# Replicated Process Allocation for Load Distribution in Fault-Tolerant Multicomputers

Jong Kim, *Member*, *IEEE*, Heejo Lee, *Student Member*, *IEEE*, and Sunggu Lee, *Member*, *IEEE*

**Abstract**—In this paper, we consider a load-balancing process allocation method for fault-tolerant multicomputer systems that balances the load before as well as after faults start to degrade the performance of the system. In order to be able to tolerate a single fault, each process (primary process) is duplicated (i.e., has a backup process). The backup process executes on a different processor from the primary, checkpointing the primary process and recovering the process if the primary process fails. In this paper, we formalize the problem of load-balancing process allocation and propose a new process allocation method and analyze the performance of the proposed method. Simulations are used to compare the proposed method with a process allocation method that does not take into account the different load characteristics of the primary and backup processes. While both methods perform well before the occurrence of a fault, only the proposed method maintains a balanced load after the occurrence of such a fault.

**Index Terms**—Backup process, checkpointing, fault-tolerant multicomputer, load balancing, process allocation.

## 1 Introduction

Process allocation in fault-tolerant multicomputer systems has been studied by several researchers [1], [2], [3]. Nieuwenhuis [1] studied transformation rules which transform an allocation of nonduplicated processes into an allocation of duplicated processes. The transformed allocation is proven optimal in terms of reliability. Shatz and Wang [2] proposed a process allocation algorithm which maximizes the reliability of nonhomogeneous systems. Bannister and Trivedi [3] proposed a process allocation algorithm which evenly distributes the load of the system to all nodes. A common assumption of all of the above research works is that each duplicated process is not only a complete replica of the original process, but also has the same execution load as the original. This kind of fault-tolerant process is called an *active* process replica [4].

The fault-tolerant computing process model considered in this paper is the primary-backup process model, which is commonly used in distributed experimental and commercial systems such as Delta-4 and Tandem [5], [6], [7]. In this model, there is a backup copy for each process in the system. However, only one process in a pair is running actively at any one time. The active process is called the primary process and the nonactive process is called the backup (or secondary) process. The active process regularly checkpoints its running state to the backup process. During normal operation, the nonactive backup process is either waiting for a checkpointing message or saving a received checkpointing message. When the node in which the primary process is running becomes faulty, the backup process takes over the role of the primary process. Thus, in order to be able to tolerate faults, primary and backup processes should not be executed on the same node. This kind of fault-tolerant process is called a *passive* process replica [6].

The difference between the computing model considered here and the models considered in other research is in the role of the backup processes. In the previous research, it is assumed that backup processes are exact copies of the primary process. Hence, the load of the backup process is exactly the same as the primary process, whereas the load of the backup process in our model is much less than that of the primary process, i.e., $5 \sim 10$ percent of the load of the primary process.

In this paper, we study the problem of static load-balancing process allocation for fault-tolerant multicomputer systems. Process allocation is called *static* when all processes are allocated at the beginning by the system. *Dynamic* process allocation assumes that processes are allocated as soon as they arrive. In the general computing environment where the occurrence, load, and duration of processes cannot be predicted in advance, dynamic process allocation is more suitable than static process allocation since dynamic process allocation allocates processes as they arrive, taking into consideration the current load of each processor. Static process allocation, on the other hand, can be used most effectively in those systems in which the computational requirement of each process is known beforehand. Examples of such systems include on-line transaction processing and real-time systems, in which most processes are running continuously or repeating in a periodic manner.

Very few works stressed the dynamic process allocation problem in the passive replica computing environment with load balancing. In the distributed system Paralex [8], which supports the fault-tolerance by passive replication, loads are balanced dynamically by the "late binding" of primary processes. Such kind of dynamic allocation is convenient when a system has no previous information about arriving processes and their load information.

The previous static process allocation algorithms [1], [3], which were proposed for the active replica computing environment, exhibit poor performance in the passive replica computing environment since they were designed either to have maximum reliability or to run with the assumption that the load of the backup process is exactly the same as the primary. To achieve load-balancing with the given computing model, we have to consider the distribution of primary and backup processes and the load increment to be added to each nonfaulty node in the event of a fault.

This paper is organized as follows. The next section presents mathematical formulations for the allocation problem and shows that it is NP-hard. In Section 3, basic primitives for allocating processes are addressed and a heuristic process allocation algorithm is given to solve the problem defined in Section 2. Section 4 analyzes the expected performance of the proposed allocation algorithm and compares the performance of the algorithm with another allocation method. Finally, in Section 5, we summarize and discuss the significance of our results.

## 2 Load-Balancing Process Allocation Problem

### 2.1 System Model

The fault-tolerant multicomputer system considered in this paper consists of $n$ nodes (processors). To tolerate a fault, each process is replicated and executed as a pair, referred to as a primary-backup process pair. In this paper, we shall consider the possibility of a single node failure only. However, by using more backup processes, the model can be extended to allow more than one fault [9]. Primary processes can be allocated to any node. However, there is one restriction on the placement of backup process. That is, the primary and backup processes cannot be allocated to the same node. It is assumed that there are $m$ primary processes running in

- *J. Kim and H. Lee are with the Department of Computer Science and Engineering, Pohang University of Science and Technology, San 31 Hyoja Dong, Pohang 790-784, Korea. E-mail:{jkim, heejo}@postech.ac.kr.*
- *S. Lee is with the Department of Electrical Engineering, Pohang University of Science and Technology, San 31 Hyoja Dong, Pohang 790-784, Korea.*

*Manuscript received 31 Aug. 1995; revised 16 Sept. 1996.*
*For information on obtaining reprints of this article, please send e-mail to: transcom@computer.org, and reference IEEECS Log Number C96308.*

0018-9340/97/$10.00 © 1997 IEEE

the system and that the cpu loads of the primary and backup processes are known in advance. This assumption about the load requirement is not unrealistic since many on-line transaction systems run the same processes continuously [5]. Fig. 1 shows a fault-tolerant multicomputer system with processes running on the system.

