# A Generalized Forward Recovery Checkpointing Scheme

Ke Huang, Jie Wu, and Eduardo B. Fernandez
Department of Computer Science and Engineering
Florida Atlantic University
Boca Raton, Florida 33431
{huangk, jie, ed}@cse.fau.edu
http://www.cse.fau.edu/{~jie, ~ed}

**Abstract.** We propose a generalized forward recovery checkpointing scheme, with lookahead execution and rollback validation. This method takes advantage of voting and comparison on multiple versions of the executing task. The proposed scheme is evaluated and compared with other existing checkpointing techniques. The processor assignment problem is studied and an optimal processor assignment is identified. Details on how to use this approach for tolerating both software and hardware faults are also discussed.
**Index terms**: *Checkpointing, fault tolerance, forward recovery, reliability analysis*

## 1 Introduction

In recently years, the *checkpointing* method has been studied frequently as one of the most important fault tolerance methods [1], [2], [3], [5], [7]. Basically, this method identifies the correctness of task execution before accepting its results. At the end of a task execution, a *checkpoint* is set and an *acceptance test* (AT) is applied. If the AT is passed, the task execution is accepted as correct and the next task is executed based on the previous result; if the AT fails, the task is rolled back and reexecuted to reach a consistent state. Two common acceptance tests are *voting* and *comparison*. If several versions of a task are executed simultaneously, their results are voted at the checkpoint or compared if only two versions are running.

The main goal of using the checkpointing method is to achieve high system reliability. The checkpointing method can be applied in software, hardware, or the combination of both. In fact, the behaviors of software errors are very different from that of hardware errors. Generally, software failure rates vary from time to time because they are mainly based on their input data, but hardware failure rates increase during the lifetime of hardware. This motivates us to study the details of the checkpointing method in applications exhibiting both hardware and software faults.

One of the main disadvantages of this basic checkpointing method is its need for *backward recovery*, i.e., when the AT fails the task is rolled back and reexecuted; as the next task is waiting for this result, time is wasted. To save

rollback time, several *forward recovery* techniques have been proposed [4], [5], [6] (most of these techniques deal with tolerating hardware faults only). In a forward recovery checkpointing method, a failure is detected by a checkpoint mismatch that triggers a validation step. However, processors continue execution and spares (processors and/or software versions) are used to determine which of the divergent processors and/or software versions are incorrect. This approach uses the fact that when a voting (comparison) fails, there might exist some correct results. By using those correct results, the rollback time can be avoided, i.e., the next task need not be kept waiting for the result of rollback.

We propose here a generalized forward recovery checkpointing scheme, which uses an $n$-version voting as the AT and $m$-version for reexecution (also called $(n, m)$-pattern). Versions here refer to processor versions. The $(n, m)$-pattern proposed in this paper is a generalization of the method proposed by Long, Fuchs, and Abraham [4], which is a $(2, 1)$-pattern, i.e., a 2-version comparison for the AT and one version for reexecution. We evaluate this method and compare it with the basic checkpointing method. We consider the processor assignment problem and find that, after a voting fails the best processor assignment, in terms of achieving high system reliability, is to assign all the used processors to a single cluster. Also, we discuss some details of applying this forward recovery checkpointing method to tolerate software and/or hardware faults, where versions are either software or processor versions.

We consider an application which is a linearly ordered sequence of tasks, i.e., the tasks are executed one after the other, and each task is based on the result of its previous task. Let this application consists of (by order) $T_1$, $T_2$, $T_3$, ..., then $T_{i+1}$ is based on the result of $T_i$. For simplicity, we assume that each task has the same execution time. At the end of the execution of each task, there is a checkpoint. We assume that each version fails independently. Although several versions may generate the same faulty result, the probability of forming a majority of the same faulty result is negligible. A model that covers correlated faults can be found in [8], where a Markov model is used to describe system reliability. Furthermore, we do not set a limit on the number of faults in the given application.

The rest of this paper is organized as follows. In Section 2, we illustrate our method by using the $(3, 1)$-pattern, i.e., 3-versions voting as the AT and one version for reexecution during the validation step. Section 3 compares the proposed scheme with an existing method, and Section 4 shows the results for the general case, i.e., $(n, m)$-pattern. Section 5 proposes an optimal solution for the processor assignment problem. Section 6 discusses some details for tolerating software and/or hardware faults, while Section 7 presents some conclusions.

## 2 Forward Recovery Checkpointing Method

In this section we only consider processor faults. Our forward recovery strategy is lookahead execution with rollback. Initially, copies of a task are executed at different processors. At the checkpoint, we vote (or compare) the results of those

versions. If the voting is successful, we have obtained a correct result. Based on this result, copies of the next task are executed. If, on the other hand, the voting fails, we execute copies of the next task based on the results of the previous task. Simultaneously, a rollback execution of the previous task is implemented, i.e., the previous task is reexecuted on spare processors in order to obtain a correct result. Then, we keep the results of the next task which were based on the correct versions of the previous task and discard the others. In this way, we avoid wasting time to wait for rollback execution. Although waiting for rollback execution cannot be avoided if all the versions fail (all the results are incorrect), or if the reexecution of the previous task does not get a correct result, rollback time is saved by taking advantage of existing correct results even when voting (or comparison) fails.

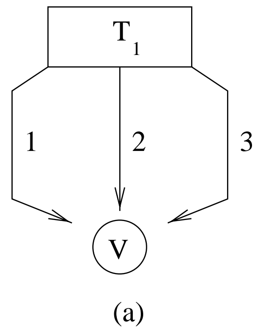
We define an $(n, m)$-pattern as $n$ versions for voting and $m$ versions for reexecution during the validation step. Figure 1 shows a $(3, 1)$-pattern, i.e., 3-version voting and one version reexecution, where we have task $T_1$ followed by task $T_2$. Initially, 3 versions of task $T_1$ are executed, described by arrows labeled 1, 2, and 3 as shown in Figure 1 (a). The results of these versions are voted. If this voting is successful, task $T_2$ starts. Note that, when one version fails, the voting is still successful. On the other hand, if the voting fails, i.e., all results are different, using the three results of $T_1$, three clusters of $T_2$ with three versions in each are executed (which are the left three clusters in Figure 1 (b), where the $T_2^i$ cluster uses the result of the $i$th version of $T_1$) and a reexecution of $T_1$ is carried out at the same time as shown in Figure 1 (b). That is, a total of ten processors are used. Using the reexecution result of $T_1$, we can determine the correct version of $T_1$ (in the last time step) and discard the others. If we can find out one correct result of $T_1$ at the end of the execution in Figure 1 (b); in other words, if the reexecution of $T_1$ is correct and not all the first three results of $T_1$ are incorrect, we get Figure 1 (c), where $T_2^*$ is one of the three clusters $T_2^1$, $T_2^2$, and $T_2^3$, i.e., the version of $T_2$ executed based on the correct results of $T_1$. We treat the configuration in Figure 1 (c) just as if the voting among the first three results of $T_1$ has been successful. Thus, if such a $T_2^*$ exists, the rollback time is saved although the voting among the first three results of $T_1$ failed. In this way, after a voting fails we still achieve forward recovery by using the correct result from one version of $T_1$.

This forward-recovery method is evaluated in the subsequent sections. But first we introduce several performance measures in the next section.

