94, pp.1-9, Oct. 1994.

[15] T.W. Williams and K.P. Parker, "Design for Testability - A Survey," Proceedings of the IEEE, Vol.71, No.1, pp.98-112, 1983.

김 성 수

1982년 서강대학교 전자공학과(공학사)

1984년 서강대학교 전자공학과(공학석사)

1995년 Texas A&M University, 전산학과(공학박사)

1983년~1986년 삼성전자(주) 종합연구소 컴퓨터연구실(주임연구원)

1986년~1996년 삼성종합기술원 수석연구원

1991년~1992년 Texas Transportation Institute 연구원

1993년~1995년 Texas A&M University, 전산학과, T.A.

1996년~현재 아주대학교 정보통신대학 정보및컴퓨터공학부 조교수

1997년~현재 한국정보처리학회, 한국정보과학회 논문지 편집위원

관심분야 : 결함 허용, 멀티미디어 시스템, 성능평가, H/W & S/W Testing, Mobile Computing 등