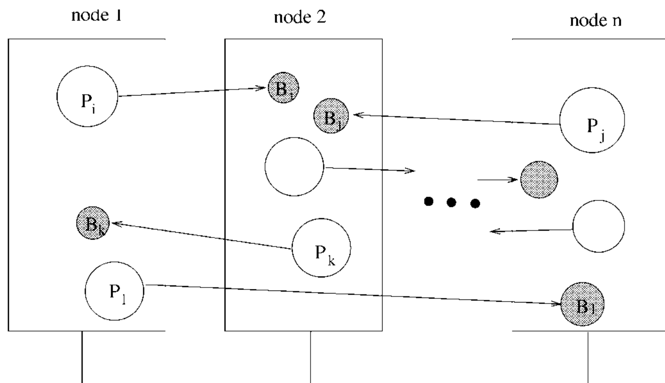


Fig. 1. Primary-backup processes running in the fault-tolerant multicomputer system.

The primary-backup process model considered in this paper is actually used in experimental and commercial systems [5], [6], [7] such as the Tandem Nonstop system [5]. In the Nonstop system, every process has an identical backup process which is allocated to a different node. The backup process is not executed concurrently with the primary process, but is in an inactive mode, prepared to assume the function of the primary process in the event of a primary process failure. To guarantee that the backup process has the information necessary to execute the required function, the primary process sends periodic checkpoint messages to its backup process. In the fault-free situation, the cpu load of the backup processes is much less than that of their respective primaries. The actual load of a backup process is determined by the interval and number of checkpoint messages sent by the primary process. Each backup process has a different percentage of the primary process's cpu load. When a fault occurs, the backup processes of the primary processes which were running on the faulty node take over the role of the primary processes. The backup process is executed continuously starting from the last point at which it received a valid checkpointing message from its primary process. Therefore, the cpu load of the backup process becomes the same as that of its primary.

Load-balancing is required to utilize system resources evenly, thereby enhancing the performance of the system. It has been reported that the approximation algorithm proposed by Bannister and Trivedi [3] has a near-optimal load-balancing result if backup processes have the same cpu load as their primaries. However, in the situation where the cpu load of a backup process is different before and after the occurrence of a fault, the system does not maintain a balanced load after a fault as the cpu loads of the newly activated backup processes increase.

The load-balancing process allocation problem considered here is to find a static process allocation algorithm which balances the cpu load of every node in the fault-free situation and also balances the cpu load when there is a fault in the system. The algorithm should consider the cpu load increment of each node in the event of a fault. In general, it is impossible for all nodes to have exactly the same cpu load. Thus, the quality of the load-balancing is measured either by the deviation from the average of all processors' loads or by the load difference between the node with the highest load and the node with the lightest load. As these parameters approach zero, the system approaches a balanced load. We consider a process allocation to be *optimal* (with respect to load-balancing) if any change in the placement of processes leads to a higher load-balancing measure.

### 2.2 Notation

The following notation is used to formulate the allocation problem.

$n$: number of nodes.
$m$: number of primary processes ($m$ backup processes also exist).
$p_i$: load of primary process $i$.
$b_i$: load of backup process $i$.
$x_{ij}$: 1 if primary process $i$ is allocated to node $j$, 0 otherwise.
$y_{ij}$: 1 if backup process $i$ is allocated to node $j$, 0 otherwise.
$\alpha_j$: 0 if node $j$ has failed, 1 otherwise.
$\tau_j$: load increment to be added to node $j$ when a fault occurs.
$B_{jk}$: the $k$th group of backup processes of the primary processes assigned to node $j$.
$G_{jk}$: the sum of the load differences of the backup group $B_{jk}$.
$P(j)$: total load of node $j$.
$P_{fj}$: set of processors' loads when node $j$ has failed.
$P_{non}$: set of processors' loads with no faulty nodes.
$P_{faulty}$: set of processors' loads with one faulty node.
$\sigma(P)$: standard deviation of load set $P$.
$\Phi(P)$: difference between the maximum and the minimum load of load set $P$.
$\Psi$: the objective cost function to be used.

### 2.3 Formal Problem Description

The load-balancing process allocation problem is represented as a constrained optimization problem. Let us assume that there are $n$ nodes and $m$ processes in the system. Because each primary process has a backup process, the total number of primary and backup processes is $2m$. One prominent constraint is on the allocation of primary-backup process pairs. To be able to tolerate a single fault in a node, a primary process and its backup should not be allocated to the same node. Thus,

$$\sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij} = m, \quad \sum_{i=1}^{m}\sum_{j=1}^{n} y_{ij} = m, \quad \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij}y_{ij} = 0. \tag{2.1}$$

At this point, let us define a set $\mathcal{A}$ with $n + 1$ elements as $\mathcal{A} = \{A_0, A_1, \ldots, A_n\}$. Each element $A_i$ ($0 \le i \le n$) is a processor status vector of the form $[\alpha_1\alpha_2 \ldots \alpha_n]$, where $\alpha_j \in \{0, 1\}$ for all $1 \le j \le n$. $A_0$ is an all-one vector (all nodes are fault-free) and $A_k$ is the vector with $\alpha_k = 0$ and $\alpha_i = 1$ if $i \ne k$ (only node $k$ is faulty).