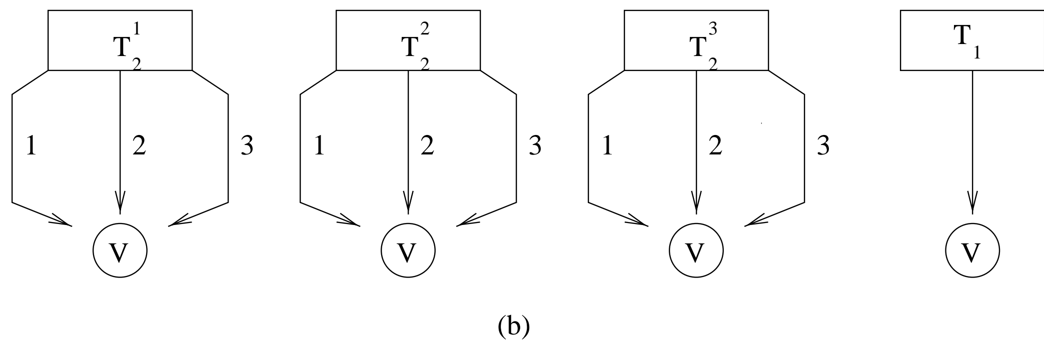
## 3 Comparisons

To achieve fault tolerance some redundant resources are used. Since a version refers to a processor, a first measure is the number of processors used in the proposed scheme:

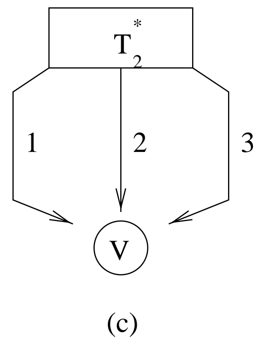
- $N_p$: the maximum number of processors used during a time slice;
- $\overline{N}_p$: the average number of processors used during a time slice.



(a)



(b)



(c)

**Fig. 1.** The forward recovery checkpointing method: $(3, 1)$-pattern.

Another important measure is the number of checkpoints:

- $N_c$: the maximum number of checkpoints needed;
- $\overline{N}_c$: the average number of checkpoints needed.

Other measures are related to time:

- $T$: the expected execution time.
- $R_t$: the ratio of $T$ over the fault-free execution time.

To simplify our discussion, we assume that the failure rate for each processor has the same value $p$, where $0 < p < 1$.

We evaluate our scheme by studying the 3-version voting and one version reexecution, i.e., a $(3, 1)$-pattern. The evaluation of general patterns is discussed in the next section. Again, the execution time for every version of any task takes the same time $t$, which also includes checkpointing time. We assume that the preparation time for reexecution, also called recovery time, can be ignored.

Clearly, the voting is successful if and only if at least two out of the three versions are correctly executed. Thus, the probability of successful voting is:

$$\begin{aligned} p_{voting-succ}(3,1) &= (1-p)^3 + 3p(1-p)^2 \\ &= 1 - 3p^2 + 2p^3. \end{aligned}$$

When the voting fails, the rollback can be avoided if and only if there existed one correct result out of the three results of the first task and its reexecution is correct. So, the probability of successful forward recovery with voting failure is:

$$\begin{aligned} p_{forward-succ}(3,1) &= (1-p) * 3p^2(1-p) \\ &= 3p^2 - 6p^3 + 3p^4. \end{aligned}$$

If all 3 versions fail, or the reexecution fails, rollback execution cannot be avoided. So, the probability of successful forward recovery is:

$$\begin{aligned} p_{succ}(3,1) &= 1 - (p^3 + p * (3p^2(1-p))) \\ &= 1 - 4p^3 + 3p^4. \end{aligned}$$

Then, the probability of forward recovery failure is:

$$\begin{aligned} p_{fail}(3,1) &= 1 - p_{succ}(3,1) \\ &= 1 - (1 - 4p^3 + 3p^4) \\ &= 4p^3 - 3p^4. \end{aligned}$$

Notice that:

$$p_{voting-succ}(3,1) + p_{forward-succ}(3,1) = p_{succ}(3,1).$$

If the voting fails, ten processors are needed (three for each of the three clusters, another for the reexecution the previous task). But only three processors are used if the voting is successful. Thus, the number of processors are:

$$\begin{aligned}
N_p(3,1) &= 10; \\
\overline{N}_p(3,1) &= 3 * p_{voting-succ}(3,1) + 10 * (1 - p_{voting-succ}(3,1)) \\
&= 10 - 7 * p_{voting-succ}(3,1) \\
&= 10 - 7(1 - 3p^2 + 2p^3) \\
&= 3 + 21p^2 - 14p^3.
\end{aligned}$$

One cluster has three checkpoints and ten more checkpoints are required if the voting fails. Thus, the numbers of checkpoints are:

$$\begin{aligned}
N_c(3,1) &= 13; \\
\overline{N}_c(3,1) &= 3 * p_{voting-succ}(3,1) + 13 * (1 - p_{voting-succ}(3,1)) \\
&= 13 - 10 * p_{voting-succ}(3,1) \\
&= 13 - 10(1 - 3p^2 + 2p^3) \\
&= 3 + 30p^2 - 20p^3.
\end{aligned}$$

Notice that the rollback time takes $2t$ (i.e., two intervals of checkpointing) when the forward recovery fails. Then, for the time measures, we have:

$$\begin{aligned}
T(3,1) &= \sum_{0<k<\infty} (2k+1)t * p_{succ}(3,1)(1 - p_{succ}(3,1))^k \\
&= (1 + \frac{2(1 - p_{succ}(3,1))}{p_{succ}(3,1)})t \\
&= (1 + \frac{2p^3(4 - 3p)}{1 - 4p^3 + 3p^4})t; \\
R_t(3,1) &= \frac{T(3,1)}{t} \\
&= 1 + \frac{2p^3(4 - 3p)}{1 - 4p^3 + 3p^4}.
\end{aligned}$$

To compare our scheme with the existing ones (in [4] and [6]), i.e., $(3,1)$-pattern v.s. $(2,1)$-pattern, we need the results for the $(2,1)$-pattern. Notice that,

$$\begin{aligned}
p_{comparison-succ}(2,1) &= 1 - 2p + p^2; \\
p_{succ}(2,1) &= 1 - 3p^2 + 2p^3.
\end{aligned}$$

Therefore,

$$\begin{aligned}
N_p(2,1) &= 5; \\
\overline{N}_p(2,1) &= 2 * p_{comparison-succ}(2,1) + 5 * (1 - p_{comparison-succ}(2,1)) \\
&= 5 - 3 * p_{comparison-succ}(2,1) \\
&= 5 - 3(1 - 2p + p^2)
\end{aligned}$$

$$
\begin{aligned}
&= 2 + 6p - 3p^2; \\
N_c(2,1) &= 7; \\
\overline{N}_c(2,1) &= 2 * p_{comparison-succ}(2,1) + 7 * (1 - p_{comparison-succ}(2,1)) \\
&= 7 - 5 * p_{comparison-succ}(2,1) \\
&= 7 - 5(1 - 2p + p^2) \\
&= 2 + 10p - 5p^2; \\
T(2,1) &= (1 + \frac{2(1 - p_{succ}(2,1))}{p_{succ}(2,1)}) * t \\
&= (1 + \frac{2p^2(3-2p)}{1 - 3p^2 + 2p^3}) * t; \\
R_t(2,1) &= \frac{T(2,1)}{t} \\
&= 1 + \frac{2p^2(3-2p)}{1 - 3p^2 + 2p^3}.
\end{aligned}
$$

For the basic backward recovery the rollback time is $t$ (i.e., one interval of checkpointing). Thus, the expected execution time of the basic checkpointing method is:

$$
\begin{aligned}
T_{basic} &= \sum_{0<k<\infty} kp(1-p)^{k-1} \\
&= (1 + \frac{p}{1-p}) * t.
\end{aligned}
$$