The load increment to be added to node $j$ in the event of a fault in node $k$ can be represented as

$$\sum_{i=1}^{m} x_{ik}y_{ij}(p_i - b_i).$$

Thus, the total load increment for node $j$ ($\tau_j$) is

$$\tau_j = \sum_{k=1}^{n}\left[(1-\alpha_k)\sum_{i=1}^{m} x_{ik}y_{ij}(p_i - b_i)\right]. \tag{2.2}$$

The load of node $j$, denoted as $P(j)$, is the sum of the load before the fault occurrence and the load increment ($\tau_j$) to be incurred upon the occurrence of a fault.

$$P(j) = \tau_j + \sum_{i=1}^{m}\left(x_{ij}p_i + y_{ij}b_i\right). \tag{2.3}$$

The commonly used metric for evaluating load-balance among nodes is the standard deviation of the processor load [3]. The standard deviation ($\sigma$) of the load when node $k$ is faulty ($A_k$) can be represented as follows:

$$\sigma(A_k) = \sqrt{\frac{1}{n-f}\sum_{i=1}^{n}\left[\alpha_i P(i) - \frac{\alpha_i}{n-f}\sum_{j=1}^{n}\alpha_j P(j)\right]^2} \tag{2.4}$$

where $f$ represents the number of faulty nodes. The second term in the bracket represents the average of $P(j)$.

The number of backup processes becomes less than $m$ when a fault occurs. Therefore, the first and second expressions in (2.1) are no longer valid when a fault occurs. In (2.4), the average load of the live nodes is computed by dividing the total load by the number of live nodes $(n - f)$. Since we consider the possibility of only a single fault, $f$ is either 1 or 0.

Another metric that can be used in the load-balancing process allocation problem is the load difference between the node with the heaviest load and the node with the lightest load. The load difference, denoted as $\Phi$, for the given set of processor loads $P$ can be represented as follows :

$$\Phi(P) = \max(P) - \min(P).$$

The set of processor loads before a failure is represented as $P_{non}$ and the set of processor loads after any node failure is represented as $P_{faulty}$.

In this paper, the load difference is used as the objective cost function because of two reasons. First, the load difference reflects the goal of load-balancing more closely than the standard deviation. This is illustrated by the following two scenarios. In the first scenario, one node has an exceptionally large load and all others are slightly less loaded than the average of all nodes. In the second scenario, some nodes are somewhat more loaded than the average and other nodes are somewhat less loaded than the average. Suppose that the standard deviations of both scenarios are the same, while the load difference are different. In such a case, the latter scenario is more preferable than the former, since there exists one heavily overloaded node in the former scenario. The measurement $\Phi$ will indicate that the latter scenario is better. The second reason is that standard deviation requires more computation than load difference.

The load-balancing process allocation problem is the problem of finding values of $x_{ij}$ and $y_{ij}$ for all possible $i$ and $j$ which minimize the multiple cost functions $\Phi(P_{non})$ and $\Phi(P_{faulty})$ with the constraint given in (2.1). There are two main approaches to solving an optimization problem that involves multiple objective functions [10]. One approach is to solve the problem a number of times with each objective in turn. When solving the problem using one of the objective functions, the other objective functions are considered as constraints. The other approach is to build a suitable linear combination of all the objective functions and optimize the combination function. In this case, it is necessary to attach a weight to each objective function depending on its relative importance. In this paper, the second approach is used to formulate the load-balancing process allocation problem.

We define a new objective function $\Psi$ as

$$\Psi = w_1 \cdot \Phi(P_{non}) + w_2 \cdot \Phi(P_{faulty}), \tag{2.5}$$

where $w_1$ and $w_2$ are the relative weights of importance before and after a fault occurrence, respectively. If we assume that the weights have the same value, i.e., $w_1 = w_2 = 1$, the objective function is

$$\Psi = \Phi(P_{non}) + \Phi(P_{faulty}). \tag{2.6}$$

An optimal allocation is an assignment of processes that minimizes the objective cost function $\Psi$.

### 2.4 Complexity of Optimal Allocation

Let us consider the following two set assignment problems, defined as the *k-grouping problem* and the *restricted grouping problem.*

***k*-Grouping Problem (*k*-GP)**: Given $m$ sets with $k$ positively weighted elements in each set, does there exist an assignment of elements to $k$ groups such that no two elements in the same set are assigned to the same group, and each group has the same weight sum?

**Restricted Grouping Problem (*RGP*)**: Given $n$ sets, $A_i = \{a_{i1}, a_{i2}, \ldots, a_{in}\}$, $i = 1..n$, where $a_{ii}$ is assigned to the group $G_i$, does there exist an assignment of elements to $n$ groups, $G_g = \{a_{1g_1}, a_{2g_2}, \ldots, a_{ng_n}\}$, $g = 1..n$ such that no two elements from the same set are assigned to the same group and each group has the same weight sum?