Notice that the failure rate $p$ of each version is also the probability of rollback for the basic checkpointing method. Figure 2 shows a comparison of the failure rates of the basic method, the $(2,1)$-pattern, and the $(3,1)$-pattern, where the failure rates are functions of $p$. The basic method uses only one version and no fault tolerance method. This figure shows that the $(3,1)$-pattern always has smaller failure rates than the $(2,1)$-pattern (except for the trivial cases $p = 0$ or 1), i.e., the following expression holds:

$$
p_{fail}(2,1) > p_{fail}(3,1) \qquad (p \neq 0, 1)
$$

Moreover, the failure rate of the $(2,1)$-pattern is smaller than that of the basic method when $p$ is small (in fact, when $p < 0.5$), but larger when $p$ is large (in fact, when $p > 0.5$). In other words, the $(2,1)$-pattern improves the basic method only when the system uses versions with small failure rates. However, the $(3,1)$-pattern improves the basic method until $p$ becomes 0.8. Thus, as a conclusion, the $(3,1)$-pattern outperforms the $(2,1)$-pattern in terms of system reliability. As a whole, however, the $(3,1)$-pattern costs more than the $(2,1)$-pattern. The $(3,1)$-pattern is a better replacement scheme for the basic method than the $(2,1)$-pattern, when $p$ is relatively large, especially within the range between 0.5 and 0.8. Figures 3 through 6 show comparisons of expected execution times between the basic method, the $(2,1)$-pattern, and the $(3,1)$-pattern, where the expected

execution times are functions of $p$ with $t = 1$. In Figure 2, $p$ ranges from 0 to 1. More detailed information is shown in Figures 3, 4, 5, and 6. From these figures, we observe that when $p$ is small (in fact, when $p < 0.22$), the $(2, 1)$-pattern improves the basic method, but the $(3, 1)$-pattern is the best among the three; when $p > 0.22$, the $(2, 1)$-pattern no longer improves the basic method, but the $(3, 1)$-pattern still does until $p > 0.53$; but after $p$ is larger than 0.53, neither the $(2, 1)$-pattern nor the $(3, 1)$-pattern improve the basic method. The $(3, 1)$-pattern is always better than the $(2, 1)$-pattern in terms of shorter expected execution time.

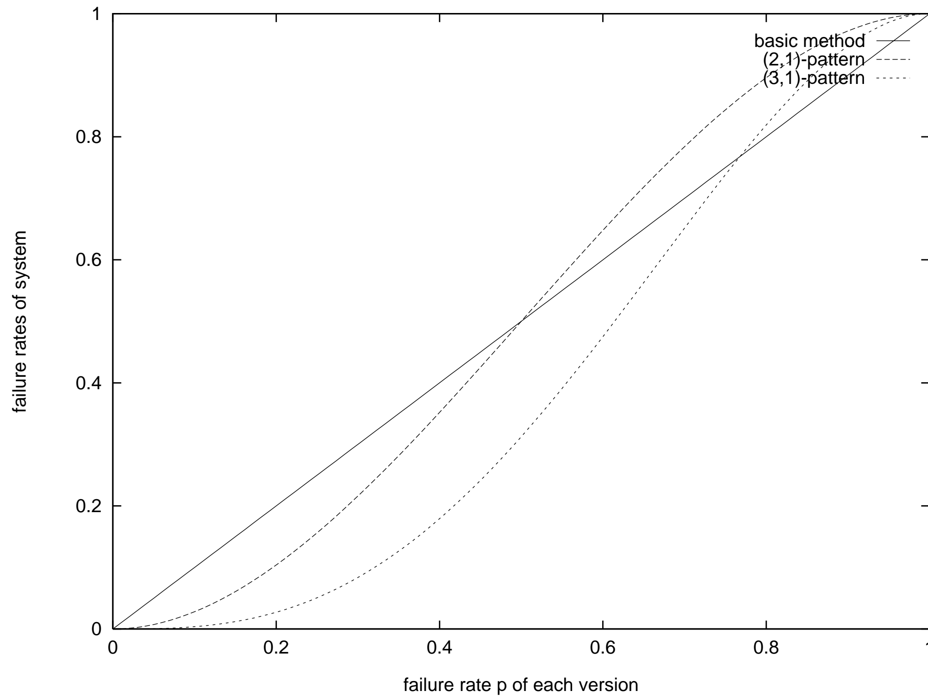



**Fig. 2.** Failure rate as a function of $p$.

The cost to achieve smaller system failure rate and better performance is a larger number of processors and checkpoints. Figures 7 and 8 show comparisons of average number of processors and number of checkpoints between the $(2, 1)$-pattern and the $(3, 1)$-pattern. Note that for certain ranges of failure rate $p$, around $p = 0.2$, the average number of processors (and checkpoints) for the (3,1)-pattern is very close to the one for (2,1)-pattern. The reason is that a vote has a much higher success probability than a comparison. Therefore, the 10-processor activation in the (3,1)-pattern occurs less frequent that the 5-processor

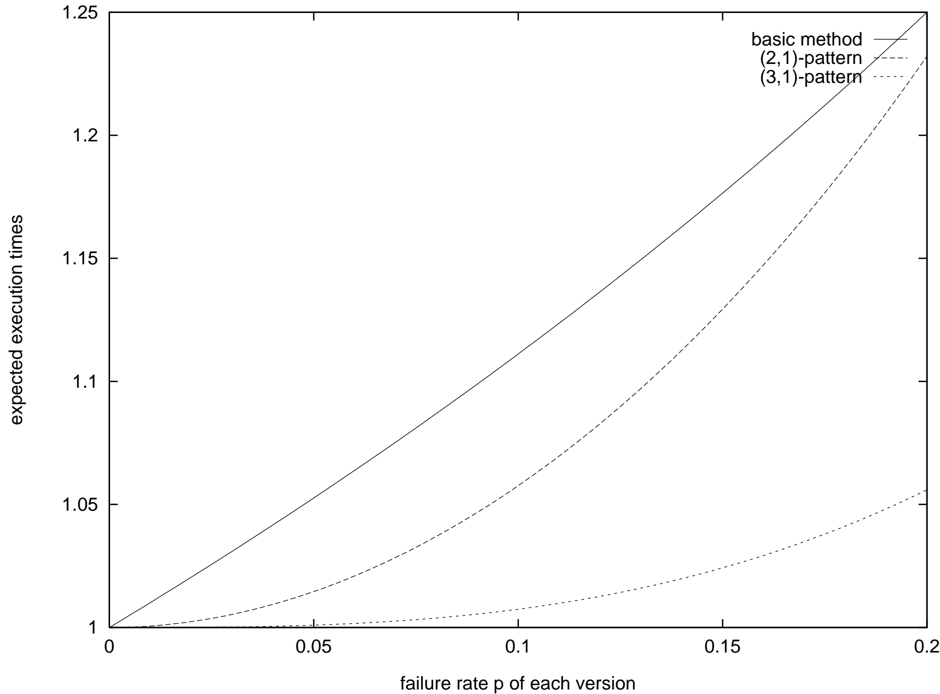



**Fig. 3.** Expected execution time as a function of $p$ (range 0 - 0.2).

activation in the (2,1)-pattern.

Figure 9 shows the ratio comparisons of average number of processors and number of checkpoints between the $(2,1)$-pattern and the $(3,1)$-pattern. Clearly, these ratios reach a minimum at around $p = 0.18$. At that point, $\bar{N}_p(3,1)/\bar{N}_p(2,1)$ $\approx$ 1.2, i.e., the (3,1)-pattern requires about 20% more processor resources than the (2,1)-pattern. On the other hand, based on Figure 3 the ratio between the (2,1)-pattern and the (3,1)-pattern in terms of the expected execution time is 1.22/1.03=1.18, i.e., the (3,1)-pattern has about a 16% reduction in the expected execution time compared to the (2,1)-pattern.

## 4 Extensions

We discuss our method by considering a special case, the $(3,1)$-pattern, i.e., 3 versions for voting and one version for reexecution. If we use $n$ versions for voting and $m$ versions for reexecution when the voting fails (where these $m$ reexecution versions vote for the reexecution result), we then have the general $(n,m)$-pattern. We derive the performance measures for the $(n,m)$-pattern in this section.

Again, we assume that the execution time for any task is the same time $t$

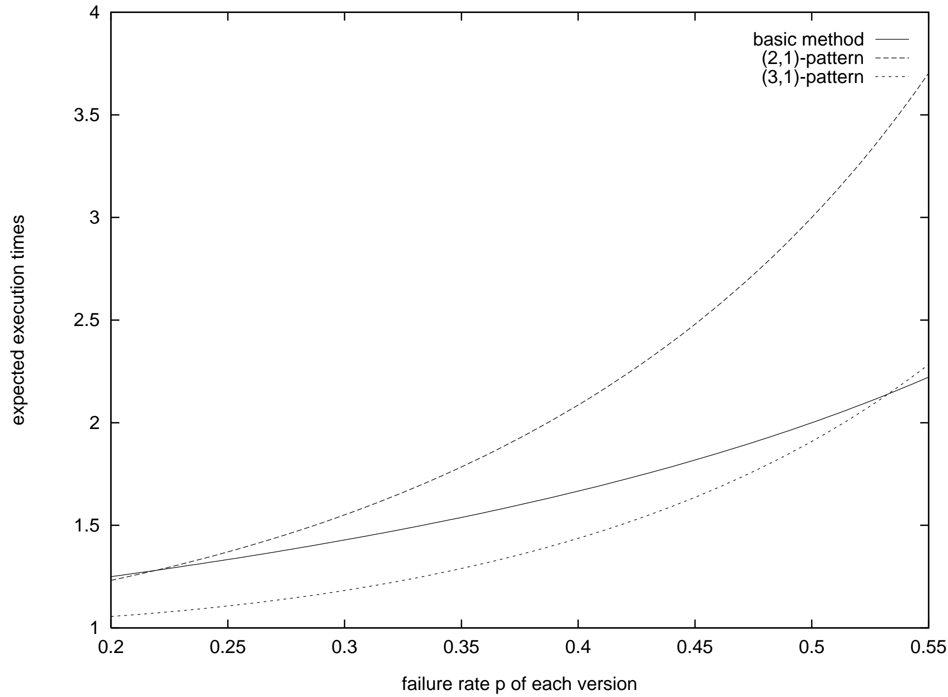



**Fig. 4.** Expected execution time as a function of $p$ (range 0.2 - 0.55).

and the probability of correct execution for each processor has the same value $p$. The preparation time for task reexecution is again ignored.

When the voting fails, rollback can be avoided if and only if there exist correct results among the $n$ versions of the first task and we can identify at least one of them. In other words, when the voting fails, we avoid rollback if and only if not all the $n$ versions of the first task fail and more than half of the $m$ reexecution versions of it are correctly reexecuted. Thus, the probabilities of successful voting, of successful forward recovery with voting failure, and of successful forward recovery are:

$$p_{voting-succ}(n,m) = \sum_{\lceil \frac{n}{2} \rceil \leq k \leq n} C_n^k p^{n-k}(1-p)^k;$$

$$p_{forward-succ}(n,m) = (\sum_{1 \leq k \leq \lceil \frac{n}{2} \rceil - 1} C_n^k p^{n-k}(1-p)^k) * (\sum_{\lceil \frac{m}{2} \rceil \leq k \leq m} C_m^k p^{m-k}(1-p)^k);$$

$$\begin{aligned} p_{succ}(n,m) &= p_{voting-succ}(n,m) + p_{forward-succ}(n,m) \\ &= \sum_{\lceil \frac{n}{2} \rceil \leq k \leq n} C_n^k p^{n-k}(1-p)^k + \end{aligned}$$

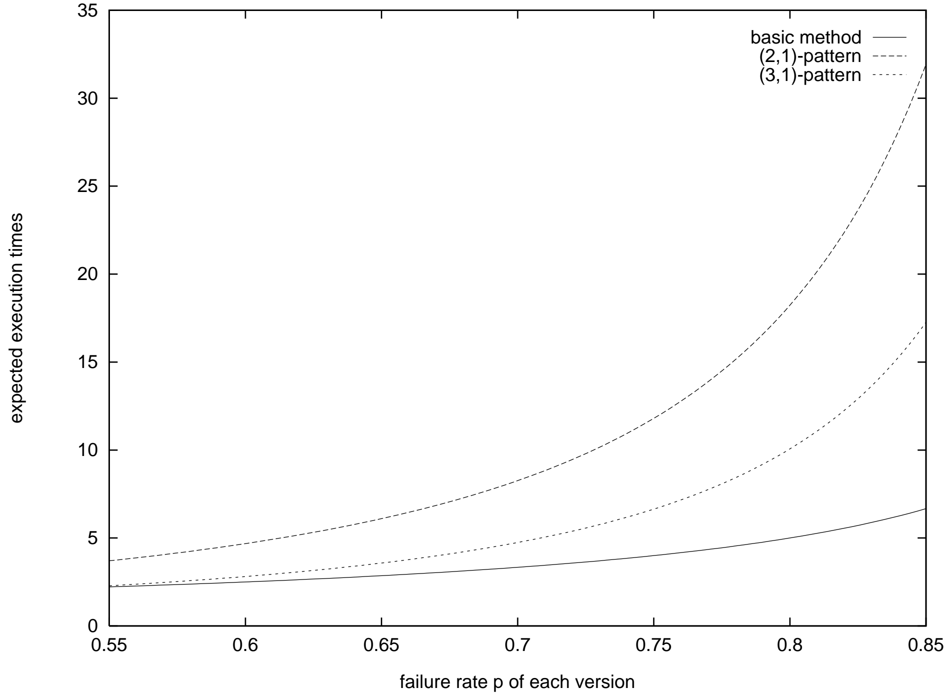



**Fig. 5.** Expected execution time as a function of $p$ (range 0.55 - 0.85).

$$(\sum_{1 \le k \le \lceil \frac{n}{2} \rceil - 1} C_n^k p^{n-k}(1-p)^k) * (\sum_{\lceil \frac{m}{2} \rceil \le k \le m} C_m^k p^{m-k}(1-p)^k).$$

Only $n$ processors are used if the voting is successful. But after the voting fails, at least $m + n^2$ processors are required: $m$ processors for reexecuting the first task, and $n$ for each of the $n$ clusters executing the second task version. That is, $m + n^2 - n$ more processors are needed after the voting fails. Thus, the numbers of processors used are:

$$\begin{aligned} N_p(n,m) &= m + n^2; \\ \overline{N}_p(n,m) &= n * p_{voting-succ}(n,m) + (m + n^2) * (1 - p_{voting-succ}(n,m)) \\ &= n + (n^2 + m - n) * (1 - p_{voting-succ}(n,m)) \\ &= n + (n^2 + m - n) \sum_{0 \le k \le \lceil \frac{n}{2} \rceil} C_n^k p^{n-k}(1-p)^k. \end{aligned}$$

Notice that each cluster of a task version normally has $n$ checkpoints, but the reexecution cluster has $m$. Therefore, the numbers of checkpoints for the