THEOREM 1. *k-GP is NP-Complete.*

THEOREM 2. *RGP is NP-Complete.*

In this paper, the proofs for all theorems and lemmas have been omitted due to lack of space. The interested reader can obtain the proofs from [11].

The *k*-GP problem can be reduced to the fault-tolerant load-balancing process allocation problem. Each of the $m$ sets contains one primary process and $k - 1$ backup processes, where the weight of each element is the load of each process. Let us change the *k*-GP problem by adding $(n - k)$ sets and changing the grouping requirement so that $n$ groups are formed (recall that $n$ is the number of nodes). In each of the new $(n - k)$ sets, the weight of the first element is equal to the average of the $k$ sets (the sum of the previous $mk$ element weights divided by $k$); the weights of the remaining $k - 1$ elements are assigned to 0. Then, if there is a solution to this modified *k*-GP problem, there is a solution to the original *k*-GP problem. Thus, since the modified *k*-GP problem can be shown to be in NP, the modified *k*-GP problem is NP-complete. This modified *k*-GP problem is equivalent to the load balancing problem with $k - 1$ backup processes. Thus, it follows that the problem of minimizing $\Phi(P_{non})$, is also NP-hard—which in turn implies that the problem of minizing $\Psi$ is NP-hard. Hence, we propose a heuristic approximation algorithm which is cost-effective and results in a well balanced processor load before and after the occurrence of a fault.

## 3 Heuristic Process Allocation Algorithm

In this section, we first present basic primitives for process allocation which form the core of the algorithm. A heuristic algorithm is presented with an example. Then the complexity of the proposed algorithm is analyzed.

### 3.1 Basic Primitives of Process Allocation

In this section, it is assumed that processes are allocated one by one, and, once allocated, processes are not reallocated to other nodes. When we allocate processes one by one, there are three questions we have to consider. First, is it better to assign a process to a lightly loaded node rather than a heavily loaded node to minimize $\Phi$? Second, is it better to assign a process with less load prior to a process with more load to minimize $\Phi$? Third, when there are two ordered sets with $n$ elements and the two sets are combined into one set by adding one element from each set, how should the two sets be combined to minimize $\Phi$? The following lemmas will address these questions one by one.

LEMMA 1. *If* $0 \le P(1) \le P(2) \le \ldots \le P(n)$ *and* $x > 0$, *then* $\Phi(P(1) + x, P(2), \ldots, P(n)) \le \Phi(P(1), P(2), \ldots, P(i) + x, \ldots, P(n))$, *for all i such that* $1 < i \le n$.

Lemma 1 implies that a process should be allocated to a node with the minimum load first to minimize $\Phi$.

The function $Alloc(P, x)$ represents the allocation of $x$ to the most lightly loaded node $P(1)$ when $P$ is an ordered set of $n$ elements, i.e., $P = \{P(i) \mid P(i) \le P(j) \text{ for } 1 \le i \le j \le n\}$.

$$Alloc(P, x) = \{P(1) + x, P(2), \ldots, P(n)\}$$

LEMMA 2. *For a given ordered set P of n elements and* $x \ge y \ge 0$,

$$\Phi(Alloc(Alloc(P, x), y)) \le \Phi(Alloc(Alloc(P, y), x)).$$

Lemma 2 implies that processes with more load should be allocated prior to processes with less load to minimize Φ.

LEMMA 3. *Given two ordered sets of n elements, $P_1$ and $P_2$, Φ is maximally reduced when the two sets are merged by adding the element with the largest weight from one set to the element with the smallest weight from the other set, i.e., the merged unordered set P is* $P = \{P_1(1) + P_2(n), P_1(2) + P_2(n-1), \ldots, P_1(n) + P_2(1)\}$. *Then,* $\Phi(P) \le \max(\Phi(P_1), \Phi(P_2))$.

The above lemma implies that the merging of the two ordered sets as proposed in the lemma does not increase the Φ of two ordered sets.

### 3.2 Two-Stage Allocation

The proposed heuristic algorithm, which satisfies the allocation primitives discussed above, works in two stages. In the first stage, primary processes are allocated using a standard load balancing algorithm. We used a greedy method which allocates the process with the highest load to the node with the lowest load. As shown in Lemmas 1 and 2, this method minimizes Φ. In the second stage, backup processes are allocated considering the load increment to be added to each node in the event of a fault. Let us assume that the algorithm is currently working on node *j*. The backup processes whose primary processes are assigned to node *j* are divided into $(n - 1)$ groups having approximately equal incremental load in the event of a fault in node *j*. These $(n - 1)$ groups of backup processes are finally allocated to the $(n - 1)$ nodes excluding the node *j* based on the actual backup loads before the occurrence of a fault. This allocation balances the total actual load of each node in the fault-free situation. The grouping of backup processes according to the load increment in the case of a fault balances the load if such a fault does occur. The algorithm is formally described below.

**Two-Stage Algorithm**

**Stage-1** *Allocate primary processes*

1) Sort primary processes in descending order of cpu load.
2) Allocate each primary process to the node with the minimum load from the highest load to the lowest.

**Stage-2** *Allocate backup processes*

Do the following steps for each node.

1) Compute the load difference between each primary process and its backup process for all primary processes assigned to the node.
2) Sort, in descending order, the backup processes using the load difference.
3) Divide the backup processes into $(n - 1)$ groups having approximately equal incremental load by assigning each backup process to the group with the smallest load, in the order of the sorted list in the previous step.
4) Computer the actual backup process load of each group.