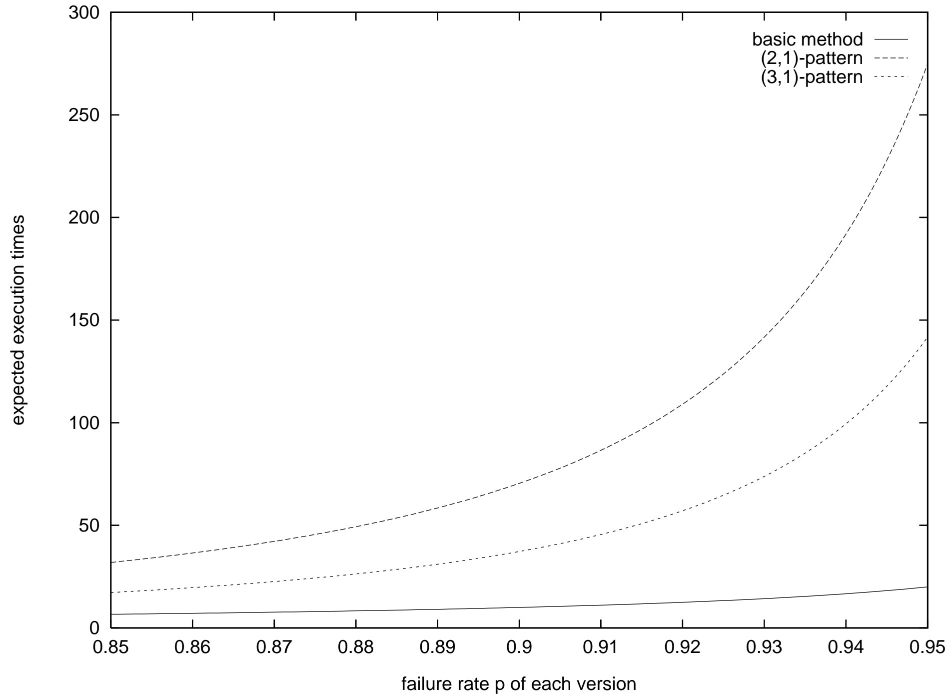



**Fig. 6.** Expected execution time as a function of $p$ (range 0.85 - 0.95).

$(n, m)$-pattern are:

$$
\begin{aligned}
N_c(n,m) &= m + n + n^2; \\
\overline{N}_c(n,m) &= n * p_{voting-succ}(n,m) + (m + n + n^2) * (1 - p_{voting-succ}(n,m)) \\
&= n + (n^2 + m) * (1 - p_{voting-succ}(n,m)) \\
&= n + (n^2 + m) \sum_{0 \le k \le \lceil \frac{n}{2} \rceil} C_n^k p^{n-k} (1-p)^k.
\end{aligned}
$$

The rollback time for the $(n, m)$-pattern is $2t$ (i.e., two intervals of check-

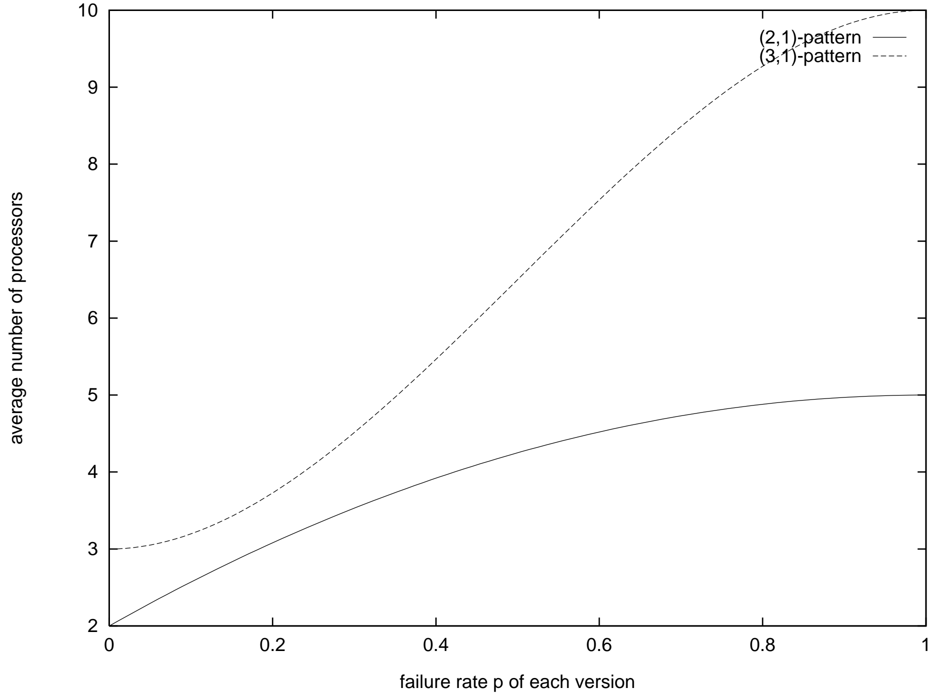



**Fig. 7.** Average number of processors as a function of $p$.

pointing), and the measures that are related to time are:

$$
\begin{aligned}
&T(n,m)\\
&\quad = \sum_{0\le k} k(2k+1)t * p_{succ}(n,m)(1-p_{succ}(n,m))^k\\
&\quad = (1+\frac{2(1-p_{succ})}{p_{succ}})t\\
&\quad = (1+\frac{2(p^n+(1+\sum_{\lceil\frac{m}{2}\rceil\le k\le m} C_m^k p^{m-k}(1-p)^k)*\sum_{1\le k\le\lceil\frac{n}{2}\rceil-1} C_n^k p^{n-k}(1-p)^k)}{\sum_{\lceil\frac{n}{2}\rceil\le k\le n} C_n^k p^{n-k}(1-p)^k+(\sum_{1\le k\le\lceil\frac{n}{2}\rceil-1} C_n^k p^{n-k}(1-p)^k)*(\sum_{\lceil\frac{m}{2}\rceil\le k\le m} C_m^k p^{m-k}(1-p)^k)})t;\\
&R_t(n,m)\\
&\quad = \frac{T(n,m)}{t}\\
&\quad = 1+\frac{2(p^n+(1+\sum_{\lceil\frac{m}{2}\rceil\le k\le m} C_m^k p^{m-k}(1-p)^k)*\sum_{1\le k\le\lceil\frac{n}{2}\rceil-1} C_n^k p^{n-k}(1-p)^k)}{\sum_{\lceil\frac{n}{2}\rceil\le k\le n} C_n^k p^{n-k}(1-p)^k+(\sum_{1\le k\le\lceil\frac{n}{2}\rceil-1} C_n^k p^{n-k}(1-p)^k)*(\sum_{\lceil\frac{m}{2}\rceil\le k\le m} C_m^k p^{m-k}(1-p)^k)}.
\end{aligned}
$$

Using common sense, it is clear that with larger $m$, as well as larger $n$, the $(n,m)$-pattern improves the system reliability and the expected execution time of the basic method. However, more processors and checkpoints are needed. Figures 10 compares the failure rates and the expected execution times among the $(3,1)$-pattern, the $(3,2)$-pattern, and the $(3,3)$-pattern, where failure rates and execution times are functions of $p$.

The method discussed above is forward recovery for only two units of time,

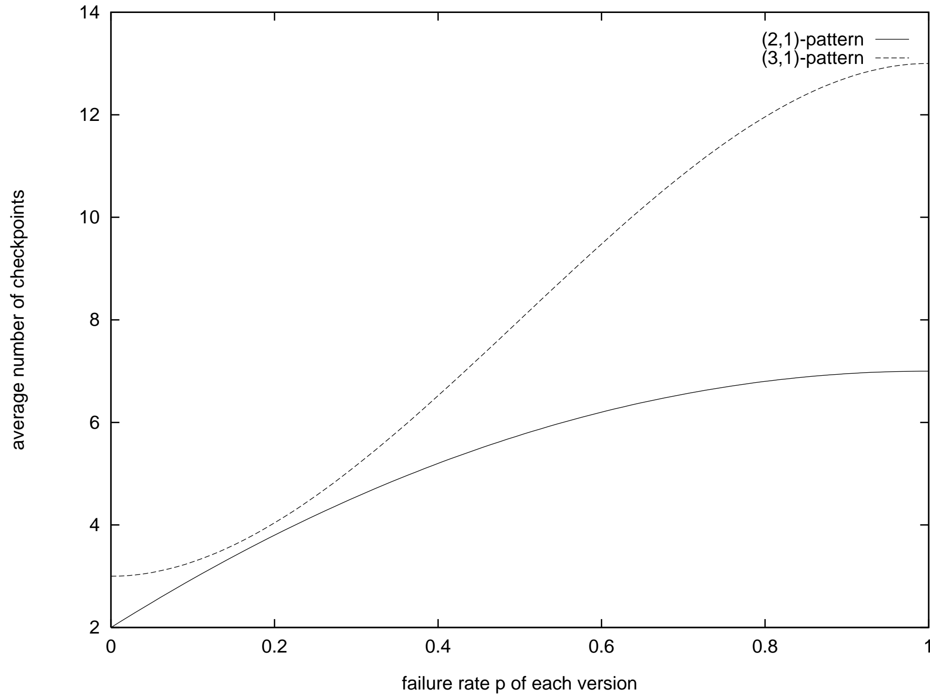



**Fig. 8.** Number of checkpoints as a function of $p$.

i.e., rollback happens when reexecution of the previous task $T_1$ does not match any existing results of it after the next task, $T_2$, execution. If we try reexecution of $T_1$ again after $T_2$, i.e., when task $T_3$ is being executed, then we have to rollback three units of time. But the advantage of this strategy is to make use of the correct result of $T_1$ more efficiently than that of the $(n, m)$-pattern discussed here. However, many more versions are needed. By introducing recovery length $l$, we can further generalize our method into a $(n, m, l)$-pattern. Again, the details of such an extension are the subject of future research.

## 5 Processor Assignment

A forward recovery checkpointing scheme to tolerate hardware faults has been discussed in Sections 2 and 3. This section discusses a non-trivial problem called the processor assignment problem.

Let's consider a system that consists of processors of the same type, i.e., all processors have the same failure rate. The failure rate increases during the lifetime of the processor. We assume that the failure rates for new and used processors are $p$ and $q$, respectively. Then, $p \leq q$.

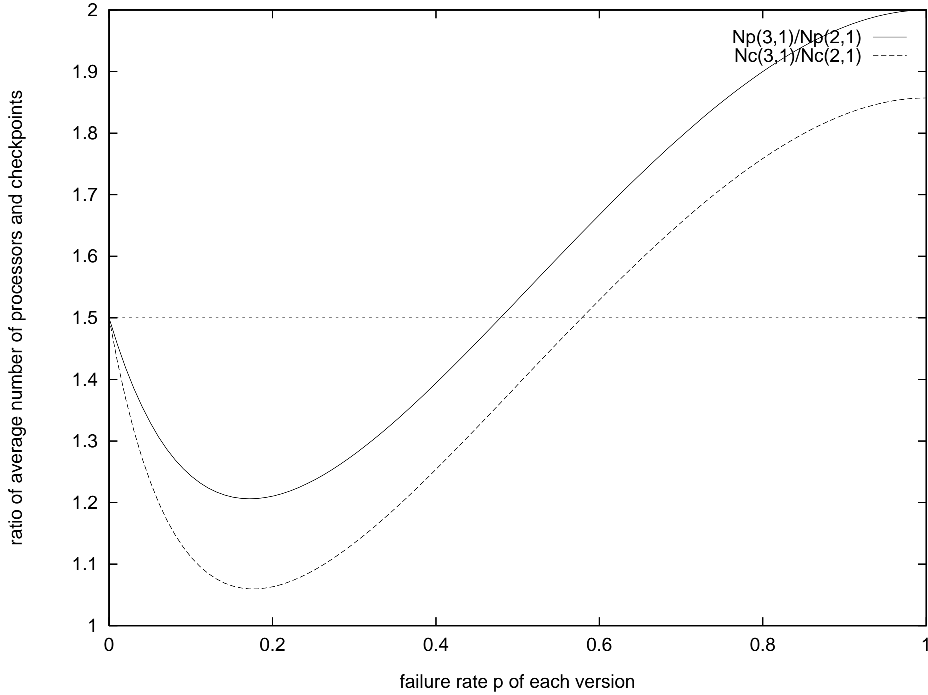



**Fig. 9.** Ratio of $(3,1)$-pattern over $(2,1)$-pattern for number of processors and checkpoints.

Now, let's examine the processor assignment problem as follows: Initially, $n$ processors run $n$ versions of the first task $T_1$. At the checkpoint, i.e., the end of $T_1$, these $n$ results from these $n$ different processors are voted. If a majority voting exists, we can find out all the faulty processors (if there are any). Thus we replace those faulty processors with new ones, and these $n$ processors continue to execute versions of the next task $T_2$ based on the voting result of $T_1$. Otherwise, if the voting is unsuccessful, $m+n^2-n$ new processors join for the next step, i.e., a total of $m+n^2$ processors ($n$ old ones and $m+n^2-n$ new ones) are running. Based on each of the $n$ results of $T_1$, $n$ processors run versions of $T_2$, and the remaining $m$ processors reexecute versions of $T_1$. These $m+n^2$ processors are not the same in term of failure rate. In fact, those $n$ used ones have failure rate $q$, but the $m+n^2-n$ new ones have failure rate $p$, which is smaller than $q$. Now, we have the following processor assignment problem: how to assign these $m+n^2$ processors? Notice that a rollback happens if and only if none of the first $n$ results of $T_1$ is correct or the voting for reexecution of $T_1$ fails after the first voting fails. Thus, to achieve larger probability of successful forward recovery, we need to assign $m$ new processors to reexecute $T_1$. Now, we should divide the remaining $n^2$ processors ($n$ used ones and $n^2-n$ new ones) into $n$ clusters with