Do the following for the $n(n - 1)$ backup groups which are generated by the above steps.

1) Sort all $n(n - 1)$ backup groups using the actual loads in descending order.
2) Sort the *n* nodes using their current loads in ascending order.
3) Allocate each backup group to the node with the minimum load. However, if the backup group has a corresponding primary processes in this node or if one of the backup groups which are already allocated to this node comes from the same node as the backup group to be allocated, choose the node with the next-to-the-minimum load.

The allocation of backup processes is the most important part of this algorithm. For each node, the backup processes are first divided into $(n - 1)$ groups using the load difference between each primary process and its backup. This load difference is the amount of load increment to be incurred upon the occurrence of a fault. Next, the algorithm computes the actual load of each group using the actual load of the backup processes. The total number of backup groups is $n(n - 1)$. Each group is assigned to nodes depending on its actual load. When we allocate each backup group, we check whether there is a pre-allocated backup group which comes from the same node as the to-be allocated backup group. In such a case, we select the node with the next-to-the-minimum load.

The purpose of dividing the backup processes into $(n - 1)$ groups for each node is to guarantee that each node has an approximately equal amount of load increment. Hence, the system will have a balanced load when a fault occurs. The purpose of computing the actual load of each group and assigning groups based on their actual loads is to guarantee that each processor's load is balanced before the occurrence of a fault. Therefore, we can balance the processor load before as well as after the occurrence of a fault.

The allocation of $n(n - 1)$ backup process groups for load-balancing is shown in Fig. 2. In this figure, $B_{jk}$ denotes one of the groups of backup processes for the primary processes assigned to node *j*. Hence, $B_{jk}$ $(k = 1..n, k \ne j)$ can be allocated to any node except node *j* to tolerate a fault on the node *j*. The problem of allocating grouped backup processes is the *RGP* problem which was discussed in Section 2.4. Detailed example of running the two-stage algorithm can be found in [12].

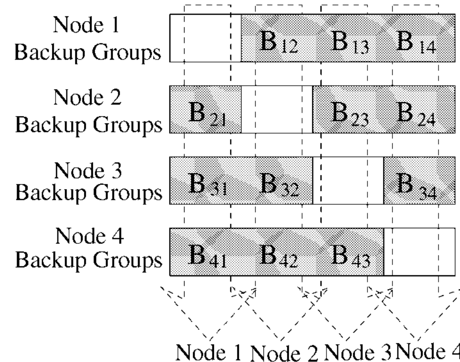


Fig. 2. Backup groups allocation problem.

### 3.3 Algorithm Complexity

The time complexity of each stage of our algorithm is analyzed as follows.

- **primary allocation**
  We can use a general sorting algorithm whose running time is $O(m \log m)$ to sort *m* primary processes. Allocating *m* primary processes to *n* nodes requires $O(m \log n)$ time. (This is a process of updating the highest value *m* times in a priority queue.)
- **backup allocation**
  Let us consider the allocation with the worst execution time in which most of the primary processes are assigned to one node. Sorting the *m* backup processes takes $O(m \log m)$ time. Dividing backup processes into $(n - 1)$ groups having equal incremental load and computing their actual loads takes $O(m \log n + m)$ time. The above grouping is repeated *n* times, which requires $O(n(m \log m + m \log n + m))$ time. Sorting these $n(n - 1)$ backup groups requires $O(n^2 \log n)$ time. Sorting *n* nodes and finding the proper node to allocate one of the $n(n - 1)$ groups to requires $O(n \log n + n)$ time. Thus, allocating $n(n-1)$ groups requires $O(n(n - 1)(n \log n + n))$ time. The time complexity of this stage is thus

$$O(n(m \log m + m \log n + m) + n^2 \log n + n(n-1)(n \log n + n)) = O(nm \log nm + n^3 \log n).$$

Hence, the total time complexity of the two stage allocation algorithm is

$$O(m \log m + m \log n + nm \log nm + n^3 \log n) = O(nm \log nm + n^3 \log n) \quad (3.1)$$

Assuming that the number of processes is much larger than the number of nodes ($m > n^2$), the execution time is bounded by $O(nm \log nm)$. This is a reasonable execution time for process allocation. The time complexity obtained above is based on the worst case scenario in which all primary processes are allocated to one node. If we assume that all primary processes are evenly distributed, the time complexity becomes $O(m \log m)$.

## 4 Performance Analysis

In this section, the performance of the two-stage algorithm is estimated by analysis and compared with related work using simulation.

### 4.1 Expected Performance

The proposed allocation algorithm consists of three parts for load-balancing. First, primary processes are allocated to nodes. Second, the backup processes of each node are grouped on the basis of their primary-backup load differences. Third, the backup process groups are allocated to each node using their actual loads. The following lemmas and theorems estimate the performance of the proposed algorithm.

LEMMA 4. *Given m primary processes with decreasing loads, the load difference between the node with the heaviest load and the node with the lightest load after the allocation of all primary processes by the proposed algorithm is not more than the load of n*th *smallest primary process,* $p_{m-n+1}$.