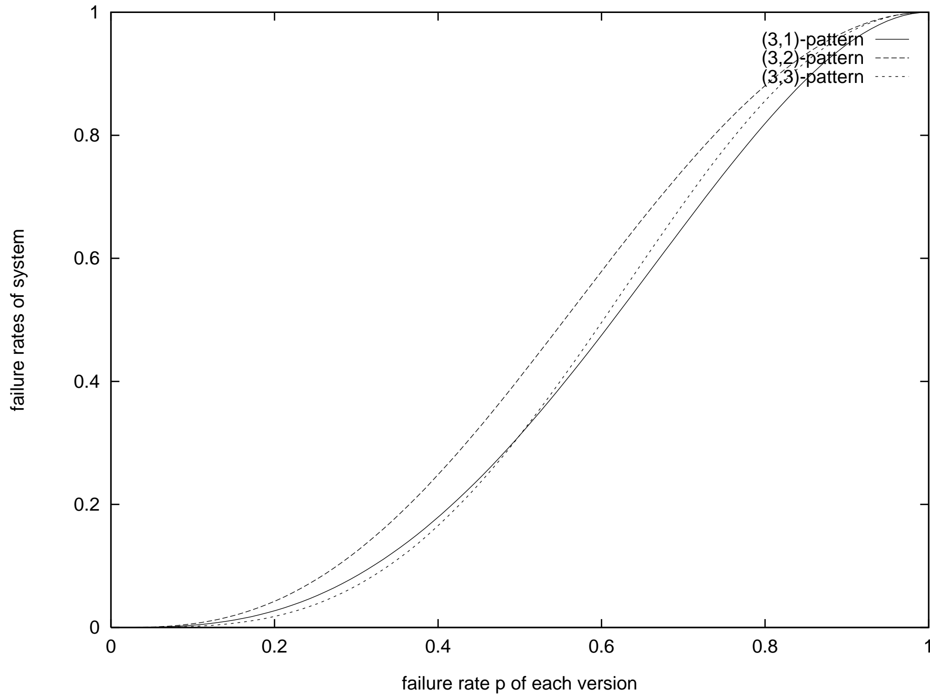



**Fig. 10.** Comparing system failure rates among different patterns.

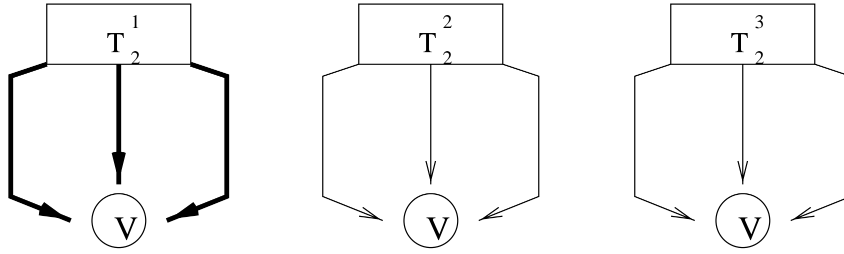
$n$ processors in each to execute the next task $T_2$. Our goal here is to achieve a high probability of correctly executing $T_2$.

To illustrate the point, we again use the $(3,1)$-pattern as an example. As shown in Figure 11, there are three possible assignments, (a), (b), and (c) to divide those 9 processors (3 used ones, and 6 new ones) into 3 clusters with each containing 3 processors:

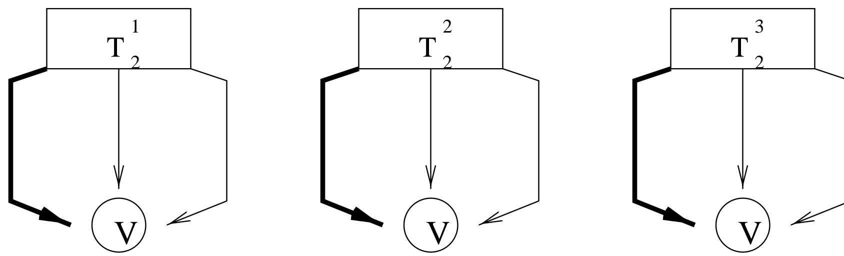
(a) 3 used processors are contained in one cluster, i.e., one cluster has 3 used processors and the rest of the clusters have 6 new processors, with 3 in each;
(b) 3 used processors are distributed to each of the 3 clusters, i.e., every cluster consists of one used processor and two new ones;
(c) 3 used processors are distributed into two clusters, i.e., one cluster has two used processors and one new one, one cluster has one used one and two new ones, and another cluster has three new ones.

The probabilities for correct voting after $T_2$ execution are defined as $p(a)$, $p(b)$, and $p(c)$, corresponding to the above three schemes (a), (b), and (c). For simplicity, we need to compare only the probabilities that are based on results from correct versions of $T_1$. Then, we have:

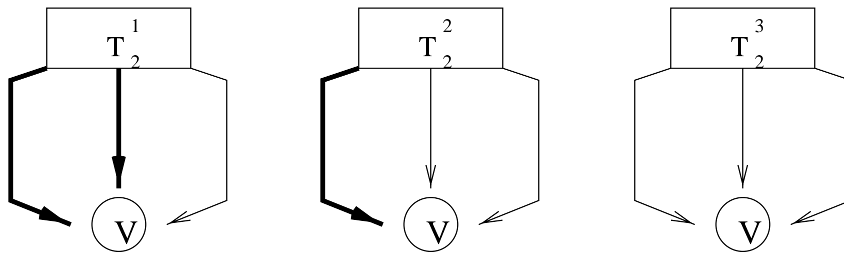
$$p(a) = 1 - (q^3 + 3q^2(1-q))(p^3 + 3p^2(1-p))^2$$




(a)




(b)




(c)

**Fig. 11.** 3 ways to assign 9 processors for $T_2$ (bold lines indicate that used processors are running).

$$
\begin{aligned}
&= 1-(3q^2-2q^3)(3p^2-2p^3)^2\\
&= 1-q^2p^4(3-2q)(3-2p);\\
p(b) &= 1-(p^2+2qp(1-p))^3\\
&= 1-(p^2+2qp-2qp^2)^3\\
&= 1-p^3(p+2q-2qp)^3;\\
p(c) &= 1-(q^2+2q(1-q)p)(p^2+2qp(1-p))(p^3+3p^2(1-p))\\
&= 1-(q^2+2qp-2q^2p)(p^2+2qp-2qp^2)(3p^2-2p^3)\\
&= 1-qp^3(q+2p-2qp)(p+2q-2pq)(3-2p).
\end{aligned}
$$

Because,

$$
\begin{aligned}
p(a)-p(c) &= qp^3(3-2p)((q+2p-2qp)(p+2q-2pq)-qp(3-2q)(3-2p))\\
&= qp^3(3-2p)(2p^2+2q^2-4pq)\\
&= 2qp^3(3-2p)(q-p)^2>0;\\
p(c)-p(b) &= p^3(p+2q-2pq)((p+2q-2pq)^2-q(q+2p-2qp)(3-2p))\\
&= p^3(p+2q-2pq)(p^2+q^2-2pq)\\
&= p^3(p+2q-2pq)(q-p)^2>0,
\end{aligned}
$$

we have

$$p(a)>p(c)>p(b).$$

Therefore, assigning all the used processors to a single cluster is the best approach.

Following the idea used in the above proof, we can prove a general conclusion for the $(n,m)$-pattern.

**Theorem**: *After a voting fails, the best processor assignment, in terms of achieving high system reliability, is to assign all the used processors to a single cluster.*

In other words, after the voting among versions of $T_1$ fails, to place all the used processors in one cluster that runs $T_2$ is the best way to achieve high probability of correct execution of $T_2$. Note that the definition of used and new processors is relative. It might happen that "new" processors themselves have been used earlier and they are "older" than the used processors. In this case, the above theorem should be interpreted as: Assign the similarly "aged" processors to the same cluster.

## 6 Tolerating Software/Hardware Faults

So far, we have discussed only hardware fault tolerance, and we assumed that the software is fault free. However, in the real world, the software might also fail. The behavior of software errors is different from that of hardware errors. Generally, software failure rates vary from time to time, while hardware failure rates increase during the lifetime of the hardware.

In our earlier discussion, the task versions are the same software running on several processors. Thus, our forward recovery checkpointing method can be used in the following three environments under different fault assumptions:

1. Hardware fault tolerance
   The software is fault free. The fault-free task versions are running on a set of processors, which are either healthy or faulty.
2. Software fault tolerance
   The hardware is fault free. The task versions (some of them might be faulty) are running on some fault-free processors.
3. Software/hardware fault tolerance
   Both software and hardware versions can be faulty.