LEMMA 5. *The load difference between the node with the heaviest load and the node with the lightest load in the allocation of backup processes is bounded by*

$$\frac{\rho}{1-\rho}\left(\frac{1}{n^2(n-1)}\sum_{i=1}^{m}(p_i - b_i) + \max_{i=1..m}(p_i - b_i)\right),$$

*where* $\rho$ *is the maximum ratio of a backup process load to its primary.*

THEOREM 3. *The expected performance of the proposed algorithm before the occurrence of a fault is*

$$\Phi(P_{non}) = \max\left(p_{m-n+1}, \frac{\rho}{1-\rho}\left(\frac{1}{n^2(n-1)}\sum_{i=1}^{m}(p_i - b_i) + \max_{i=1..m}(p_i - b_i)\right)\right).$$

THEOREM 4. *The objective cost function* $\Psi$ *of the proposed algorithm is*

$$\Psi \le \max_{i=1..n}(p_i - b_i) + 2 \cdot \max\left(p_{m-n+1}, \frac{\rho}{1-\rho}\left(\frac{1}{n^2(n-1)}\sum_{i=1}^{m}(p_i - b_i) + \max_{i=1..m}(p_i - b_i)\right)\right).$$

In the next subsection, we will show by comparison with simulation that this bound can serve as a quick approximation to the actual $\Psi$ value.

### 4.2 Performance Comparison

In this subsection, we analyze the performance of the two-stage algorithm using simulations. Note that there has been no previous research on load-balancing with passive replicas. The closest research of this kind is Bannister and Trivedi's work on load-balancing with active replicas [3]. Thus, our algorithm will be compared to Bannister and Trivedi's process allocation (BT) algorithm.

The BT algorithm in the active replica model works as follows. For allocating $n$ processes with $r$ replication, sort all $n$ processes in descending order, and allocate the $r$ replicas of each task to the $r$ least loaded nodes. Process allocation by the BT algorithm is proven to be near-optimal under the assumption that backup processes have exactly the same load as their primaries [3].

The BT algorithm is applied to the problem of load-balancing with passive replicas as follows. First, all primary and backup processes are sorted in descending order of their cpu load. Starting from the highest load process, each process is allocated to the node with the minimum load. When the node with the minimum load in allocating a backup process is the node on which the primary process has already been allocated, the backup process is allocated to the node with the next smallest load.

The environment parameters used in the simulation are as follows. To keep the total load of each node below 100%, the load of the primary processes are chosen randomly in the range of 0.2 to 2 times $100 \cdot (n-1)/m$ based on a uniform distribution. The load of the backup processes are also chosen randomly between 5 ~ 10% of the load of their primaries, also based on a uniform distribution.

Fig. 3 shows each node's load when processes are allocated using the BT algorithm. It is assumed that the number of nodes ($n$) is eight. The horizontal dotted line in the figure represents the average load of a faulty system and the solid line represents the average load of a fault-free system. The figure also shows the load difference of the maximum and the minimum load when the number of primary processes varies from 50 to 300. Fig. 4 shows the simulation results using the two-stage allocation algorithm with the same parameters. When the number of primary processes is 150, the load difference between the maximum and the minimum load is approximately 2 ~ 3% in Fig. 4. By contrast, Fig. 3 shows that the load difference between the maximum and the minimum load is almost 25% of the total load when the BT algorithm is used. The reason that the BT algorithm shows such poor performance is that the allocation of backup processes is done without considering the load variation after the occurrence of a fault.

The load differences $\Phi$ between the minimum and the maximum load before and after the occurrence of a fault is shown in Fig. 5 for each algorithm. These results were obtained with $n = 8$ nodes. The upper two lines show $\Phi(P_{faulty})$ for the two algorithms after a fault has occurred in one node of the system and the lower two lines show $\Phi(P_{non})$ before the occurrence of the fault. It can be seen that before the occurrence of a fault, $\Phi(P_{non})$ using the two-stage allocation algorithm is similar to that of the BT algorithm. However, after the occurrence of a fault, $\Phi(P_{faulty})$ using the two-stage allocation algorithm is significantly less than that using the BT algorithm.

Next, we experimented with the effect of the number of nodes in the system. The simulation was conducted by varying the number of nodes while keeping the number of processes fixed. The simulation results are shown in Fig. 6. In this and the following figures, the simulation results before and after the fault occurrence were combined and represented as $\Psi$, as shown in (2.6). The upper two lines show the simulation results when the number of processes is 200 and the lower two lines show the simulation results when the number of processes is 400. $\Psi$ increases as the number of nodes increases. This result was expected since the number of processes allocated to each node decreases as the number of nodes increases when the number of processes is fixed. This figure again shows that the two-stage allocation algorithm is significantly better than the BT algorithm.

Fig. 7 shows the simulation results when the number of processes is varied from 50 to 300 while the number of nodes is fixed at eight. The upper line shows $\Psi$ when the BT algorithm is used in the simulation, the middle line is the upper bound of our algorithm based on Theorem 4, and the lower line shows the simulated result for the two-stage algorithm. As the number of processes increases, $\Psi$ decreases since the number of processes per node

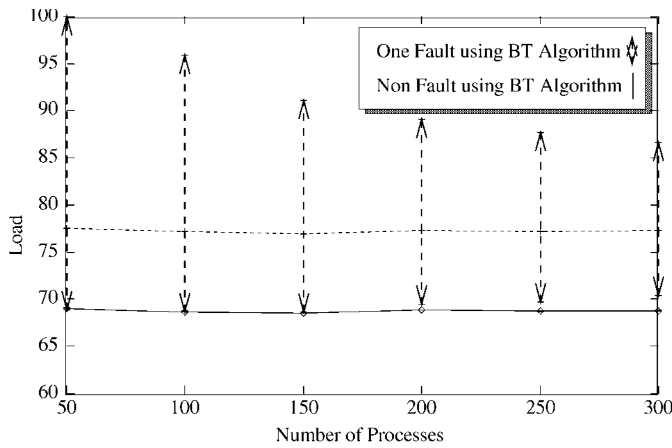


Fig. 3. Minimum, maximum load using the BT algorithm.

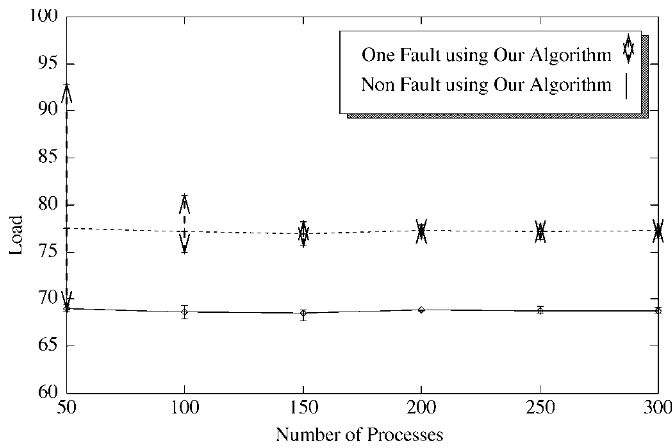


Fig. 4. Minimum, maximum load using our algorithm.

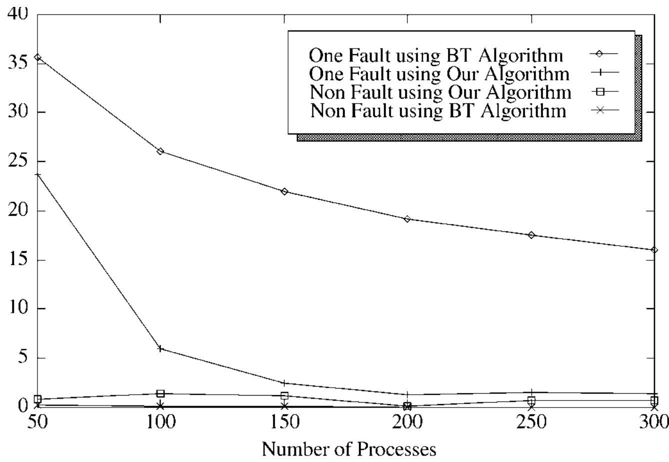


Fig. 5. $\Phi$ before and after a fault occurrence.

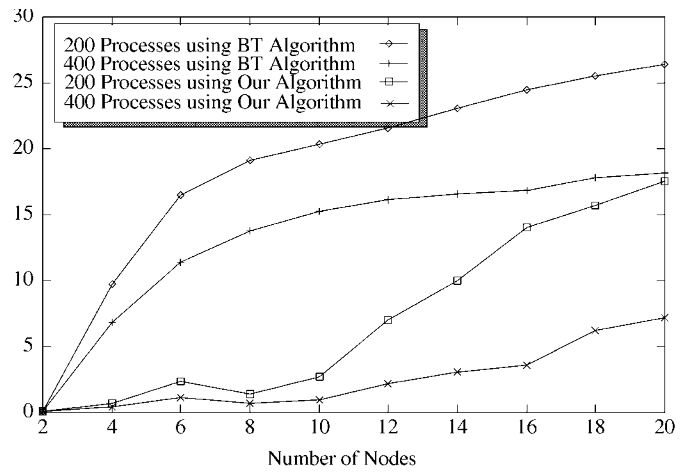


Fig. 6. $\Psi$ affected by the number of nodes.

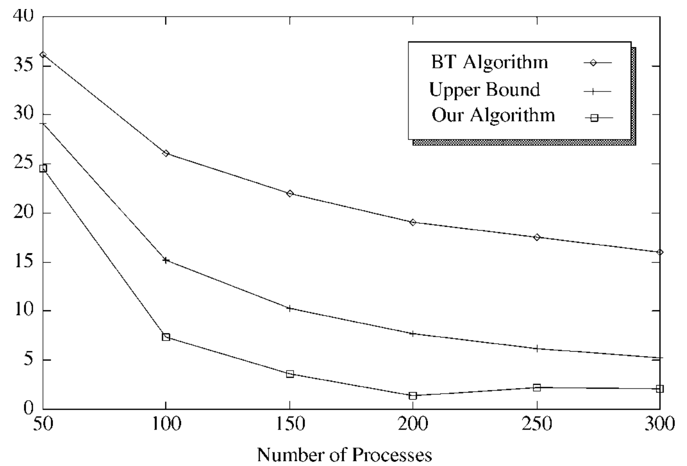


Fig. 7. $\Psi$ affected by the number of processes.