Now, we consider using our forward recovery checkpointing method for software fault-tolerance. To achieve software fault-tolerance, different software versions of a task are executed by a set of fault-free processors. Semantically, a software fault exhibits the same behavior as a hardware fault in the forward recovery checkpointing scheme. Therefore, we can use the same approach used in achieving hardware fault tolerance for software fault tolerance, i.e., the expressions for system failure rate and execution time derived in Section 4 still hold. One additional measure needed is the number of software versions that are required:

- $n_v$: maximal number of software versions needed for each task;
- $\overline{n}_v$: the expected number of software versions needed for each task.

When the first voting fails, some software versions of the first task are incorrect. As we proposed before, $n$ clusters of the next task are running based on the $n$ results of the first task, and $m$ versions of the first task are reexecuted. Thus $n + m$ software versions of the first task are needed. For the next task, each of the $n$ clusters has $n$ different software versions, but the versions can be replicated at different clusters. Thus $n$ software versions are enough for the $n$ different clusters. Of course, $m$ additional software versions are needed for fault identification. Based on the results in Section 4 for the $(n, m)$-pattern, we have the following conclusion:

$$\begin{aligned}
n_v(n,m) &= m + n; \\
\overline{n}_v(n,m) &= n * p_{voting-succ}(n,m) + (n+m) * (1 - p_{voting-succ}(n,m)) \\
&= n + m - m * p_{voting-succ}(n,m) \\
&= n + m - m * \sum_{\lceil \frac{n}{2} \rceil \le k \le n} C_n^k p^{n-k} (1-p)^k.
\end{aligned}$$

We can also use our forward recovery checkpointing method to achieve software and hardware fault tolerance. Again, we consider the $(n, m)$-pattern. We define a task version as a software version of a task running on a processor, i.e., two task versions are identical if and only if they are the identical software version of a task running on identical processors. Now, we need to answer the following

question: how many software versions and processors are needed? Again, let $n_v$ ($\overline{n}_v$) and $n_p$ ($\overline{n}_p$) be the maximal (expected) numbers of software versions and processors needed for each task respectively, then we can drive as many as $n_v * n_p$ ($\overline{n}_v * \overline{n}_p$) different versions of each task in the worst case (average). At first, $n$ versions of the first task are executed. If, however, the voting for the $n$ results of the first task fails, a total of $n^2 + m$ versions ($n^2$ versions of the next task and $m$ versions reexecuting the first task) are executed simultaneously. Thus, we have:

$$\begin{aligned} n_p(n,m) &= m + n^2; \\ \overline{n}_p(n,m) &= n * p_{voting-succ}(n,m) + (m + n^2) * (1 - p_{voting-succ}(n,m)) \\ &= n + (n^2 + m - n) * (1 - p_{voting-succ}(n,m)). \end{aligned}$$

This is the same result as for tolerating hardware faults only. Furthermore, the number of software versions of each task does not matter for using our forward recovery checkpointing scheme. As an extreme case, if we have only one software version of a task, we cannot tolerate software faults. But we can still use our forward recovery scheme to tolerate hardware faults. To tolerate software faults, multiple software versions are needed. Of course, we can assign one software version to several different processors to form several different task versions. But this approach has a high system failure rate. A single faulty software version will result in failure of several task versions, hence the voting at the end of this cluster has a high probability of failure. In the worst case, it might invalidate the assumption that the probability of forming a majority of the same faulty result is negligible. Therefore, to achieve high system reliability, it is better to assign different software versions to different processors, i.e., many software versions are needed for each task. As we discussed above for tolerating software faults only, $m + n$ different software versions for each task are enough. In other words, we cannot achieve higher system reliability by providing more than $m + n$ software versions for each task in our $(n, m)$-pattern forward recovery checkpointing scheme, i.e.,

$$n_v = m + n.$$

Let $p_p$ ($p_v$) be the failure rate for a single processor (software version) and assume that the processor and software failures be independent, then the failure rate for every task version (a software version running on a processor) is $p = p_p * p_v$. Thus, if we have $m + n$ different software versions for each task, the expressions for system failure rate and time measurement derived in Section 4 still hold after replacing the version (processor) failure rate $p$ by the task version failure rate $p_p * p_v$, and

$$\begin{aligned} \overline{n}_p(n,m) &= n * p_{voting-succ}(n,m) + (n^2 + m) * (1 - p_{voting-succ}(n,m)) \\ &= n + (n^2 + m - n) * (1 - p_{voting-succ}(n,m)) \\ &= n + (n^2 + m - n) * \sum_{0 \le k < \lceil \frac{n}{2} \rceil} C_n^k (p_p * p_v)^{n-k} (1 - p_p * p_v)^k; \\ \overline{n}_v(n,m) &= n * p_{voting-succ}(n,m) + (n + m) * (1 - p_{voting-succ}(n,m)) \\ &= n + m - m * p_{voting-succ}(n,m) \end{aligned}$$

$$= n + m - m * \sum_{\lceil \frac{n}{2} \rceil \le k \le n} C_n^k (p_p * p_v)^{n-k} (1 - p_p * p_v)^k.$$

It is clear that it is better to group all used processors to run one cluster of the next task, and use $m$ new processors to reexecute the first task after the voting among the $n$ results of the first task fails.

## 7 Conclusions

We have proposed a generalized forward recovery checkpointing scheme based on $n$-version voting as acceptance test and $m$-version for reexecution. We evaluated this method, and compared it with the basic checkpointing method. We showed that the $(3, 1)$-pattern improves on the basic method and on the $(2, 1)$-pattern in terms of higher reliability and shorter expected execution time. Some details for tolerating software and/or hardware faults have also been discussed. We discussed how to assign processors after the voting fails to achieve high system reliability; and we determined the number of program versions per task needed to achieve software fault tolerance. For future work, we will introduce recovery length $l$ in the model to further generalize our method into a $(n, m, l)$-pattern.

## References


[1 ] N. S. Bowen and D. K. Pradhan, "Processor-and-Memory-Based Checkpoint Rollback Recovery", *Computer*, Vol. 26, No. 2, pp.22-31, February 1993.

[2 ] C. M. Krishna, G. S. Kang, and Y.-H. Lee, "Optimization Criteria for Checkpoint Placement", *CACM*, Vol. 27, No. 6, pp.1008-1012, October 1984.

[3 ] C. C. Li and W. K. Fuchs, "CATCH: Compiler-Assisted Techniques for Checkpointing", *Proc. 20th Int'l. Symp. on Fault-Tolerant Computing Systems*, pp.74-81, 1990.

[4 ] J. Long, W. K. Fuchs, and J. A. Abraham, "Forward Recovery Using Checkpointing in Parallel Systems", *Proc. Int'l. Conf. Parallel Processing*, pp.272-275, August 1990.

[5 ] J. Long, W. K. Fuchs, and J. A. Abraham, "Implementing Forward Recovery Using Checkpoints in Distributed Systems", *Proc. IFIP Working Conf. on Dependable Comp. for Critical Appl.*, 1991.

[6 ] D. K. Pradhan and N. H. Vaidya, "Roll-Forward Checkpointing Scheme: Concurrent Retry with Nondedicated Spares", *Proc. IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pp.166-173, 1992.

[7 ] A. Tantwai and M. Ruschitzka, "Performance Analysis of Checkpointing Strategies", *ACM Trans. on Computer Systems*, Vol. 2, No. 2, pp.123-144, May 1984

[8 ] A. Ziv and J. Bruck, "Analysis of Checkpointing Schemes for Multiprocessor Systems", *Proc. of the 13th Symp. on Reliable Distributed Systems*, pp. 52-61, 1994


This article was processed using the LaTeX macro package with LLNCS style