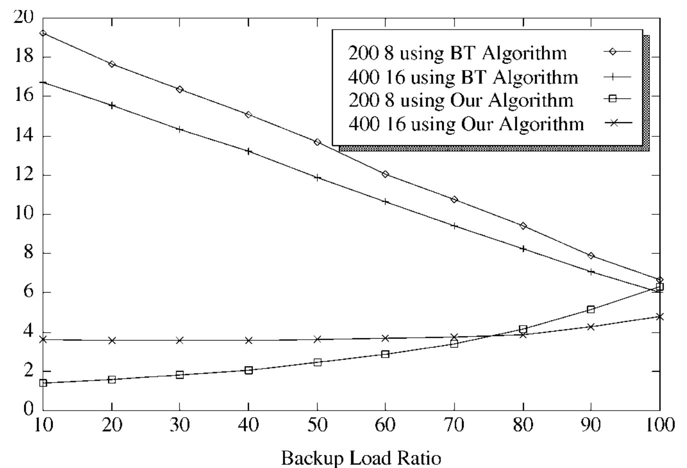


Fig. 8. $\Psi$ affected by varying the load ratio $\rho$ of backup processes.

increases. From this figure, we can confirm that it is easier to balance the load of a system when there are many processes with small loads rather than a few processes with large loads.

In the previous figures, we assumed that the backup processes have 5 ~ 10% of the load of their primary processes. Next, we compare the performance of the two algorithms by varying the load of the backup processes from 10% to 100% of their primaries. For a backup process load ratio $\rho$ in Fig. 8, the load of the backup processes was chosen randomly between $50\rho$ to $100\rho$% of the primary process load in order to provide some variance in the backup process load. We tested for 200 processes with eight nodes, and for 400 processes with 16 nodes. The two-stage allocation algorithm shows better performance in Fig. 8. However, as the load of the backup processes approaches 100% of the primary process load, both algorithms have similar performance. In our simulation, 100% means that a backup process has 50% ~ 100% of its primary process. This result implies that the BT algorithm has better performance when a backup process has the same load of its primary process.

## 5 Summary and Conclusion

A load-balancing static process allocation algorithm for fault-tolerant multicomputer systems is proposed and analyzed. The fault-tolerant process model considered in this paper is the passive process replica model. The passive replica of a process is inactive during normal operation and becomes active only when the main active process becomes faulty or the node on which the active process was running becomes faulty. The load of a passive replica varies before and after the occurrence of a fault. The load-balancing problem with passive replicas is formalized as a constrained optimization problem. Since optimal process allocation is an NP-hard problem, a heuristic approximation algorithm is proposed. The proposed algorithm is compared with the load-balancing algorithm for active replicas proposed by Bannister and Trivedi [3] using simulations. The simulation results show that the proposed algorithm has significantly better performance in the passive replica model. Also, it is shown that the system has a better load balance if we can divide a "big load" process into many "small load" processes.

The main contribution of this paper is the presentation and analysis of a new static load-balancing algorithm for a passive replica fault-tolerant process model. The proposed static process allocation algorithm is applicable to on-line transaction processing and real-time systems. We are currently working on extending this algorithm to handle the dynamic situation. Also, we plan to study the problem of finding an allocation algorithm which guarantees optimal performance and reliability at the same time.

## Acknowledgments

This research was supported in part by KOSEF under Grant 941-0900-055-2 and ETRI under Contract 94231. A preliminary version of this paper was presented at the 25th FTCS.

## References

[1] L.J.M. Nieuwenhuis, "Static Allocation of Process Replicas in Fault-Tolerant Computing Systems," *Proc. FTCS-20*, pp. 298-306, June 1990.

[2] S.M. Shatz, J.P. Wang, and M. Goto, "Task Allocation for Maximizing Reliability of Distributed Computer Systems," *IEEE Trans. Computers*, vol. 41, no. 9, pp. 1,156-1,168, Sept. 1992.

[3] J.A. Bannister and K.S. Trivedi, "Task Allocation in Fault-Tolerant Distributed Systems," *Acata Informatica*, vol. 20, pp. 261-281, 1983.

[4] M. Chereque, D. Powell et al., "Active Replication in Delta-4," *Proc. FTCS-22*, pp. 28-37, June 1992.

[5] D.P. Siewiorek and R.S. Swartz, *Reliable System Design: The Theory and Practice*. New York: Digital Press, 1992.

[6] N. Speirs and P. Barrett, "Using Passive Replicates in Delta-4 to Provide Dependable Distributed Computing," *Proc. FTCS-19*, pp. 184-190, June 1989.

[7] A. Nangia and D. Finkel, "Transaction-Based Fault-Tolerant Computing in Distributed Systems," *Proc. 1992 Workshop Fault-Tolerant Parallel and Distributed Systems*, pp. 92-97, July 1992.

[8] R. Davoli, L.-A. Giachini, Ö. Babaoglu, A. Amoroso, and L. Alvisi, "Parallel Computing in Networks of Workstations with Paralex," *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 4, pp. 371-384, Apr. 1996.

[9] H. Lee, J. Kim, S. Hong et al., "Fault-Tolerant Process Allocation with Load Balancing," *Proc. 1995 Pacific Rim Int'l Symp. Fault-Tolerant Systems*, pp. 124-129, Dec. 1995.

[10] H.P. Williams, *Model Building in Mathematical Programming*, second edition. John Wiley & Sons Ltd., 1985.

[11] J. Kim, H. Lee, and S. Lee, "Load Balancing Process Allocation in Fault-Tolerant Multicomputers," Technical Report CS-95-001, Pohang Univ. of Science and Technology, 1995.

[12] J. Kim, H. Lee, and S. Lee, "Process Allocation for Load Distribution in Fault-Tolerant Multicomputers," *Proc. FTCS-25*, pp. 174-183, June 1995.